

JAVA PROGRAMMING

NOTES



Compile By Arjun

All right belong to www.arjun00.com.np

CHAPTER 1

AN INTRODUCTION TO JAVA

- **In this chapter**
 - 1.1 Java as a Programming Platform
 - 1.2 The Java “White Paper” Buzzwords
 - 1.3 Java Applets and the Internet
 - 1.4 A Short History of Java
- There are lots of programming languages out there, but few of them make much of a splash. Java is a whole *platform*, with a huge library, containing lots of reusable code, and an execution environment that provides services such as security, portability across operating systems, and automatic garbage collection.
- As a programmer, you will want a language with a pleasant syntax and comprehensible semantics (i.e., not C++). Java fits the bill, as do dozens of other fine languages. Some languages give you portability, garbage collection, and the like, but they don’t have much of a library, forcing you to roll your own if you want fancy graphics or networking or database access.
- Well, Java has everything—a good language, a high-quality execution environment, and a vast library. That combination is what makes Java an irresistible proposition to so many programmers.

○ 1.2 THE JAVA “WHITE PAPER” BUZZWORDS

- The authors of Java wrote an influential white paper that explains their design goals and accomplishments. They also published a shorter overview that is organized along the following 11 buzzwords:

- Simple
- Object-Oriented
- Distributed
- Robust
- Secure
- Architecture-Neutral
- Portable
- Interpreted
- High-Performance
- Multithreaded
- Dynamic



- Simple
- Primary characteristics of Java include a simple language that can be programmed without extensive programmer training while being attuned to current software practices. The fundamental concepts of Java are grasped quickly; programmers can be productive from the very beginning.

○ Object-Oriented

- Java is designed to be object oriented from the ground up. Object technology has finally found its way into the programming mainstream after a gestation period of thirty years. The needs of distributed, client-server based systems coincided with the encapsulated, message-passing paradigms of object-based software. To function within increasingly complex, network-based environments, programming systems must adopt object-oriented concepts. Java provides a clean and efficient object-based development environment

○ Distributed

- *Java has an extensive library of routines for coping with TCP/IP protocols like HTTP and FTP. Java applications can open and access objects across the Net via URLs with the same ease as when accessing a local file system.*

○ Robust

- Java is designed for creating highly reliable software. It provides extensive compile-time checking, followed by a second level of run-time checking. Language features guide programmerstowards reliable programming habits.
- The Java compiler detects many problems that in other languages would show up only at runtime. As for the second point, anyone who has spent hours chasing memory corruption caused by a pointer bug will be very happy with this aspect of Java.

○ Secure

- Java is designed to operate in distributed environments, which means that security is of paramount importance. With security features designed into the language and run-time system,Java lets you construct applications that can't beinvaded from outside. In the networked environment, applications written in Java are secure from intrusion by unauthorized code attempting to get behind the scenes and create viruses or invade file systems.

○ Architecture-Neutral

- Java is designed to support applications that will be deployed into heterogeneous networked environments. In such environments, applications must be capable of executing on a variety of hardware architectures. Within this variety of hardware platforms, applications must execute atop a variety of operating systems and interoperate with multiple programming language interfaces. To accommodate the diversity of operating environments, the Java compiler generates bytecodes—an architecture neutral intermediate format designed to transport code efficiently to multiple hardware and software platforms.

○ Portable

- Architecture neutrality is just one part of a truly portable system. Java takes portability a stage further by being strict in its definition of the basic language. Java puts a stake in the ground and specifies the sizes of its basic data types and the behavior of its arithmetic operators. Your programs are the same on every platform—there are no data type incompatibilities across hardware and software architectures.

- The architecture-neutral and portable language environment of Java is known as the Java Virtual Machine. It's the specification of an abstract machine for which Java language compilers can generate code.
- Interpreted
 - *The Java interpreter can execute Java bytecodes directly on any machine to which the interpreter has been ported. Since linking is a more incremental and lightweight process, the development process can be much more rapid and exploratory.*
- High Performance:
 - Performance is always a consideration. Java achieves superior performance by adopting a scheme by which the interpreter can run at full speed without needing to check the run-time environment. The automatic garbage collector runs as a low-priority background thread, ensuring a high probability that memory is available when required, leading to better performance.

Applications requiring large amounts of compute power can be designed such that compute- intensive sections can be rewritten in native machinecode as required and interfaced with the Java environment. In general, users perceive that interactive applications respond quickly even though they're interpreted

○ **Multithreaded**

- *[The] benefits of multithreading are better interactive responsiveness and real-time behavior.*
- It was the first mainstream language to support concurrent programming.
- At the time, multicore processors were exotic, but web programming had just started, and processors spent a lot of time waiting for a response from the server. Concurrent programming was needed to make sure the user interface didn't freeze. Concurrent programming is never easy, but Java has done a very good job making it manageable.

○ **Dynamic**

- *In a number of ways, Java is a more dynamic language than C or C++. It was designed to adapt to an evolving environment. Libraries can freely add new methods and instance variables without any effect on their clients. In Java, finding out runtime type information is straightforward.*

○ **JAVA APPLETS AND THE INTERNET**

- The idea here is simple: Users will download Java bytecodes from the Internet and run them on their own machines. Java programs that work on web pages are called *applets*. To use an applet, you only need a Java-enabled web browser, which will execute the bytecodes for you. You need not install any software. You get the latest version of the program whenever you visit the web page containing the applet. Most importantly, thanks to the security of the virtual machine, you never need to worry about attacks from hostile code.
- Inserting an applet into a web page works much like embedding an image. The applet becomes a part of the page, and the text flows around the space used for the applet. The point is, this image is *alive*. It reacts to user commands, changes its appearance, and exchanges data between the computer presenting the applet and the computer serving it.

(UNIT-II)

Fundamental Programming Structures

- 2.1. Writing Comments
- 2.2. Basic Data Types
- 2.3. Variables and Constants
- 2.4. Operators
- 2.5. Type Casting
- 2.6. Control Flow
- 2.7. Arrays

Comments

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

Types of Java Comments

There are two types of comments in java.

1. Single Line Comment
2. Multi Line Comment

Documentation Comment

1) Java Single Line Comment

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

Syntax:

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

Example:

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

Output:

10

2) Java Multi Line Comment

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

Syntax:

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

Example:

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

Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

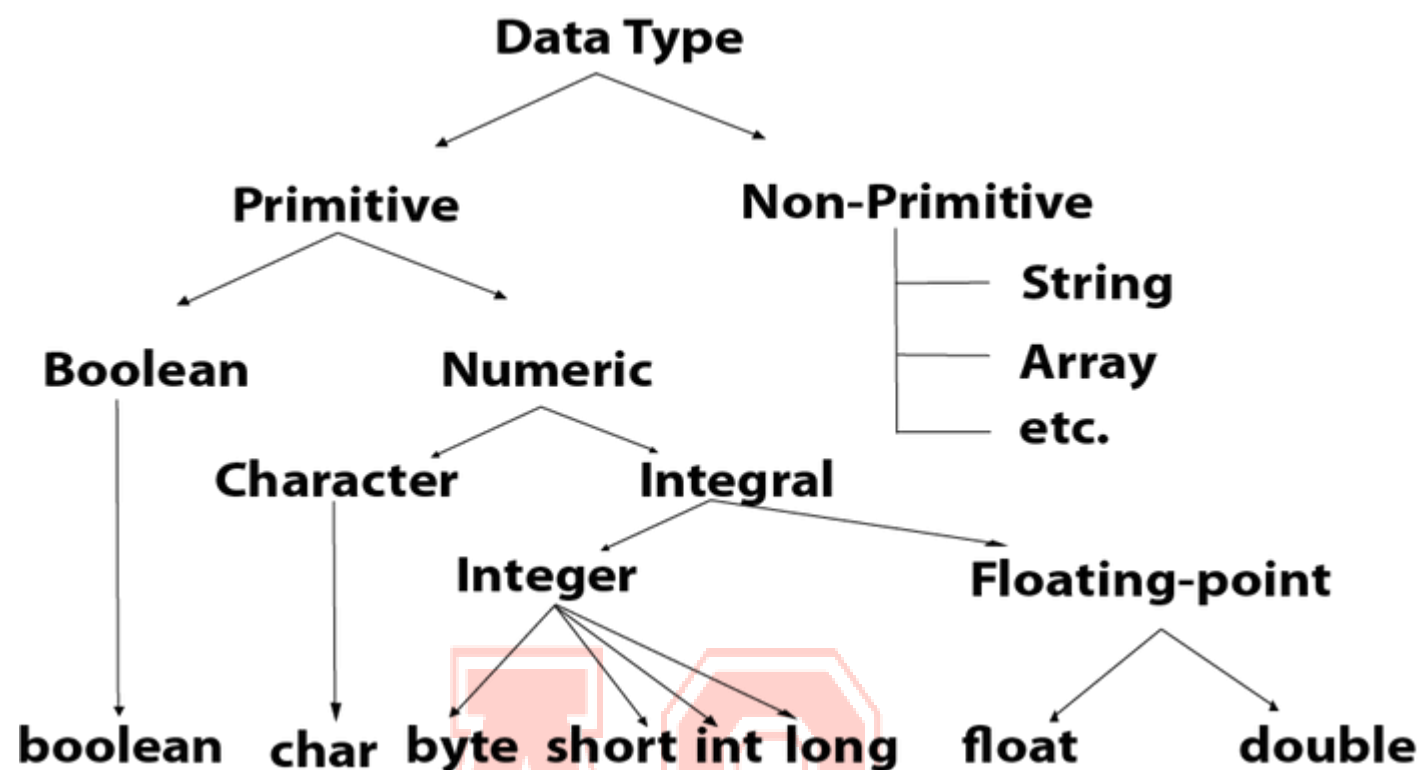
Primitive data types: The primitive data types include boolean, char, byte, short, int, long, float and double.

Non-primitive data types: The non-primitive data types include Classes, Interfaces, and Arrays.
Java Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

There are 8 types of primitive data types:

1. boolean data type
2. byte data type
3. char data type
4. short data type
5. int data type
6. long data type
7. float data type
8. double data type



Data Type	Default Value	Default size
Boolean	False	1 bit
Char	'\u0000'	2 byte
Byte	0	1 byte
Short	0	2 byte
Int	0	4 byte
Long	0L	8 byte
Float	0.0f	4 byte
Double	0.0d	8 byte

Boolean Data Type

The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.

The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

Example: Boolean one = false

Byte Data Type

The byte data type is an example of primitive data type. It is an 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.

The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

Example: byte a = 10, byte b = -20

Short Data Type

The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.

The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

Example: short s = 10000, short r = -5000

Int Data Type

The int data type is a 32-bit signed two's complement integer. Its value-range lies between -2,147,483,648 (-2^{31}) to 2,147,483,647 ($2^{31} - 1$) (inclusive). Its minimum value is -2,147,483,648 and maximum value is 2,147,483,647. Its default value is 0.

The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

Example: int a = 100000, int b = -200000

Long Data Type

The long data type is a 64-bit two's complement integer. Its value-range lies between -9,223,372,036,854,775,808 (-2^{63}) to 9,223,372,036,854,775,807 ($2^{63} - 1$) (inclusive). Its minimum value is -9,223,372,036,854,775,808 and maximum value is 9,223,372,036,854,775,807. Its default value is 0. The long data type is used when you need a range of values more than those provided by int.

Example: long a = 100000L, long b = -200000L

Float Data Type

The float data type is a single-precision 32-bit IEEE 754 floating point. Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.

Example: float f1 = 234.5f

Double Data Type

The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

Example: double d1 = 12.3

Char Data Type

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive). The char data type is used to store characters.

Example: char letterA = 'A'

Why char uses 2 byte in java and what is \u0000 ?

It is because java uses Unicode system not ASCII code system. The \u0000 is the lowest range of Unicode system.

Variables and Types

2.1 Variables - Name, Type and Value

Computer programs manipulate (or process) data. A *variable* is used to *store a piece of data* for processing. It is called *variable* because you can change the value stored.

More precisely, a *variable* is a *named* storage location, that stores a *value* of a particular data *type*. In other words, a *variable* has a *name*, a *type* and stores a *value*.

A variable has a *name* (aka *identifier*), e.g., radius, area, age, height and numStudents. The name is needed to *uniquely* identify and reference each variable. You can use the *name* to assign a value to the variable (e.g., radius = 1.2), and to retrieve the value stored (e.g., radius*radius*3.1419265).

A variable has a *data type*. The frequently-used Java *data types* are:

int: meant for integers (whole numbers) such as 123 and -456.

double: meant for floating-point number (real numbers) having an optional decimal point and fractional part, such as 3.1416, -55.66, 1.2e3, or -4.5E-6, where e or E denotes exponent of base 10.

String: meant for texts such as "Hello" and "Good Morning!". Strings are enclosed within a pair of double quotes.

char: meant for a single character, such as 'a', '8'. A char is enclosed by a pair of single quotes.

In Java, you need to declare the *name* and the *type* of a variable before using a variable. For examples,

```
int sum;    // Declare an "int" variable named "sum"
double average; // Declare a "double" variable named "average"
String message; // Declare a "String" variable named "message"
char grade; // Declare a "char" variable named "grade"
```

A variable can store a *value* of the declared data *type*. It is important to take note that a variable in most programming languages is associated with a type, and can only store value of that particular type. For example, an int variable can store an integer value such as 123, but NOT floating-point number such as 12.34, nor string such as "Hello".

The concept of *type* was introduced in the early programming languages to *simplify* interpretation of data made up of binary sequences (0's and 1's). The type determines the size and layout of the data, the range of its values, and the set of operations that can be applied.

The following diagram illustrates three types of variables: int, double and String. An int variable stores an integer (or whole number or fixed-point number); a double variable stores a floating-point number (or real number); a String variable stores texts.

TYPE	NAME	VALUE	
int	number	1	Stored only Integer
int	sum	500500	Stored only Integer
double	radius	5.5	Stored only floating-point number
double	area	95.0334	Stored only floating-point number
String	greeting	Hello	Stored only texts
String	statusMsg	Game Over	Stored only texts

*A variable has a **name**, stores a **value** of the declared **type**.*

Identifiers (or Names)

An *identifier* is needed to *name* a variable (or any other entity such as a method or a class). Java imposes the following *rules on identifiers*:

An identifier is a sequence of characters, of any length, comprising uppercase and lowercase letters (a-z, A-Z), digits (0-9), underscore (_), and dollar sign (\$).

White space (blank, tab, newline) and other special characters (such as +, -, *, /, @, &, commas, etc.) are not allowed. Take note that blank and dash (-) are not allowed, i.e., "max value" and "max-value" are not valid names. (This is because blank creates two tokens and dash crashes with minus sign!)

An identifier must begin with a letter (a-z, A-Z) or underscore (_). It cannot begin with a digit (0-9) (because that could confuse with a number). Identifiers begin with dollar sign (\$) are reserved for system-generated entities.

An identifier cannot be a reserved keyword or a reserved literal (e.g., class, int, double, if, else, for, true, false, null).

Identifiers are case-sensitive. A rose is NOT a Rose, and is NOT a ROSE.

Examples: abc, _xyz, \$123, _1_2_3 are valid identifiers. But 1abc, min-value, surface area, ab@c are NOT valid identifiers.

Variable Naming Convention

A variable name is a noun, or a noun phrase made up of several words with no spaces between words. The first word is in lowercase, while the remaining words are initial-capitalized. For examples, radius, area, fontSize, numStudents, xMax, yMin, xTopLeft, isValidInput, and thisIsAVeryLongVariableName. This convention is also known as *camel-case*.

Recommendations

It is important to choose a name that is *self-descriptive* and closely reflects the meaning of the variable, e.g., numberOfStudents or numStudents, but not n or x, to store the number of students. It is alright to use abbreviations.

Do not use *meaningless* names like a, b, c, i, j, k, n, i1, i2, i3, j99, exercise85 (what is the purpose of this exercise?), and example12 (What is this example about?).

Avoid *single-letter* names like i, j, k, a, b, c, which are easier to type but often meaningless.

Exceptions are common names like x, y, z for coordinates, i for index. Long names are harder to type, but self-document your program. (I suggest you spend sometimes practicing your typing.)

Use *singular* and *plural* nouns prudently to differentiate between singular and plural variables. For example, you may use the variable row to refer to a single row number and the variable rows to refer to many rows (such as an array of rows - to be discussed later).

2.2 Variable Declaration

To use a variable in your program, you need to first *introduce* it by *declaring* its *name* and *type*, in one of the following syntaxes. The act of declaring a variable allocates a storage of size capable of holding a value of the type.

Syntax	Example
<pre>// Declare a variable of a specified type type identifier;</pre>	<pre>int sum; double average; String statusMsg; int number, count; double sum, difference, product, quotient; String helloMsg, gameOverMsg; int magicNumber = 99; double pi = 3.14169265; String helloMsg = "hello,"; int sum = 0, product = 1; double height = 1.2; length = 3.45; String greetingMsg = "hi!", quitMsg = "bye!";</pre>
<pre>// Declare multiple variables of the SAME type, // separated by commas type identifier1, identifier2, ..., identifierN; // Declare a variable and assign an initial value type identifier = initialValue;</pre>	
<pre>// Declare multiple variables of the SAME type, // with initial values type identifier1 = initValue1, ..., identifierN = initValueN;</pre>	

2.4 Constants (**final** variables)

Constants are *non-modifiable (immutable)* variables, declared with keyword final. You can only assign values to final variables ONCE. Their values cannot be changed during program execution.

For examples:

```
final double PI = 3.14159265; // Declare and initialize the constant
```

```
final int SCREEN_X_MAX = 1280;
```

```
SCREEN_X_MAX = 800; // compilation error: cannot assign a value to final variable
```

```
// You can only assign value to final variables ONCE
```

```
final int SCREEN_Y_MIN;
```



```
SCREEN_Y_MIN = 0; // First assignment
```

```
SCREEN_Y_MIN = 10; // compilation error: variable might already have been assigned
```

Constant Naming Convention: Use uppercase words, joined with underscore. For example, MIN_VALUE, MAX_SIZE, and INTEREST_RATE_6_MTH.

Operators in Java

Operator in [Java](#) is a symbol which is used to perform operations. For example: +, -, *, / etc. There are many types of operators in Java which are given below:

3. Unary Operator,
4. Arithmetic Operator,
5. Shift Operator,
6. Relational Operator,
7. Bitwise Operator,
8. Logical Operator,
9. Ternary Operator and
10. Assignment Operator.

Java Operator Precedence

Operator Type	Category	Precedence
Unary	Postfix	<i>expr</i> ++ <i>expr</i> --
	Prefix	++ <i>expr</i> -- <i>expr</i> + <i>expr</i> - <i>expr</i> ~ !
Arithmetic	Multiplicative	* / %
	Additive	+ -
Shift	Shift	<< >> >>>
Relational	Comparison	< > <= >= instanceof
	Equality	== !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	
Logical	logical AND	&&
	logical OR	
Ternary	Ternary	? :
Assignment	Assignment	= += -= *= /= %= &= ^= = <<= >>= >>>=

Java Unary Operator

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

incrementing/decrementing a value by one

negating an expression

inverting the value of a boolean

Java Unary Operator Example: ++ and --

```
class OperatorExample{
public static void main(String args[]){
int x=10;
System.out.println(x++); //10 (11)
System.out.println(++x); //12
System.out.println(x--); //12 (11)
System.out.println(--x); //10
}}
```

Output:

```
10
12
12
10
```

Java Unary Operator Example 2: ++ and --

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=10;
System.out.println(a++ + ++a); //10+12=22
System.out.println(b++ + b++); //10+11=21
}}
```

Output:

```
22
21
```

Java Unary Operator Example: ~ and !

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=-10;
boolean c=true;
boolean d=false;
System.out.println(~a); // -11 (minus of total positive value which starts from 0)
System.out.println(~b); // 9 (positive of total minus, positive starts from 0)
System.out.println(!c); // false (opposite of boolean value)
System.out.println(!d); // true
}}
```

Output:

```
-11
9
false
```

true

Java Arithmetic Operators

Java arithmetic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

Java Arithmetic Operator Example

```
class OperatorExample{  
public static void main(String args[]){  
int a=10;  
int b=5;  
System.out.println(a+b);//15  
System.out.println(a-b);//5  
System.out.println(a*b);//50  
System.out.println(a/b);//2  
System.out.println(a%b);//0  
}}
```

Output:

15
5
50
2
0

Java Arithmetic Operator Example: Expression

```
class OperatorExample{  
public static void main(String args[]){  
System.out.println(10*10/5+3-1*4/2);  
}}
```

Output:

21

Java Left Shift Operator

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

Java Left Shift Operator Example

```
class OperatorExample{  
public static void main(String args[]){  
System.out.println(10<<2);//10*2^2=10*4=40  
System.out.println(10<<3);//10*2^3=10*8=80  
System.out.println(20<<2);//20*2^2=20*4=80  
System.out.println(15<<4);//15*2^4=15*16=240  
}}
```

Output:

40

80
80
240

Java Right Shift Operator

The Java right shift operator >> is used to move left operands value to right by the number of bits specified by the right operand.

Java Right Shift Operator Example

```
class OperatorExample{
public static void main(String args[]){
System.out.println(10>>2); //10/2^2=10/4=2
System.out.println(20>>2); //20/2^2=20/4=5
System.out.println(20>>3); //20/2^3=20/8=2
}}
```

Output:

2
5
2

Java Shift Operator Example: >> vs >>>

```
class OperatorExample{
public static void main(String args[]){
//For positive number, >> and >>> works same
System.out.println(20>>2);
System.out.println(20>>>2);
//For negative number, >>> changes parity bit (MSB) to 0
System.out.println(-20>>2);
System.out.println(-20>>>2);
}}
```

Output:

5
5
-5
1073741819

Java AND Operator Example: Logical && and Bitwise &

The logical && operator doesn't check second condition if first condition is false. It checks second condition only if first one is true.

The bitwise & operator always checks both conditions whether first condition is true or false.

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&& a<c); //false && true = false
}
```

```
System.out.println(a<b&a<c);//false & true = false
}}
Output:
false
false
```

Java AND Operator Example: Logical && vs Bitwise &

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a++<c);//false && true = false
System.out.println(a);//10 because second condition is not checked
System.out.println(a<b&a++<c);//false && true = false
System.out.println(a);//11 because second condition is checked
}}
Output:
false
10
false
11
```

Java OR Operator Example: Logical || and Bitwise |

The logical || operator doesn't check second condition if first condition is true. It checks second condition only if first one is false.

The bitwise | operator always checks both conditions whether first condition is true or false.

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a>b||a<c);//true || true = true
System.out.println(a>b|a<c);//true | true = true
//|| vs |
System.out.println(a>b||a++<c);//true || true = true
System.out.println(a);//10 because second condition is not checked
System.out.println(a>b|a++<c);//true | true = true
System.out.println(a);//11 because second condition is checked
}}
Output:
true
true
true
10
true
11
```

Java Ternary Operator

Java Ternary operator is used as one liner replacement for if-then-else statement and used a lot in Java programming. it is the only conditional operator which takes three operands.

Java Ternary Operator Example

```
class OperatorExample{
public static void main(String args[]){
int a=2;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}}
Output:
2
```

Another Example:

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}}
Output:
5
```

Java Assignment Operator

Java assignment operator is one of the most common operator. It is used to assign the value on its right to the operand on its left.

Java Assignment Operator Example

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=20;
a+=4; //a=a+4 (a=10+4)
b-=4; //b=b-4 (b=20-4)
System.out.println(a);
System.out.println(b);
}}
Output:
14
16
```

Java Assignment Operator Example

```
class OperatorExample{
public static void main(String[] args){
int a=10;
a+=3; //10+3
```

```

System.out.println(a);
a-=4;//13-4
System.out.println(a);
a*=2;//9*2
System.out.println(a);
a/=2;//18/2
System.out.println(a);
}

```

Output:

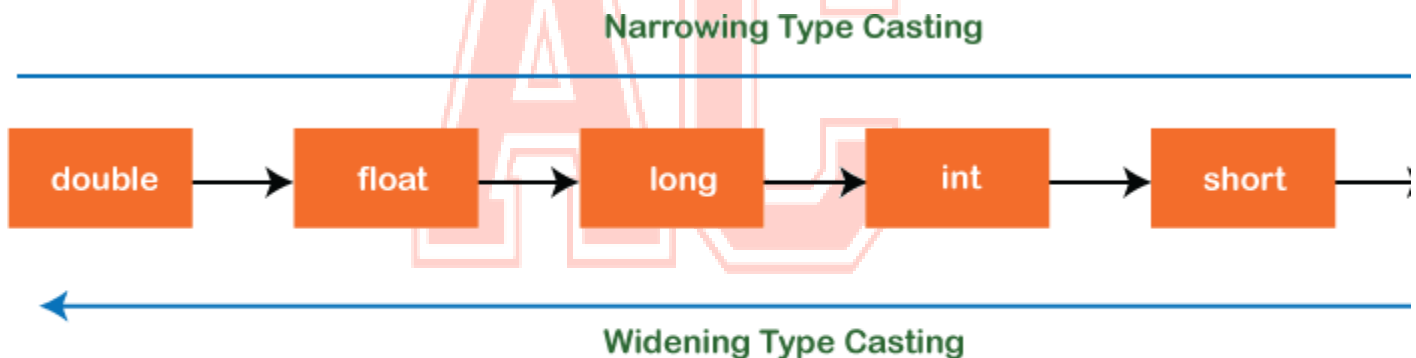
```

13
9
18
9

```

Type Casting in Java

In Java, **type casting** is a method or process that converts a data type into another data type in both ways manually and automatically. The **automatic conversion** is done by the compiler and manual conversion performed by the programmer. In this section, we will discuss **type casting** and its **types** with proper examples.



Type Casting in Java

Type casting

Convert a value from one data type to another data type is known as **type casting**.

Types of Type Casting

There are two types of type casting:

1. Widening Type Casting
2. Narrowing Type Casting

Widening Type Casting

Converting a lower data type into a higher one is called **widening** type casting. It is also known as **implicit conversion** or **casting down**. It is done automatically. It is safe because there is no chance to lose data. It takes place when:

Both data types must be compatible with each other.

The target type must be larger than the source type.

byte -> **short** -> **char** -> **int** -> **long** -> **float** -> **double**

For example, the conversion between numeric data type to char or Boolean is not done automatically. Also, the char and Boolean data types are not compatible with each other. Let's see an example.

WideningTypeCastingExample.java

```
public class WideningTypeCastingExample
{
    public static void main(String[] args)
    {
        int x = 7;
        //automatically converts the integer type into long type
        long y = x;
        //automatically converts the long type into float type
        float z = y;
        System.out.println("Before conversion, int value "+x);
        System.out.println("After conversion, long value "+y);
        System.out.println("After conversion, float value "+z);
    }
}
```

Output

Before conversion, the value is: 7

After conversion, the long value is: 7

After conversion, the float value is: 7.0

In the above example, we have taken a variable x and converted it into a long type. After that, the long type is converted into the float type.

Narrowing Type Casting

Converting a higher data type into a lower one is called **narrowing** type casting. It is also known as **explicit conversion** or **casting up**. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

double -> **float** -> **long** -> **int** -> **char** -> **short** -> **byte**

Let's see an example of narrowing type casting.

In the following example, we have performed the narrowing type casting two times. First, we have converted the double type into long data type after that long data type is converted into int type.

NarrowingTypeCastingExample.java

```
public class NarrowingTypeCastingExample
{
    public static void main(String args[])
    {
        double d = 166.66;
        //converting double data type into long data type
    }
}
```



```

long l = (long)d;
//converting long data type into int data type
int i = (int)l;
System.out.println("Before conversion: "+d);
//fractional part lost
System.out.println("After conversion into long type: "+l);
//fractional part lost
System.out.println("After conversion into int type: "+i);
}
}

```

Output

Before conversion: 166.66
 After conversion into long type: 166
 After conversion into int type: 166

Control Flow Statement:

Java If-else Statement

The Java if statement is used to test the condition. It checks boolean condition: *true* or *false*. There are various types of if statement in Java.

1. if statement
2. if-else statement
3. if-else-if ladder
4. nested if statement

Java if Statement

The Java if statement tests the condition. It executes the *if block* if condition is true.

Syntax:

```

if(condition){
//code to be executed
}

```

Example:

```

//Java Program to demonstrate the use of if statement.
public class IfExample {
public static void main(String[] args) {
//defining an 'age' variable
int age=20;
//checking the age
if(age>18){
System.out.print("Age is greater than 18");
}
}
}

```

Output:

Age is greater than 18

Java if-else Statement:

The java if-else statement also tests the condition. It executes the if block if condition is true.

Syntax

```
if(condition){  
    //code if condition is true  
}else{  
    //code if condition is false  
}
```

Example:

//A Java Program to demonstrate the use of if-else statement.

//It is a program of odd and even number.

```
public class IfElseExample {  
    public static void main(String[] args) {  
        //defining a variable  
        int number=13;  
        //Check if the number is divisible by 2 or not  
        if(number%2==0){  
            System.out.println("even number");  
        }else{  
            System.out.println("odd number");  
        }  
    }  
}
```

Output:

odd number

Leap Year Example:

A year is leap, if it is divisible by 4 and 400. But, not by 100.

```
public class LeapYearExample {  
    public static void main(String[] args) {  
        int year=2020;  
        if(((year % 4 ==0) && (year % 100 !=0)) || (year % 400 ==0)){  
            System.out.println("LEAP YEAR");  
        }  
        else{  
            System.out.println("COMMON YEAR");  
        }  
    }  
}
```

Output:

LEAP YEAR

Using Ternary Operator

We can also use ternary operator (?:) to perform the task of if...else statement. It is a shorthand way to check the condition. If the condition is true, the result of ? is returned. But, if the condition is false, the result of : is returned.

Example:

```
public class IfElseTernaryExample {
```

```

public static void main(String[] args) {
    int number=13;
    //Using ternary operator
    String output=(number%2==0)?"even number":"odd number";
    System.out.println(output);
}

```

Output:

odd number

Java if-else-if ladder Statement

The if-else-if ladder statement executes one condition from multiple statements.

Syntax:

```

if(condition1){
    //code to be executed if condition1 is true
}else if(condition2){
    //code to be executed if condition2 is true
}
else if(condition3){
    //code to be executed if condition3 is true
}
...
else{
    //code to be executed if all the conditions are false
}

```

Example:

//Java Program to demonstrate the use of If else-if ladder.

//It is a program of grading system for fail, D grade, C grade, B grade, A grade and A+.

```

public class IfElseIfExample {
public static void main(String[] args) {
    int marks=65;

    if(marks<50){
        System.out.println("fail");
    }
    else if(marks>=50 && marks<60){
        System.out.println("D grade");
    }
    else if(marks>=60 && marks<70){
        System.out.println("C grade");
    }
    else if(marks>=70 && marks<80){
        System.out.println("B grade");
    }
    else if(marks>=80 && marks<90){
        System.out.println("A grade");
    }
    else if(marks>=90 && marks<100){
        System.out.println("A+ grade");
    }
    else{

```


Output:

You are eligible to donate blood

Example 2:

//Java Program to demonstrate the use of Nested If Statement.

```
public class JavaNestedIfExample2 {
    public static void main(String[] args) {
        //Creating two variables for age and weight
        int age=25;
        int weight=48;
        //applying condition on age and weight
        if(age>=18){
            if(weight>50){
                System.out.println("You are eligible to donate blood");
            } else{
                System.out.println("You are not eligible to donate blood");
            }
        } else{
            System.out.println("Age must be greater than 18");
        }
    }
}
```

Output:

You are not eligible to donate blood

Java Switch Statement

The Java *switch statement* executes one statement from multiple conditions. It is like if-else-if ladder statement. The switch statement works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long. Since Java 7, you can use strings in the switch statement.

In other words, the switch statement tests the equality of a variable against multiple values.

Syntax

```
switch(expression){
    case value1:
        //code to be executed;
        break; //optional
    case value2:
        //code to be executed;
        break; //optional
    .....
    default:
        code to be executed if all cases are not matched;
}
```

Example:

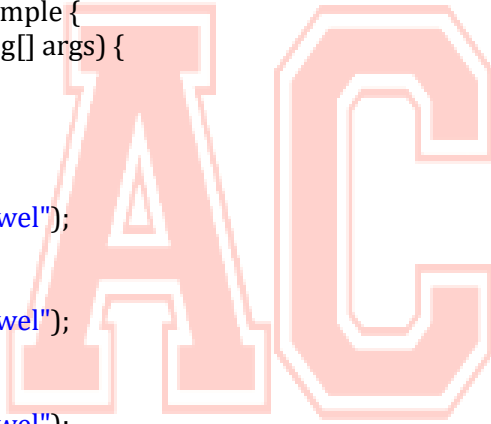
```
public class SwitchExample {
    public static void main(String[] args) {
        //Declaring a variable for switch expression
        int number=20;
        //Switch expression
    }
}
```

```
switch(number){  
    //Case statements  
    case 10: System.out.println("10");  
    break;  
    case 20: System.out.println("20");  
    break;  
    case 30: System.out.println("30");  
    break;  
    //Default case statement  
    default: System.out.println("Not in 10, 20 or 30");  
}  
}  
}  
Output:  
20
```

Program to check Vowel or Consonant:

If the character is A, E, I, O, or U, it is vowel otherwise consonant. It is not case-sensitive.

```
public class SwitchVowelExample {  
    public static void main(String[] args) {  
        char ch='O';  
        switch(ch)  
        {  
            case 'a':  
                System.out.println("Vowel");  
                break;  
            case 'e':  
                System.out.println("Vowel");  
                break;  
            case 'i':  
                System.out.println("Vowel");  
                break;  
            case 'o':  
                System.out.println("Vowel");  
                break;  
            case 'u':  
                System.out.println("Vowel");  
                break;  
            case 'A':  
                System.out.println("Vowel");  
                break;  
            case 'E':  
                System.out.println("Vowel");  
                break;  
            case 'I':  
                System.out.println("Vowel");  
                break;  
            case 'O':  
                System.out.println("Vowel");  
                break;  
            case 'U':  
                System.out.println("Vowel");  
                break;  
            default:  
                System.out.println("Consonant");  
        }  
    }  
}
```



```

        System.out.println("Vowel");
        break;
    case 'U':
        System.out.println("Vowel");
        break;
    default:
        System.out.println("Consonant");
    }
}
}
}
Output:
Vowel

```

Java Switch Statement is fall-through

The Java switch statement is fall-through. It means it executes all statements after the first match if a break statement is not present.

Example:

```

//Java Switch Example where we are omitting the
//break statement
public class SwitchExample2 {
    public static void main(String[] args) {
        int number=20;
        //switch expression with int value
        switch(number){
            //switch cases without break statements
            case 10: System.out.println("10");
            case 20: System.out.println("20");
            case 30: System.out.println("30");
            default: System.out.println("Not in 10, 20 or 30");
        }
    }
}
}
Output:
20
30
Not in 10, 20 or 30

```

Loops in Java

In programming languages, loops are used to execute a set of instructions/functions repeatedly when some conditions become true. There are three types of loops in Java.

1. for loop
2. while loop
3. do-while loop

Java For Loop vs While Loop vs Do While Loop

Comparison	for loop	while loop	do while loop
Introduction	The Java for loop is a control flow statement that iterates a part of the programs multiple times.	The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition.	The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition.
When to use	If the number of iteration is fixed, it is recommended to use for loop.	If the number of iteration is not fixed, it is recommended to use while loop.	If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop.
Syntax	<pre>for(init;condition;incr/decr){ // code to be executed }</pre>	<pre>while(condition){ //code to be executed }</pre>	<pre>do{ //code to be executed }while(condition);</pre>
Example	<pre>//for loop for(int i=1;i<=10;i++){ System.out.println(i); }</pre>	<pre>//while loop int i=1; while(i<=10){ System.out.println(i); i++; }</pre>	<pre>//do-while loop int i=1; do{ System.out.println(i); i++; }while(i<=10);</pre>
Syntax for infinitive loop	<pre>for(;;){ //code to be executed }</pre>	<pre>while(true){ //code to be executed }</pre>	<pre>do{ //code to be executed }while(true);</pre>

Java For Loop

The Java *for loop* is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop.

There are three types of for loops in java.

Simple For Loop

For-each or Enhanced For Loop

Labeled For Loop

Java Simple For Loop

A simple for loop is the same as C/C++. We can initialize the [variable](#), check condition and increment/decrement value. It consists of four parts:

Initialization: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.

Condition: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.

Statement: The statement of the loop is executed each time until the second condition is false.

Increment/Decrement: It increments or decrements the variable value. It is an optional condition.

Syntax:

```
for(initialization;condition;incr/decr){  
    //statement or code to be executed  
}
```

Example:

//Java Program to demonstrate the example of for loop

//which prints table of 1

```
public class ForExample {  
    public static void main(String[] args) {  
        //Code of Java for loop  
        for(int i=1;i<=10;i++){  
            System.out.println(i);  
        }  
    }  
}
```

Output:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```



Java Nested For Loop

If we have a for loop inside the another loop, it is known as nested for loop. The inner loop executes completely whenever outer loop executes.

Example:

```
public class NestedForExample {  
    public static void main(String[] args) {  
        //loop of i  
        for(int i=1;i<=3;i++){  
            //loop of j  
            for(int j=1;j<=3;j++){  
                System.out.println(i+ " "+j);  
            }  
        }  
    }  
}
```

Output:

1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3

Pyramid Example 1:

```
public class PyramidExample {  
    public static void main(String[] args) {  
        for(int i=1;i<=5;i++){  
            for(int j=1;j<=i;j++){  
                System.out.print(" * ");  
            }  
            System.out.println();//new line  
        }  
    }  
}
```

Output:

*
* *
* * *
* * * *
* * * * *

Pyramid Example 2:

```
public class PyramidExample2 {  
    public static void main(String[] args) {  
        int term=6;  
        for(int i=1;i<=term;i++){  
            for(int j=term;j>=i;j--){  
                System.out.print(" * ");  
            }  
            System.out.println();//new line  
        }  
    }  
}
```

Output:

* * * * *
* * * *
* * * *
* * *
* *
*
*

Java for-each Loop

The for-each loop is used to traverse array or collection in java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation.

It works on elements basis not index. It returns element one by one in the defined variable.

Syntax:

```
for(Type var:array){  
    //code to be executed  
}
```

Example:

```
//Java For-each loop example which prints the  
//elements of the array
```

```
public class ForEachExample {  
    public static void main(String[] args) {  
        //Declaring an array  
        int arr[]={12,23,44,56,78};  
        //Printing array using for-each loop  
        for(int i:arr){  
            System.out.println(i);  
        }  
    }  
}
```

Output:

```
12  
23  
44  
56  
78
```

Java Labeled For Loop

We can have a name of each Java for loop. To do so, we use label before the for loop. It is useful if we have nested for loop so that we can break/continue specific for loop.

Usually, break and continue keywords breaks/continues the innermost for loop only.

Syntax:

```
labelname:  
for(initialization;condition;incr/decr){  
    //code to be executed  
}
```

Example:

```
//A Java program to demonstrate the use of labeled for loop
```

```
public class LabeledForExample {  
    public static void main(String[] args) {  
        //Using Label for outer and for loop  
        aa:  
        for(int i=1;i<=3;i++){  
            bb:  
            for(int j=1;j<=3;j++){  
                if(i==2&& j==2){  
                    break aa;  
                }  
            }  
        }  
    }  
}
```

```

        System.out.println(i+ " "+j);
    }
}
}

```

Output:

```

1 1
1 2
1 3
2 1

```

If you use **break bb**, it will break inner loop only which is the default behavior of any loop.

```

public class LabeledForExample2 {
public static void main(String[] args) {

```

```

    aa:

```

```

        for(int i=1;i<=3;i++){

```

```

            bb:

```

```

                for(int j=1;j<=3;j++){

```

```

                    if(i==2&&j==2){

```

```

                        break bb;

```

```

                    }
                    System.out.println(i+ " "+j);

```

```

                }

```

```

            }

```

```

        }

```

```

    }

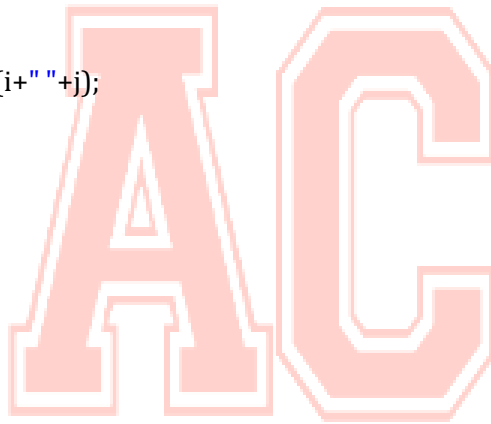
```

Output:

```

1 1
1 2
1 3
2 1
3 1
3 2
3 3

```



Java Infinite For Loop

If you use two semicolons ;; in the for loop, it will be infinite for loop.

Syntax:

```

for(;;){

```

```

    //code to be executed

```

```

}

```

Example:

```

//Java program to demonstrate the use of infinite for loop

```

```

//which print an statement

```

```

public class ForExample {

```

```

public static void main(String[] args) {

```

```

    //Using no condition in for loop

```

```

    for(;;){

```

```

        System.out.println("infinite loop");

```

```
}  
}  
}
```

Output:

```
infinite loop  
infinite loop  
infinite loop  
infinite loop  
infinite loop  
ctrl+c
```

Java While Loop

The [Java while loop](#) is used to iterate a part of the [program](#) several times. If the number of iteration is not fixed, it is recommended to use while [loop](#).

Syntax:

```
while(condition){  
//code to be executed  
}  
public class WhileExample {  
public static void main(String[] args) {  
    int i=1;  
    while(i<=10){  
        System.out.println(i);  
        i++;  
    }  
}  
}
```

Output:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

Java do-while Loop

The Java *do-while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

The Java *do-while loop* is executed at least once because condition is checked after loop body.

Syntax:

```
do{  
//code to be executed  
}while(condition);
```

Example:

```

public class DoWhileExample {
public static void main(String[] args) {
    int i=1;
    do{
        System.out.println(i);
        i++;
    }while(i<=10);
}
}

```

Output:

```

1
2
3
4
5
6
7
8
9
10

```

Java Break Statement

When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

The Java *break* statement is used to break loop or [switch](#) statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.

We can use Java break statement in all types of loops such as [for loop](#), [while loop](#) and [do-while loop](#).

Syntax:

jump-statement;

break;

Java Break Statement with Loop

Example:

//Java Program to demonstrate the use of break statement

//inside the for loop.

```

public class BreakExample {
public static void main(String[] args) {
    //using for loop
    for(int i=1;i<=10;i++){
        if(i==5){
            //breaking the loop
            break;
        }
        System.out.println(i);
    }
}
}

```

Output:

1
2
3
4

Java Break Statement with Inner Loop

It breaks inner loop only if you use break statement inside the inner loop.

Example:

//Java Program to illustrate the use of break statement

//inside an inner loop

```
public class BreakExample2 {  
    public static void main(String[] args) {  
        //outer loop  
        for(int i=1;i<=3;i++){  
            //inner loop  
            for(int j=1;j<=3;j++){  
                if(i==2&&j==2){  
                    //using break statement inside the inner loop  
                    break;  
                }  
                System.out.println(i+" "+j);  
            }  
        }  
    }  
}
```

Output:

1 1
1 2
1 3
2 1
3 1
3 2
3 3

Java Continue Statement

The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately. It can be used with for loop or while loop.

The Java *continue statement* is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition. In case of an inner loop, it continues the inner loop only.

We can use Java continue statement in all types of loops such as for loop, while loop and do-while loop.

Syntax:

jump-statement;

continue;

Java Continue Statement Example

Example:

```
//Java Program to demonstrate the use of continue statement  
//inside the for loop.
```

```
public class ContinueExample {  
public static void main(String[] args) {  
    //for loop  
    for(int i=1;i<=10;i++){  
        if(i==5){  
            //using continue statement  
            continue;//it will skip the rest statement  
        }  
        System.out.println(i);  
    }  
}  
}
```

Output:

```
1  
2  
3  
4  
6  
7  
8  
9  
10
```

As you can see in the above output, 5 is not printed on the console. It is because the loop is continued when it reaches to 5.

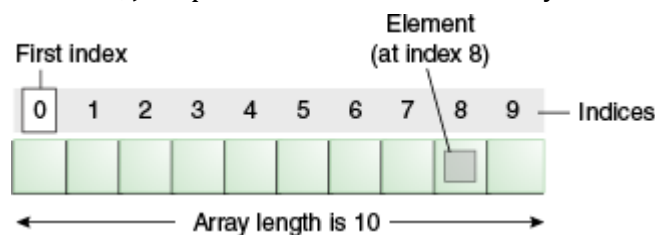
Java Arrays

Normally, an array is a collection of similar type of elements which has contiguous memory location.

Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.



Advantages

Code Optimization: It makes the code optimized, we can retrieve or sort the data efficiently.

Random access: We can get any data located at an index position.

Disadvantages

Size Limit: We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Types of Array in java

There are two types of array.

Single Dimensional Array

Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

dataType[] arr; (or)

dataType []arr; (or)

dataType arr[];

Instantiation of an Array in Java

arrayRefVar=**new** datatype[size];

Example of Java Array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

//Java Program to illustrate how to declare, instantiate, initialize

//and traverse the Java array.

```
class Testarray{
    public static void main(String args[]){
        int a[]=new int[5]; //declaration and instantiation
        a[0]=10; //initialization
        a[1]=20;
        a[2]=70;
        a[3]=40;
        a[4]=50;
        //traversing array
        for(int i=0;i<a.length;i++) //length is the property of array
            System.out.println(a[i]);
    }
}
```

Output:

10

20

70

40

50

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

```
int a[]={33,3,4,5}; //declaration, instantiation and initialization
```

Let's see the simple example to print this array.

//Java Program to illustrate the use of declaration, instantiation

//and initialization of Java array in a single line

```
class Testarray1{
    public static void main(String args[]){
        int a[]={33,3,4,5}; //declaration, instantiation and initialization
        //printing array
        for(int i=0;i<a.length;i++) //length is the property of array
```

```
System.out.println(a[i]);
}}
```

Output:

```
33
3
4
5
```

For-each Loop for Java Array

We can also print the Java array using **for-each loop**. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop. The syntax of the for-each loop is given below:

```
for(data_type variable:array){
//body of the loop
}
```

Let us see the example of print the elements of Java array using the for-each loop.

//Java Program to print the array elements using for-each loop

```
class Testarray1{
public static void main(String args[]){
int arr[]={33,3,4,5};
//printing array using for-each loop
for(int i:arr)
System.out.println(i);
}}
```

Output:

```
33
3
4
5
```

Passing Array to a Method in Java

We can pass the java array to method so that we can reuse the same logic on any array. Let's see the simple example to get the minimum number of an array using a method.

//Java Program to demonstrate the way of passing an array
//to method.

```
class Testarray2{
//creating a method which receives an array as a parameter
static void min(int arr[]){
int min=arr[0];
for(int i=1;i<arr.length;i++)
if(min>arr[i])
min=arr[i];
```

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

```
public static void main(String args[]){
int a[]={33,3,4,5};//declaring and initializing an array
```

```
min(a);//passing array to method
}}
Output:
3
```

Anonymous Array in Java

Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.

//Java Program to demonstrate the way of passing an anonymous array
//to method.

```
public class TestAnonymousArray{
//creating a method which receives an array as a parameter
static void printArray(int arr[]){
for(int i=0;i<arr.length;i++)
System.out.println(arr[i]);
}
```

```
public static void main(String args[]){
printArray(new int[]{10,22,44,66});//passing anonymous array to method
}}
```

Output:

```
10
22
44
66
```

Returning Array from the Method

We can also return an array from the method in Java.

//Java Program to return an array from the method

```
class TestReturnArray{
//creating method which returns an array
static int[] get(){
return new int[]{10,30,50,90,60};
}
```

```
public static void main(String args[]){
//calling method which returns an array
int arr[]=get();
//printing the values of an array
for(int i=0;i<arr.length;i++)
System.out.println(arr[i]);
}}
```

Output:

```
10
30
50
90
60
```

ArrayIndexOutOfBoundsException

The Java Virtual Machine (JVM) throws an ArrayIndexOutOfBoundsException if length of the array is negative, equal to the array size or greater than the array size while traversing the array.

//Java Program to demonstrate the case of
//ArrayIndexOutOfBoundsException in a Java Array.

```
public class TestArrayException{
    public static void main(String args[]){
        int arr[]={50,60,70,80};
        for(int i=0;i<=arr.length;i++){
            System.out.println(arr[i]);
        }
    }
}
```

Output:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
at TestArrayException.main(TestArrayException.java:5)

50
60
70
80

Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java

```
dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[];
```

Example to instantiate Multidimensional Array in Java

```
int[][] arr=new int[3][3]; //3 row and 3 column
```

Example to initialize Multidimensional Array in Java

```
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

Example of Multidimensional Java Array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

//Java Program to illustrate the use of multidimensional array

```
class Testarray3{
    public static void main(String args[]){
        //declaring and initializing 2D array
        int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
        //printing 2D array
```

```

for(int i=0;i<3;i++){
    for(int j=0;j<3;j++){
        System.out.print(arr[i][j]+" ");
    }
    System.out.println();
}
}
}

```

Output:

```

1 2 3
2 4 5
4 4 5

```

Jagged Array in Java

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

//Java Program to illustrate the jagged array

```

class TestJaggedArray{
    public static void main(String[] args){
        //declaring a 2D array with odd columns
        int arr[][] = new int[3][];
        arr[0] = new int[3];
        arr[1] = new int[4];
        arr[2] = new int[2];
        //initializing a jagged array
        int count = 0;
        for (int i=0; i<arr.length; i++){
            for(int j=0; j<arr[i].length; j++){
                arr[i][j] = count++;
            }

            //printing the data of a jagged array
            for (int i=0; i<arr.length; i++){
                for (int j=0; j<arr[i].length; j++){
                    System.out.print(arr[i][j]+" ");
                }
                System.out.println();//new line
            }
        }
    }
}

```

Output:

```

0 1 2
3 4 5 6
7 8

```

Addition of 2 Matrices in Java

Let's see a simple example that adds two matrices.

//Java Program to demonstrate the addition of two matrices in Java

```

class Testarray5{
    public static void main(String args[]){
        //creating two matrices

```



```

int b[][]={{1,1,1},{2,2,2},{3,3,3}};

//creating another matrix to store the multiplication of two matrices
int c[][]=new int[3][3]; //3 rows and 3 columns

//multiplying and printing multiplication of 2 matrices
for(int i=0;i<3;i++){
for(int j=0;j<3;j++){
c[i][j]=0;
for(int k=0;k<3;k++)
{
c[i][j]+=a[i][k]*b[k][j];
} //end of k loop
System.out.print(c[i][j]+" "); //printing matrix element
} //end of j loop
System.out.println();//new line
}
}}

```

Output:
6 6 6
12 12 12
18 18 18



Objects and Classes

- 3.1. An Introduction to Object Oriented Programming
- 3.2. Using Predefined Classes
- 3.3. Defining Your Own Class
- 3.4. Static Fields and Methods
- 3.5. Method Parameters
- 3.6. Object Construction
- 3.7. Packages

What is OOP?

OOP stands for **Object-Oriented Programming**.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

OOP is faster and easier to execute

OOP provides a clear structure for the programs

OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug

OOP makes it possible to create full reusable applications with less code and shorter development time

OOPs (Object-Oriented Programming System)

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

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Object

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

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.



Class

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

A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

Inheritance

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.



Polymorphism

If *one task is performed in different ways*, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

In Java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

Abstraction

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

In Java, we use abstract class and interface to achieve abstraction.



Capsule

Encapsulation

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

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

The Concept of Java Classes and Objects

Classes and objects are the two most essential Java concepts that every programmer must learn. Classes and objects are closely related and work together. An object has behaviors and states, and is an instance of class. For instance, a cat is an object—it's color and size are states, and its meowing and clawing furniture are behaviors. A class models the object, a blueprint or template that describes the state or behavior supported by objects of that type.

What Is a Class?

We can define a class as a container that stores the data members and methods together. These data members and methods are common to all the objects present in a particular package.

Every class we use in Java consists of the following components, as described below:

Access Modifier

Object-oriented programming languages like Java provide the programmers with four types of access modifiers.

1. Public
2. Private
3. Protected
4. Default

These access modifiers specify the accessibility and users permissions of the methods and members of the class.

Class Name

This describes the name given to the class that the programmer decides on, according to the predefined naming conventions.

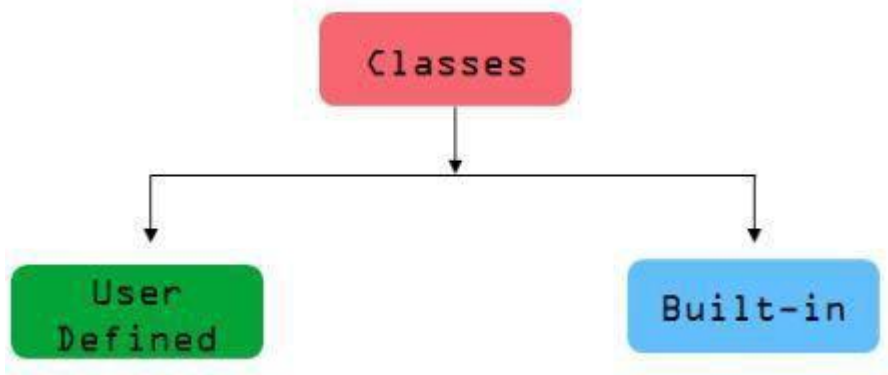
Body of the Class

The body of the class mainly includes executable statements.

Apart from these, a class may include keywords like "super" if you use a superclass, "implements" if you are inheriting members, methods, and instances from the same class, and "interface" if you are inheriting any members, methods, and instances from a different class.

Type of Classes

In Java, we classify classes into two types:



Built-in Classes

Built-in classes are just the predefined classes that come along with the Java Development Kit (JDK). We also call these built-in classes libraries. Some examples of built-in classes include:

java.lang.System
java.util.Date
java.util.ArrayList
java.lang.Thread

Predefined Classes and methods

Lots of classes and methods are already predefined by the time you start writing your own code: some already written by other programmers in your team many predefined packages, classes, and methods come from the Java Library.

Library: collection of packages

Package: contains several classes

class: contains several methods

Method: a set of instructions

Using Predefined Classes and Methods

To use a method, you must know:

- Name of the **class** containing the method (like **Math**)
- Name of the package containing the **class** (like **java.lang**)
- Name of the method (like **pow**) and list of parameters

```
import java.lang.*; // imports package
Math.pow( 2, 3 ); // calls power method in class Math
Math.pow( x, y ); // another call
```

java.lang.String

String class will be the undisputed champion on any day by popularity and none will deny that. This is a final class and used to create / operate immutable string literals. It was available from JDK 1.0

java.lang.System

Usage of System depends on the type of project you work on. You may not be using it in your project but still it is one of the popular java classes around. This is a utility class and cannot be instantiated. Main uses of this class are access to standard input, output, environment variables, etc. Available since JDK 1.0

java.lang.Exception

Throwable is the super class of all Errors and Exceptions. All abnormal conditions that can be handled comes

under Exception. [NullPointerException](#) is the most popular among all the exceptions. Exception is at top of hierarchy of all such exceptions. Available since JDK 1.0

java.util.ArrayList

An implementation of array data structure. This class implements List interface and is the most popular member or java collections framework. [Difference between ArrayList and Vector](#) is one popular topic among the beginners and frequently asked question in java interviews. It was introduced in JDK 1.2

java.util.HashMap

An implementation of a key-value pair data structure. This class implements Map interface. As similar to ArrayList vs Vector, we have HashMap vs [Hashtable](#) popular comparisons. This happens to be a popular collection class that acts as a container for property-value pairs and works as a transport agent between multiple layers of an application. It was introduced in JDK 1.2

java.lang.Object

Great grandfather of all java classes. Every java class is a subclass of Object. It will be used often when we work on a platform/framework. It contains the important methods like equals, hashCode, clone, toString, etc. It is available from day one of java (JDK 1.0)

java.lang.Thread

A thread is a single sequence of execution, where multiple thread can co-exist and share resources. We can extend this Thread class and create our own threads. Using Runnable is also another option. Usage of this class depends on the domain of your application. It is not absolutely necessary to build a usual application. It was available from JDK 1.0

java.lang.Class

Class is a direct subclass of Object. There is no constructor in this class and their objects are loaded in JVM by [classloaders](#). Most of us may not have used it directly but I think its an essential class. It is an important class in doing reflection. It is available from JDK 1.0

java.util.Date

This is used to work with date. Sometimes we feel that this class should have added more utility methods and we end up creating those. Every enterprise application we create has a date utility. Introduced in JDK 1.0 and later made huge changes in JDK1.1 by deprecating a whole lot of methods.

java.util.Iterator

This is an interface. It is very popular and came as a replacement for Enumeration. It is a simple to use convenience utility and works in sync with Iterable. It was introduced in JDK 1.2

Using predefined class name as Class or Variable name in Java

In Java, Using predefined class name as Class or Variable name is allowed. However, According to Java Specification Language (§3.9) the basic rule for naming in Java is that you cannot use a keyword as name of a class, name of a variable nor the name of a folder used for package.

Using any predefined class in Java won't cause such compiler error as Java predefined classes are not keywords.

Following are some invalid variable declarations in Java:

boolean break = false; // not allowed as break is keyword

int boolean = 8; // not allowed as boolean is keyword

boolean goto = false; // not allowed as goto is keyword

String final = "hi"; // not allowed as final is keyword

Using predefined class name as User defined class name

Question : Can we have our class name as one of the predefined class name in our program?

Answer : Yes we can have it. Below is example of using **Number** as user defined class

```
// Number is predefined class name in java.lang package
// Note : java.lang package is included in every java
program by default
public class Number
```

```

{
    public static void main (String[] args)
    {
        System.out.println("It works");
    }
}

```

Output:

It works

Using String as User Defined Class:

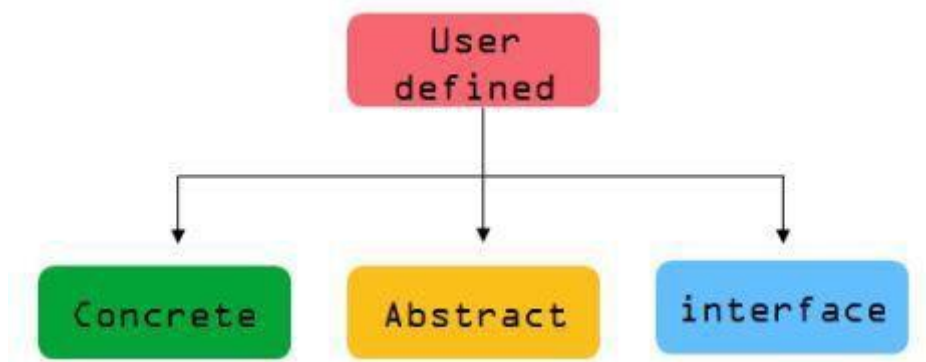
```

// String is predefined class name in java.lang package
// Note : java.lang package is included in every java
program by default
public class String
{
    public static void main (String[] args)
    {
        System.out.println("I got confused");
    }
}

```

User-Defined Classes

User-defined classes are rather self-explanatory. The name says it all. They are classes that the user defines and manipulates in the real-time programming environment. User-defined classes are broken down into three types:



Concrete Class

Concrete class is just another standard class that the user defines and stores the methods and data members in.

Syntax:

```

class con{
    //class body;
}

```

Abstract Class

Abstract classes are similar to concrete classes, except that you need to define them using the "abstract" keyword. If you wish to instantiate an abstract class, then it should include an abstract method in it. Otherwise, it can only be inherited.

Syntax:

```
abstract class AbstClas{
    //method();
    abstract void demo();
}
```

Interfaces

Interfaces are similar to classes. The difference is that while class describes an object's attitudes and behaviors, interfaces contain the behaviors a class implements. These interfaces will only include the signatures of the method but not the actual method.

Syntax:

```
public interface demo{
    public void signature1();
    public void signature2();
}
public class demo2 implements demo{
    public void signature1(){
        //implementation;
    }
    public void signature2(){
        //implementation;
    }
}
```

Rules for Creating Classes

The following rules are mandatory when you're working with Java classes:

1. The keyword "class" must be used to declare a class
2. Every class name should start with an upper case character, and if you intend to include multiple words in a class name, make sure you use the camel case
3. A Java project can contain any number of default classes but should not hold more than one public class
4. You should not use special characters when naming classes
5. You can implement multiple interfaces by writing their names in front of the class, separated by commas
6. You should expect a Java class to inherit only one parent class

What is an Object in Java?

An object in Java is the most fundamental unit of the object-oriented programming paradigm. It includes the real-world entities and manipulates them by invoking methods. An object in Java consists of the following:

- Identity
- Behavior
- State

Identity

This is the unique name given by the user that allows it to interact with other objects in the project.

Example:

Name of the student

Behavior

The behavior of an object is the method that you declare inside it. This method interacts with other objects present in the project.

Example:
Studying, Playing, Writing

State

The parameters present in an object represent its state based on the properties reflected by the parameters of other objects in the project.

Example:
Section, Roll number, Percentage

Key Differences Between Java Classes and Objects

The key differences between a class and an object are:

Class:

A class is a blueprint for creating objects
A class is a logical entity
The keyword used is "class"
A class is designed or declared only once
The computer does not allocate memory when you declare a class

Objects:

An object is a copy of a class
An object is a physical entity
The keyword used is "new"
You can create any number of objects using one single class
The computer allocates memory when you declare a class

Create a Class

To create a class, use the keyword **class**:

Main.java

Create a class named "**Main**" with a variable x:

```
public class Main {  
    int x = 5;  
}
```

Create an Object

In Java, an object is created from a class. We have already created the class named **MyClass**, so now we can use this to create objects.

To create an object of **MyClass**, specify the class name, followed by the object name, and use the keyword **new**:

Example

Create an object called "**myObj**" and print the value of x:

```
public class Main {  
    int x = 5;  
  
    public static void main(String[] args) {  
        Main myObj = new Main();  
        System.out.println(myObj.x);  
    }  
}
```

Multiple Objects

You can create multiple objects of one class:

Example

Create two objects of **Main**:

```
public class Main {
    int x = 5;

    public static void main(String[] args) {
        Main myObj1 = new Main(); // Object 1
        Main myObj2 = new Main(); // Object 2
        System.out.println(myObj1.x);
        System.out.println(myObj2.x);
    }
}
```

Using Multiple Classes

You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the attributes and methods, while the other class holds the `main()` method (code to be executed)).

Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory/folder:

Main.java

Second.java

Main.java

```
public class Main {
    int x = 5;
}
```

Second.java

```
class Second {
    public static void main(String[] args) {
        Main myObj = new Main();
        System.out.println(myObj.x);
    }
}
```

When both files have been compiled:

C:\Users\Your Name>javac Main.java

C:\Users\Your Name>javac Second.java

Run the Second.java file:

C:\Users\Your Name>java Second

And the output will be:

5

Java Class Attributes

In the previous chapter, we used the term "variable" for `x` in the example (as shown below). It is actually an **attribute** of the class. Or you could say that class attributes are variables within a class:

Example

Create a class called "Main" with two attributes: `x` and `y`:

```
public class Main {
    int x = 5;
    int y = 3;
}
```

Accessing Attributes

You can access attributes by creating an object of the class, and by using the dot syntax (`.`):

The following example will create an object of the `Main` class, with the name `myObj`. We use the `x` attribute on the object to print its value:

Example

Create an object called "`myObj`" and print the value of `x`:


```
public class Main {
    int x = 5;

    public static void main(String[] args) {
        Main myObj = new Main();
        System.out.println(myObj.x);
    }
}
```

Modify Attributes

You can also modify attribute values:

Example

Set the value of **x** to 40:

```
public class Main {
    int x;

    public static void main(String[] args) {
        Main myObj = new Main();
        myObj.x = 40;
        System.out.println(myObj.x);
    }
}
```

Or override existing values:

Example

Change the value of **x** to 25:

```
public class Main {
    int x = 10;

    public static void main(String[] args) {
        Main myObj = new Main();
        myObj.x = 25; // x is now 25
        System.out.println(myObj.x);
    }
}
```



If you don't want the ability to override existing values, declare the attribute as **final**:

Example

```
public class Main {
    final int x = 10;

    public static void main(String[] args) {
        Main myObj = new Main();
        myObj.x = 25; // will generate an error: cannot assign a value to a final variable
        System.out.println(myObj.x);
    }
}
```

Multiple Objects

If you create multiple objects of one class, you can change the attribute values in one object, without affecting the attribute values in the other:

Example

Change the value of **x** to 25 in **myObj2**, and leave **x** in **myObj1** unchanged:

```
public class Main {
```

```
int x = 5;
```

```
public static void main(String[] args) {  
    Main myObj1 = new Main(); // Object 1  
    Main myObj2 = new Main(); // Object 2  
    myObj2.x = 25;  
    System.out.println(myObj1.x); // Outputs 5  
    System.out.println(myObj2.x); // Outputs 25  
}
```

Multiple Attributes

You can specify as many attributes as you want:

Example

```
public class Main {  
    String fname = "John";  
    String lname = "Doe";  
    int age = 24;  
  
    public static void main(String[] args) {  
        Main myObj = new Main();  
        System.out.println("Name: " + myObj.fname + " " + myObj.lname);  
        System.out.println("Age: " + myObj.age);  
    }  
}
```

Static vs. Non-Static

You will often see Java programs that have either **static** or **public** attributes and methods.

In the example above, we created a **static** method, which means that it can be accessed without creating an object of the class, unlike **public**, which can only be accessed by objects:

Example

An example to demonstrate the differences between **static** and **public methods**:

```
public class Main {  
    // Static method  
    static void myStaticMethod() {  
        System.out.println("Static methods can be called without creating objects");  
    }  
  
    // Public method  
    public void myPublicMethod() {  
        System.out.println("Public methods must be called by creating objects");  
    }  
  
    // Main method  
    public static void main(String[] args) {  
        myStaticMethod(); // Call the static method  
        // myPublicMethod(); This would compile an error  
  
        Main myObj = new Main(); // Create an object of Main  
        myObj.myPublicMethod(); // Call the public method on the object  
    }  
}
```

```
}
```

Using Multiple Classes

It is a good practice to create an object of a class and access it in another class.

Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory:

Main.java

Second.java

Main.java

```
public class Main {  
    public void fullThrottle() {  
        System.out.println("The car is going as fast as it can!");  
    }  
}
```

```
    public void speed(int maxSpeed) {  
        System.out.println("Max speed is: " + maxSpeed);  
    }  
}
```

Second.java

```
class Second {  
    public static void main(String[] args) {  
        Main myCar = new Main(); // Create a myCar object  
        myCar.fullThrottle(); // Call the fullThrottle() method  
        myCar.speed(200); // Call the speed() method  
    }  
}
```

When both files have been compiled:

C:\Users\Your Name>javac Main.java

C:\Users\Your Name>javac Second.java

Run the Second.java file:

C:\Users\Your Name>java Second

And the output will be:

The car is going as fast as it can!

Max speed is: 200

Java Constructors

A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes:

Example

Create a constructor:

// Create a Main class

```
public class Main {  
    int x; // Create a class attribute
```

// Create a **class constructor** for the Main class

```
    public Main() {  
        x = 5; // Set the initial value for the class attribute x  
    }  
}
```

```
    public static void main(String[] args) {  
        Main myObj = new Main(); // Create an object of class Main (This will call the constructor)
```

```
System.out.println(myObj.x); // Print the value of x
}
}
```

// Outputs 5

Constructor Parameters

Constructors can also take parameters, which is used to initialize attributes.

The following example adds an **int y** parameter to the constructor. Inside the constructor we set x to y (x=y). When we call the constructor, we pass a parameter to the constructor (5), which will set the value of x to 5:

Example

```
public class Main {
    int x;

    public Main(int y) {
        x = y;
    }

    public static void main(String[] args) {
        Main myObj = new Main(5);
        System.out.println(myObj.x);
    }
}
```

// Outputs 5

You can have as many parameters as you want:

Example

```
public class Main {
    int modelYear;
    String modelName;

    public Main(int year, String name) {
        modelYear = year;
        modelName = name;
    }

    public static void main(String[] args) {
        Main myCar = new Main(1969, "Mustang");
        System.out.println(myCar.modelYear + " " + myCar.modelName);
    }
}
```

// Outputs 1969 Mustang

Java Packages & API

A package in Java is used to group related classes. Think of it as a **folder in a file directory**. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

Built-in Packages (packages from the Java API)

User-defined Packages (create your own packages)

Built-in Packages

The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment. The library contains components for managing input, database programming, and much much more.

The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

To use a class or a package from the library, you need to use the **import** keyword:

Syntax

```
import package.name.Class; // Import a single class
```

```
import package.name.*; // Import the whole package
```

Import a Class

If you find a class you want to use, for example, the **Scanner** class, **which is used to get user input**, write the following code:

Example

```
import java.util.Scanner;
```

In the example above, **java.util** is a package, while **Scanner** is a class of the **java.util** package.

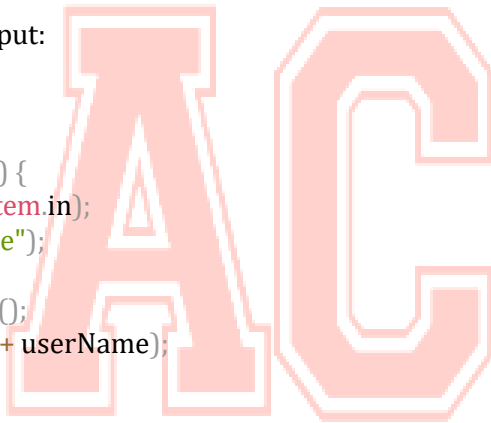
To use the **Scanner** class, create an object of the class and use any of the available methods found in the **Scanner** class documentation. In our example, we will use the **nextLine()** method, which is used to read a complete line:

Example

Using the **Scanner** class to get user input:

```
import java.util.Scanner;
```

```
class MyClass {  
    public static void main(String[] args) {  
        Scanner myObj = new Scanner(System.in);  
        System.out.println("Enter username");  
  
        String userName = myObj.nextLine();  
        System.out.println("Username is: " + userName);  
    }  
}
```



Import a Package

There are many packages to choose from. In the previous example, we used the **Scanner** class from the **java.util** package. This package also contains date and time facilities, random-number generator and other utility classes.

To import a whole package, end the sentence with an asterisk sign (*). The following example will import ALL the classes in the **java.util** package:

Example

```
import java.util.*;
```

User-defined Packages

To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer:

Example

```
└─ root  
  └─ mypack
```

└─ MyPackageClass.java

To create a package, use the **package** keyword:

MyPackageClass.java

```
package mypack;
class MyPackageClass {
    public static void main(String[] args) {
        System.out.println("This is my package!");
    }
}
```

Save the file as **MyPackageClass.java**, and compile it:

C:\Users\Your Name>javac MyPackageClass.java

- Then compile the package:
- C:\Users\Your Name>javac -d . MyPackageClass.java
- This forces the compiler to create the "mypack" package.
- The **-d** keyword specifies the destination for where to save the class file. You can use any directory name, like c:/user (windows), or, if you want to keep the package within the same directory, you can use the dot sign ".", like in the example above.

Note: The package name should be written in lower case to avoid conflict with class names.

When we compiled the package in the example above, a new folder was created, called "mypack".

To run the **MyPackageClass.java** file, write the following:

C:\Users\Your Name>java mypack.MyPackageClass

The output will be:

This is my package!

Java Methods

A **method** is a block of code which only runs when it is called.

You can pass data, known as parameters, into a method.

Methods are used to perform certain actions, and they are also known as **functions**.

Why use methods? To reuse code: define the code once, and use it many times.

Create a Method

A method must be declared within a class. It is defined with the name of the method, followed by parentheses ().

Java provides some pre-defined methods, such as **System.out.println()**, but you can also create your own methods to perform certain actions:

Example

Create a method inside Main:

```
public class Main {
    static void myMethod() {
        // code to be executed
    }
}
```

Example Explained

myMethod() is the name of the method

static means that the method belongs to the Main class and not an object of the Main class. You will learn more about objects and how to access methods through objects later in this tutorial.

void means that this method does not have a return value. You will learn more about return values later in this chapter

Call a Method

To call a method in Java, write the method's name followed by two parentheses () and a semicolon;

In the following example, `myMethod()` is used to print a text (the action), when it is called:

Example

Inside `main`, call the `myMethod()` method:

```
public class Main {
    static void myMethod() {
        System.out.println("I just got executed!");
    }

    public static void main(String[] args) {
        myMethod();
    }
}

// Outputs "I just got executed!"
```

A method can also be called multiple times:

Example

```
public class Main {
    static void myMethod() {
        System.out.println("I just got executed!");
    }

    public static void main(String[] args) {
        myMethod();
        myMethod();
        myMethod();
    }
}

// I just got executed!
// I just got executed!
// I just got executed!
```



Java Method Parameters

Parameters and Arguments

Information can be passed to methods as parameter. Parameters act as variables inside the method.

Parameters are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a method that takes a `String` called `fname` as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

Example

```
public class Main {
    static void myMethod(String fname) {
        System.out.println(fname + " Refsnes");
    }

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

```

    myMethod("Liam");
    myMethod("Jenny");
    myMethod("Anja");
}
}
// Liam Refsnes
// Jenny Refsnes
// Anja Refsnes

```

Multiple Parameters

You can have as many parameters as you like:

Example

```

public class Main {
    static void myMethod(String fname, int age) {
        System.out.println(fname + " is " + age);
    }

    public static void main(String[] args) {
        myMethod("Liam", 5);
        myMethod("Jenny", 8);
        myMethod("Anja", 31);
    }
}

// Liam is 5
// Jenny is 8
// Anja is 31

```

Return Values

The **void** keyword, used in the examples above, indicates that the method should not return a value. If you want the method to return a value, you can use a primitive data type (such as **int**, **char**, etc.) instead of **void**, and use the **return** keyword inside the method:

Example

```

public class Main {
    static int myMethod(int x) {
        return 5 + x;
    }

    public static void main(String[] args) {
        System.out.println(myMethod(3));
    }
}

// Outputs 8 (5 + 3)

```

This example returns the sum of a method's **two parameters**:

Example

```

public class Main {
    static int myMethod(int x, int y) {
        return x + y;
    }
}

```



```
public static void main(String[] args) {  
    System.out.println(myMethod(5, 3));  
}  
}  
// Outputs 8 (5 + 3)
```

You can also store the result in a variable (recommended, as it is easier to read and maintain):

Example

```
public class Main {  
    static int myMethod(int x, int y) {  
        return x + y;  
    }  
  
    public static void main(String[] args) {  
        int z = myMethod(5, 3);  
        System.out.println(z);  
    }  
}  
// Outputs 8 (5 + 3)
```

A Method with If...Else

It is common to use **if...else** statements inside methods:

Example

```
public class Main {  
  
    // Create a checkAge() method with an integer variable called age  
    static void checkAge(int age) {  
  
        // If age is less than 18, print "access denied"  
        if (age < 18) {  
            System.out.println("Access denied - You are not old enough!");  
        }  
  
        // If age is greater than, or equal to, 18, print "access granted"  
        else {  
            System.out.println("Access granted - You are old enough!");  
        }  
    }  
  
    public static void main(String[] args) {  
        checkAge(20); // Call the checkAge method and pass along an age of 20  
    }  
}  
  
// Outputs "Access granted - You are old enough!"
```

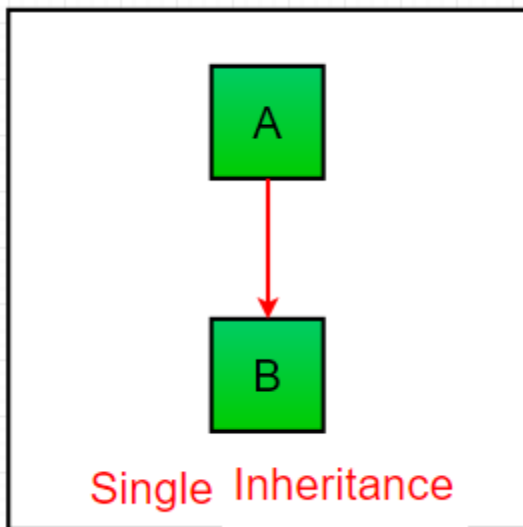
Inheritance and Interfaces

- 4.1. Classes, Super classes, and Subclasses
- 4.2. Polymorphism
- 4.3. Dynamic Binding
- 4.4. Final Classes and Methods
- 4.5. Abstract Classes
- 4.6. Access Specifiers
- 4.7. Interfaces

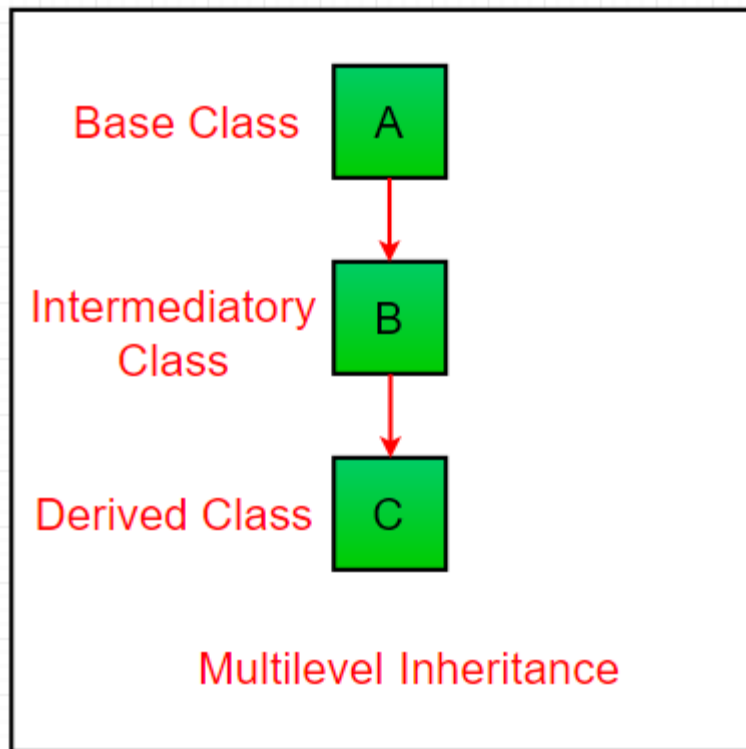
Interface: Interfaces are the blueprints of the classes. They specify what a class must do and not how. Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (i.e.) they only contain method signature and not the body of the method. Interfaces are used to implement a complete abstraction.

Inheritance: It is a mechanism in java by which one class is allowed to inherit the features of the another class. There are multiple inheritances possible in java. They are:

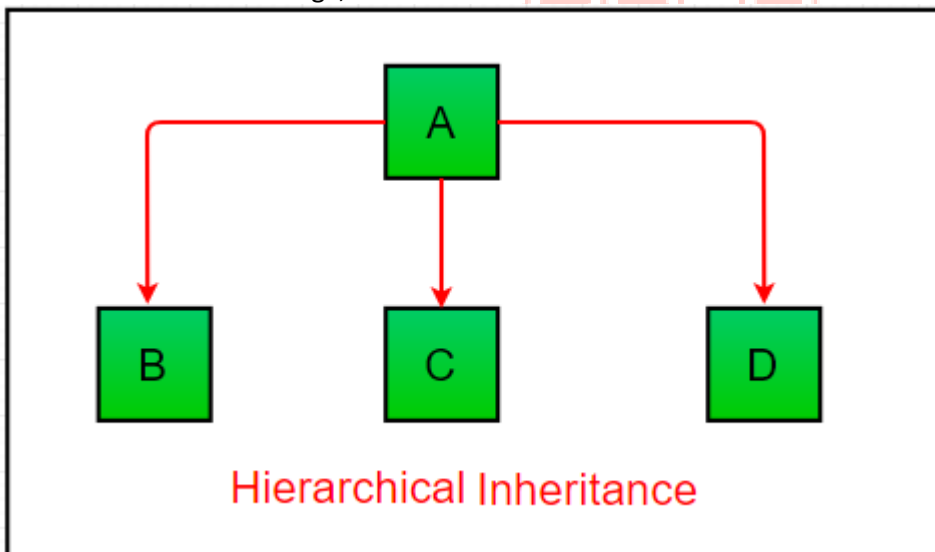
Single Inheritance: In single inheritance, subclasses inherit the features of one superclass. In the image below, class A serves as a base class for the derived class B.



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 the below image, 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.



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



The following table describes the difference between the inheritance and interface:

Category	Inheritance	Interface
Description	Inheritance is the mechanism in java by which one class is allowed to inherit the features of another class.	Interface is the blueprint of the class. It specifies what a class must do and not how. Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).
Use	It is used to get the features of another class.	It is used to provide total abstraction.
Functionality Provided	It does not provide the functionality of loose coupling	It provides the functionality of loose coupling.
Multiple Inheritance	We cannot do multiple inheritance (causes compile time error).	We can do multiple inheritance using interfaces.

Inheritance in Java

Inheritance

Types of Inheritance

Why multiple inheritance is not possible in Java in case of class?

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Why use inheritance in java

For Method Overriding (so runtime polymorphism can be achieved).

For Code Reusability.

Terms used in Inheritance

Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

Sub Class/Child Class: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

Super Class/Parent Class: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

Reusability: As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

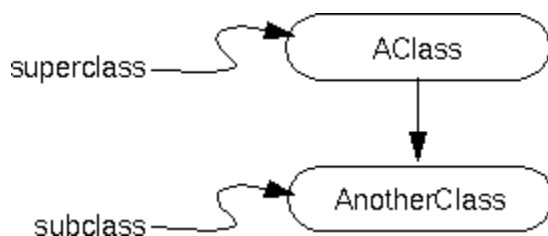
```
class Subclass-name extends Superclass-name
{
    //methods and fields
}
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

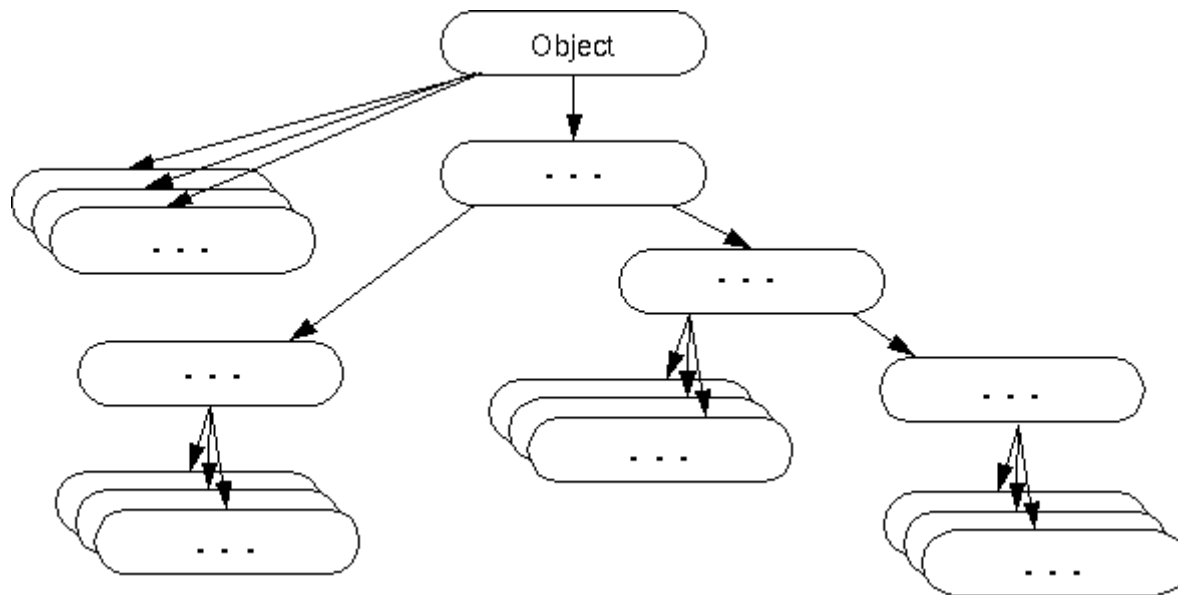
In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Subclasses, Superclasses, and Inheritance

In Java, as in other object-oriented programming languages, classes can be derived from other classes. The derived class (the class that is derived from another class) is called a subclass. The class from which its derived is called the superclass.



In fact, in Java, all classes must be derived from some class. Which leads to the question "Where does it all begin?" The top-most class, the class from which all other classes are derived, is the Object class defined in java.lang. Object is the root of a hierarchy of classes.



The subclass inherits state and behavior in the form of variables and methods from its superclass. The subclass can just use the items inherited from its superclass as is, or the subclass can modify or override it. So, as you drop down in the hierarchy, the classes become more and more specialized:

Definition: A subclass is a class that derives from another class. A subclass inherits state and behavior from all of its ancestors. The term superclass refers to a class's direct ancestor as well as all of its ascendant classes.

Creating Subclasses

To create a subclass of another class use the `extends` clause in your class declaration. ([The Class Declaration](#) explains all of the components of a class declaration in detail.) As a subclass, your class inherits member variables and methods from its superclass. Your class can choose to hide variables or override methods inherited from its superclass.

Writing Final Classes and Methods

Sometimes, for security or design reasons, you want to prevent your class from being subclassed. Or, you may just wish to prevent certain methods within your class from being overridden. In Java, you can achieve either of these goals by marking the class or the method as *final*.

Writing Abstract Classes and Methods

On the other hand, some classes are written for the sole purpose of being subclassed (and are not intended to ever be instantiated). These classes are called *abstract classes* and often contain *abstract methods*.

The Object Class

All objects in the Java environment inherit either directly or indirectly from the `Object` class. This section talks about the interesting methods in `Object`--methods that you may wish to invoke or override.

In the example below, the `Car` class (subclass) inherits the attributes and methods from the `Vehicle` class (superclass):

Example

```
class Vehicle {
    protected String brand = "Ford";    // Vehicle attribute
    public void honk() {                  // Vehicle method
```

```

    System.out.println("Tuut, tuut!");
}
}

```

```

class Car extends Vehicle {
    private String modelName = "Mustang"; // Car attribute
    public static void main(String[] args) {

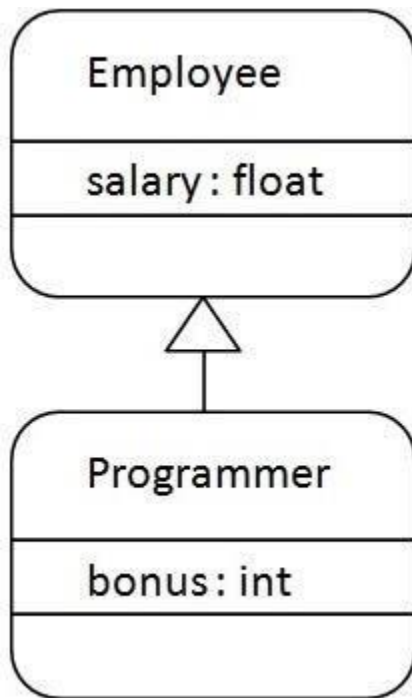
        // Create a myCar object
        Car myCar = new Car();

        // Call the honk() method (from the Vehicle class) on the myCar object
        myCar.honk();

        // Display the value of the brand attribute (from the Vehicle class) and the value of the modelName from the Car
        class
        System.out.println(myCar.brand + " " + myCar.modelName);
    }
}

```

Java Inheritance Example



ACC

As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

```

class Employee{
    float salary=40000;
}
class Programmer extends Employee{
    int bonus=10000;
    public static void main(String args[]){

```

```

Programmer p=new Programmer();
System.out.println("Programmer salary is:"+p.salary);
System.out.println("Bonus of Programmer is:"+p.bonus);
}
}

```

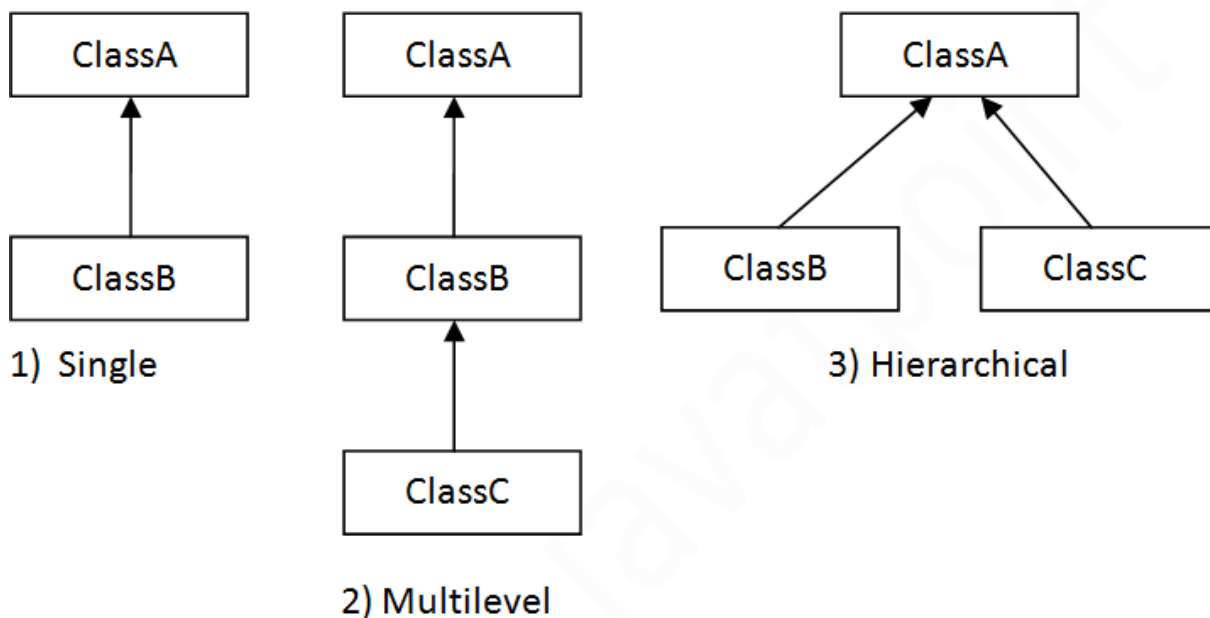
Programmer salary is:40000.0
 Bonus of programmer is:10000

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

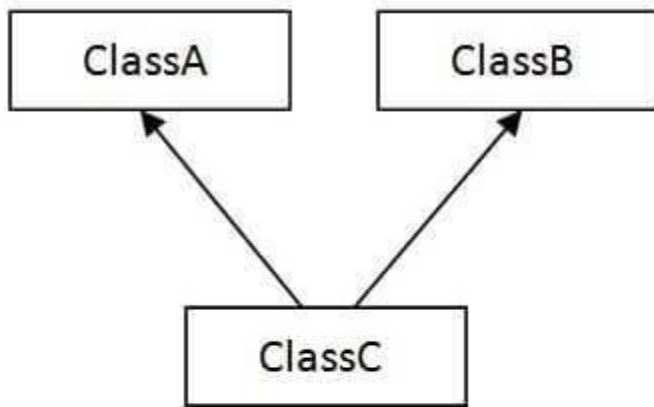
Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

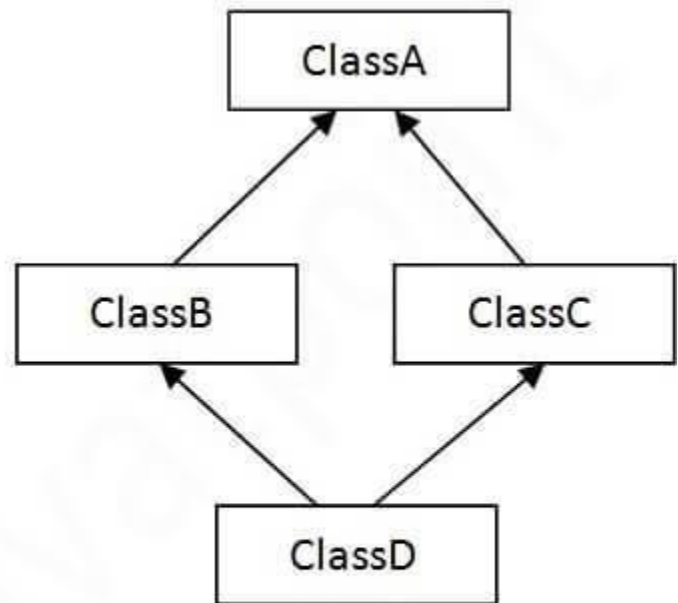
In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



When one class inherits multiple classes, it is known as multiple inheritance. For Example:



4) Multiple



5) Hybrid

Single Inheritance Example

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

File: TestInheritance.java

```

class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
  
```

Output:
barking...
eating...

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

File: TestInheritance2.java

```

class Animal{
  
```

```

void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
Output:
weeping...
barking...
eating...

```

Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

File: TestInheritance3.java

```

class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
Output:
meowing...
eating...

```

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

public static void main(String args[]){
    C obj=new C();
    obj.msg();//Now which msg() method would be invoked?
}
}
Compile Time Error
Output : Compile time error
```

Java Polymorphism

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified **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 pig oinks, and the cat meows, etc.):

Example

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

```
class Pig extends Animal {
    public void animalSound() {
        System.out.println("The pig says: wee wee");
    }
}
```

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

Remember from the Inheritance chapter that we use the extends keyword to inherit from a class.

Now we can create Pig and Dog objects and call the animalSound() method on both of them:

Example

```
class Animal {
    public void animalSound() {
```

```
    System.out.println("The animal makes a sound");  
}  
}
```

```
class Pig extends Animal {  
    public void animalSound() {  
        System.out.println("The pig says: wee wee");  
    }  
}
```

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

```
class Main {  
    public static void main(String[] args) {  
        Animal myAnimal = new Animal(); // Create a Animal object  
        Animal myPig = new Pig(); // Create a Pig object  
        Animal myDog = new Dog(); // Create a Dog object  
        myAnimal.animalSound();  
        myPig.animalSound();  
        myDog.animalSound();  
    }  
}
```

Why And When To Use "Inheritance" and "Polymorphism"?

- It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.

Static Binding and Dynamic Binding

Connecting a method call to the method body is known as binding.

There are two types of binding

Static Binding (also known as Early Binding).

Dynamic Binding (also known as Late Binding).

Static vs Dynamic Binding

Static Binding

When type of the object is determined at compiled time, it is known as static binding.

When type of the object is determined at run-time, it is known as dynamic binding.

Dynamic Binding

Understanding Type

Let's understand the type of instance.

1) variables have a type

Each variable has a type, it may be primitive and non-primitive.

```
int data=30;
```

Here data variable is a type of int.

2) References have a type

```
class Dog{  
    public static void main(String args[]){  
        Dog d1;//Here d1 is a type of Dog  
    }  
}
```

3) Objects have a type

An object is an instance of particular java class, but it is also an instance of its superclass.

```
class Animal{
```

```
    class Dog extends Animal{  
        public static void main(String args[]){  
            Dog d1=new Dog();  
        }  
    }
```

Here d1 is an instance of Dog class, but it is also an instance of Animal.

static binding

When type of the object is determined at compiled time (by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

Example of static binding

```
class Dog{
    private void eat(){System.out.println("dog is eating...");}

    public static void main(String args[]){
        Dog d1=new Dog();
        d1.eat();
    }
}
```

Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding

```
class Animal{
    void eat(){System.out.println("animal is eating...");}
}
```

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

```
    public static void main(String args[]){
        Animal a=new Dog();
        a.eat();
    }
}
```

Output:dog is eating...

In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal. So compiler doesn't know its type, only its base type.

In dynamic binding, the method call is bonded to the method body at runtime. This is also known as late binding. This is done using instance methods.

Example

```
class Super {
    public void sample() {
        System.out.println("This is the method of super class");
    }
}
```

```
Public class extends Super {
    Public static void sample() {
        System.out.println("This is the method of sub class");
    }
}
```

```
    Public static void main(String args[]) {
        new Sub().sample()
    }
}
```

Output

This is the method of sub class

Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be: variable
method
class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike9{
    final int speedlimit=90;//final variable
    void run(){
        speedlimit=400;
    }
    public static void main(String args[]){
        Bike9 obj=new Bike9();
        obj.run();
    }
} //end of class
```

Output:Compile Time Error

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

```
class Bike{
    final void run(){System.out.println("running");}
}

class Honda extends Bike{
    void run(){System.out.println("running safely with 100kmph");}

    public static void main(String args[]){
        Honda honda= new Honda();
        honda.run();
    }
}
```

Output:Compile Time Error

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
final class Bike{}
```

```

class Honda1 extends Bike{
    void run(){System.out.println("running safely with 100kmph");}

    public static void main(String args[]){
        Honda1 honda= new Honda1();
        honda.run();
    }
}

```

Output:Compile Time Error

Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

```

class Bike{
    final void run(){System.out.println("running...");}
}
class Honda2 extends Bike{
    public static void main(String args[]){
        new Honda2().run();
    }
}

```

Output:running...

Q) What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

Example of blank final variable

```

class Student{
    int id;
    String name;
    final String PAN_CARD_NUMBER;
    ...
}

```

Que) Can we initialize blank final variable?

Yes, but only in constructor. For example:

```

class Bike10{
    final int speedlimit;//blank final variable

    Bike10(){
        speedlimit=70;
        System.out.println(speedlimit);
    }

    public static void main(String args[]){
        new Bike10();
    }
}

```



```
}
```

Output: 70

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example of static blank final variable

```
class A{
    static final int data;//static blank final variable
    static{ data=50;}
    public static void main(String args[]){
        System.out.println(A.data);
    }
}
```

Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

```
class Bike11{
    int cube(final int n){
        n=n+2;//can't be changed as n is final
        n*n*n;
    }
    public static void main(String args[]){
        Bike11 b=new Bike11();
        b.cube(5);
    }
}
```



Output: Compile Time Error

Abstract class in Java

A class which is declared with the abstract keyword is known as an abstract class in **Java**. It can have abstract and non-abstract methods (method with the body).

Before learning the Java abstract class, let's understand the abstraction in Java first.

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the **object** does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

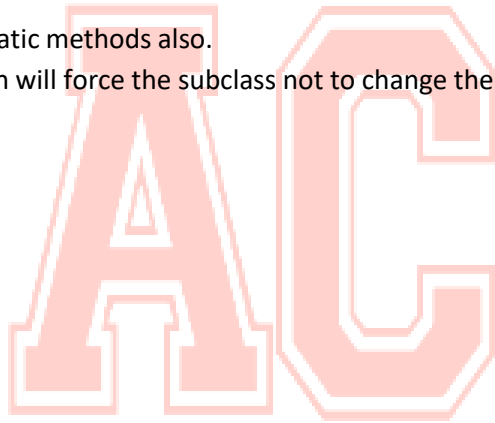
1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method



Rules for Java Abstract class



1

An abstract class must be declared with an abstract keyword.

2

It can have abstract and non-abstract methods.

3

It cannot be instantiated.

4

It can have final methods

5

It can have constructors and static methods also.

Example of abstract class

```
1. abstract class A{
```

Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of abstract method

```
1. abstract void printStatus();//no method body and abstract
```

Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
1. abstract class Bike{
2.     abstract void run();
```

```

3.     }
4.     class Honda4 extends Bike{
5.     void run(){System.out.println("running safely");}
6.     public static void main(String args[]){
7.         Bike obj = new Honda4();
8.         obj.run();
9.     }
10.    }

```

Access Specifiers In Java

Definition :

- Java Access Specifiers (also known as Visibility Specifiers) regulate access to classes, fields and methods in Java. These Specifiers determine whether a field or method in a class, can be used or invoked by another method in another class or sub-class. Access Specifiers can be used to restrict access. Access Specifiers are an integral part of object-oriented programming.

Types Of Access Specifiers :

In java we have four Access Specifiers and they are listed below.

1. public
2. private
3. protected
4. default(no specifier)

We look at these Access Specifiers in more detail.

Access Modifiers	Default	private	protected	public
Accessible inside the class	yes	yes	yes	yes
Accessible within the subclass inside the same package	yes	no	yes	yes
Accessible outside the package	no	no	no	yes
Accessible within the subclass outside the package	no	no	yes	yes

public specifiers :

Public Specifiers achieves the highest level of accessibility. Classes, methods, and fields declared as public can be accessed from any class in the Java program, whether these classes are in the same package or in another package.

Example :

```
public class Demo { // public class
public x, y, size; // public instance variables
}
```

private specifiers :

Private Specifiers achieves the lowest level of accessibility. private methods and fields can only be accessed within the same class to which the methods and fields belong. private methods and fields are not visible within subclasses and are not inherited by subclasses. So, the private access specifier is opposite to the public access specifier. Using Private Specifier we can achieve encapsulation and hide data from the outside world.

Example :

```
public class Demo { // public class
private double x, y; // private (encapsulated) instance variables

public set(int x, int y) { // setting values of private fields
this.x = x;
this.y = y;
}

public get() { // setting values of private fields
return Point(x, y);
}
}
```

protected specifiers :

Methods and fields declared as protected can only be accessed by the subclasses in other package or any class within the package of the protected members' class. The protected access specifier cannot be applied to class and interfaces.

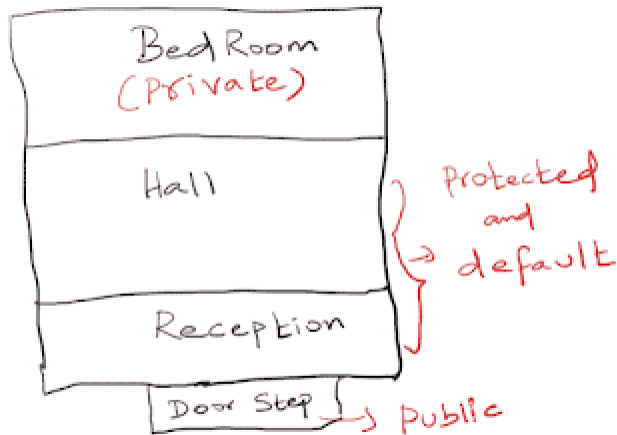
default(no specifier):

When you don't set access specifier for the element, it will follow the default accessibility level. There is no default specifier keyword. Classes, variables, and methods can be default accessed. Using default specifier we can access class, method, or field which belongs to same package, but not from outside this package.

Example :

```
class Demo
{
int i; (Default)
}
```

Real Time Example



Interfaces in Java

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

- Interfaces specify what a class must do and not how. It is the blueprint of the class.
- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
- A Java library example is, [Comparator Interface](#). If a class implements this interface, then it can be used to sort a collection.

Syntax :

```
interface <interface_name> {
```

```
// declare constant fields
```

```
// declare methods that abstract
```

```
// by default.
```

```
}
```

To declare an interface, use **interface** keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static and final by default. A

class that implements an interface must implement all the methods declared in the interface. To implement interface use **implements** keyword.

To implement an interface we use keyword: implements

```
// Java program to demonstrate working of
```

```
// interface.
```

```
import java.io.*;
```

```
// A simple interface
```

```
interface In1
```

```
{
```

```
    // public, static and final
```

```
    final int a = 10;
```

```
    // public and abstract
```

```
    void display();
```

```
}
```



```
// A class that implements the interface.
```

```
class TestClass implements In1
```

```
{
```

```
    // Implementing the capabilities of
```

```
    // interface.
```

```
    public void display()
```

```
    {
```

```
        System.out.println("Geek");
```

```

}

// Driver Code

public static void main (String[] args)

{

    TestClass t = new TestClass();

    t.display();

    System.out.println(a);

}

}

```

Output:

Geek

10

A real-world example:

Let's consider the example of vehicles like bicycle, car, bike....., they have common functionalities. So we make an interface and put all these common functionalities. And lets Bicycle, Bike, caretc implement all these functionalities in their own class in their own way.

```

import java.io.*;

interface Vehicle {

    // all are the abstract methods.
    void changeGear(int a);
    void speedUp(int a);
    void applyBrakes(int a);
}

class Bicycle implements Vehicle{

    int speed;
    int gear;

    // to change gear
    @Override
    public void changeGear(int newGear){

```



```

        gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){

        speed = speed + increment;
    }

    // to decrease speed
    @Override
    public void applyBrakes(int decrement){

        speed = speed - decrement;
    }

    public void printStates() {
        System.out.println("speed: " + speed
            + " gear: " + gear);
    }
}

class Bike implements Vehicle {

    int speed;
    int gear;

    // to change gear
    @Override
    public void changeGear(int newGear){

        gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){

        speed = speed + increment;
    }

    // to decrease speed
    @Override
    public void applyBrakes(int decrement){

        speed = speed - decrement;
    }
}

```



```

    public void printStates() {
        System.out.println("speed: " + speed
            + " gear: " + gear);
    }
}

class GFG {

    public static void main (String[] args) {

        // creating an inatance of Bicycle
        // doing some operations
        Bicycle bicycle = new Bicycle();
        bicycle.changeGear(2);
        bicycle.speedUp(3);
        bicycle.applyBrakes(1);

        System.out.println("Bicycle present state :");
        bicycle.printStates();

        // creating instance of the bike.
        Bike bike = new Bike();
        bike.changeGear(1);
        bike.speedUp(4);
        bike.applyBrakes(3);

        System.out.println("Bike present state :");
        bike.printStates();
    }
}

```

Output;

```

Bicycle present state :
speed: 2 gear: 2
Bike present state :
speed: 1 gear: 1

```



Unit –V Exception Handling

5.1. Dealing With Errors

5.2. Catching Exceptions

5.3. try, catch, throw, throws, and finally

What is an exception?

An Exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs program execution gets terminated. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled in Java. By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.

Why an exception occurs?

There can be several reasons that can cause a program to throw exception. For example: Opening a non-existing file in your program, Network connection problem, bad input data provided by user etc.

Exception Handling

If an exception occurs, which has not been handled by programmer then program execution gets terminated and a system generated error message is shown to the user. For example look at the system generated exception below:

An exception generated by the system is given below

```
Exception in thread "main" java.lang.ArithmeticException: / by zero at ExceptionDemo.main(ExceptionDemo.java:5)
ExceptionDemo : The class name
main : The method name
ExceptionDemo.java : The filename
java:5 : Line number
```

This message is not user friendly so a user will not be able to understand what went wrong. In order to let them know the reason in simple language, we handle exceptions. We handle such conditions and then prints a user friendly warning message to user, which lets them correct the error as most of the time exception occurs due to bad data provided by user.

Advantage of exception handling

Exception handling ensures that the flow of the program doesn't break when an exception occurs. For example, if a program has bunch of statements and an exception occurs mid way after executing certain statements then the statements after the exception will not execute and the program will terminate abruptly.

By handling we make sure that all the statements execute and the flow of program doesn't break.

Difference between error and exception

Errors indicate that something severe enough has gone wrong, the application should crash rather than try to handle the error.

Exceptions are events that occurs in the code. A programmer can handle such conditions and take necessary corrective actions. Few examples:

NullPointerException – When you try to use a reference that points to null.

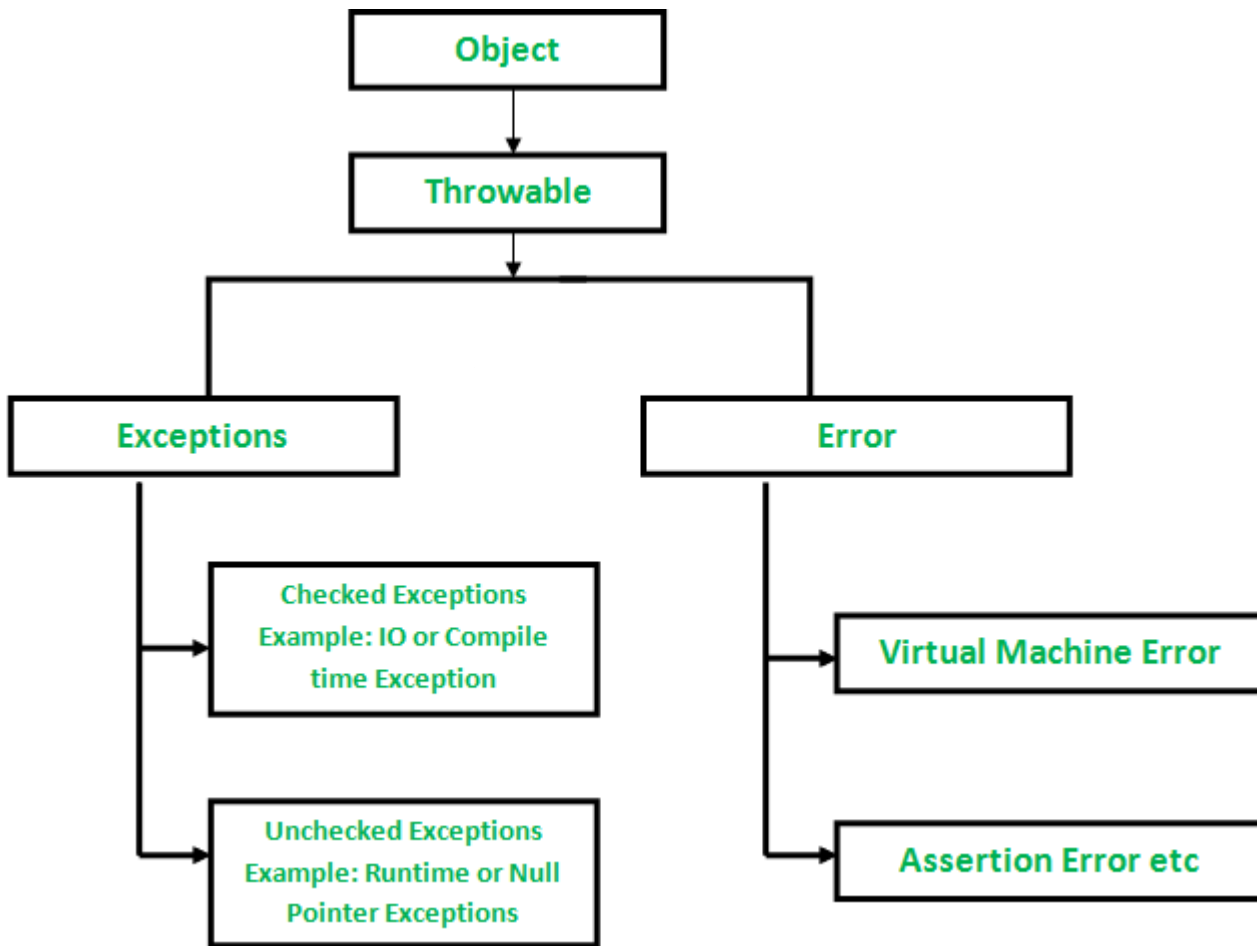
ArithmeticException – When bad data is provided by user, for example, when you try to divide a number by zero this

exception occurs because dividing a number by zero is undefined.

ArrayIndexOutOfBoundsException – When you try to access the elements of an array out of its bounds, for example array size is 5 (which means it has five elements) and you are trying to access the 10th element.

Exception Hierarchy

All exception and errors types are sub classes of class **Throwable**, which is base class of hierarchy. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch, **Error** are used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.



Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Java Exception Handling Example

Let's see an example of Java Exception Handling where we using a try-catch statement to handle the exception.

```
1. public class JavaExceptionExample{
2.     public static void main(String args[]){
3.         try{
4.             //code that may raise exception
5.             int data=100/0;
6.         }catch(ArithmeticException e){System.out.println(e);}
7.         //rest code of the program
8.         System.out.println("rest of the code...");
9.     }
10. }
```

11. Output:

12. Exception in thread main java.lang.ArithmeticException:/ by zero

13. rest of the code...

14. In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where `ArithmeticException` occurs

If we divide any number by zero, there occurs an `ArithmeticException`.

1. `int a=50/0;//ArithmeticException`
-

2) A scenario where `NullPointerException` occurs

If we have a null value in any `variable`, performing any operation on the variable throws a `NullPointerException`.

1. `String s=null;`
 2. `System.out.println(s.length());//NullPointerException`
-

3) A scenario where `NumberFormatException` occurs

The wrong formatting of any value may occur `NumberFormatException`. Suppose I have a `string` variable that has characters, converting this variable into digit will occur `NumberFormatException`.

1. `String s="abc";`
 2. `int i=Integer.parseInt(s);//NumberFormatException`
-

4) A scenario where `ArrayIndexOutOfBoundsException` occurs

If you are inserting any value in the wrong index, it would result in `ArrayIndexOutOfBoundsException` as shown below:

1. `int a[]=new int[5];`
2. `a[10]=50; //ArrayIndexOutOfBoundsExceptionJava`

try-catch block

Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement of try block, the rest of the block code will not execute. So, it is recommended not to keeping the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

1. `try{`
2. `//code that may throw an exception`
3. `}catch(Exception_class_Name ref){}`**Syntax**

of try-finally block

1. `try{`
2. `//code that may throw an exception`
3. `}finally{}`**Java**

catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Problem without exception handling

Let's try to understand the problem if we don't use a try-catch block.

Example 1

- ```
1. public class TryCatchExample1 {
2.
3. public static void main(String[] args) {
4.
5. int data=50/0; //may throw exception
6.
7. System.out.println("rest of the code");
8.
9. }
10.
11. }
```

### Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

As displayed in the above example, the **rest of the code** is not executed (in such case, the **rest of the code** statement is not printed).



There can be 100 lines of code after exception. So all the code after exception will not be executed.

---

### Solution by exception handling

Let's see the solution of the above problem by a java try-catch block.

#### Example 2

```
1. public class TryCatchExample2 {
2.
3. public static void main(String[] args) {
4. try
5. {
6. int data=50/0; //may throw exception
7. }
8. //handling the exception
9. catch(ArithmeticException e)
10. {
11. System.out.println(e);
12. }
13. System.out.println("rest of the code");
14. }
15.
16. }
```

17. Output:

```
18. java.lang.ArithmeticException: / by zero
19. rest of the code
```

20. Now, as displayed in the above example, the **rest of the code** is executed, i.e., the **rest of the code** statement is printed.

#### Example 3

In this example, we also kept the code in a try block that will not throw an exception.

```
1. public class TryCatchExample3 {
2.
3. public static void main(String[] args) {
4. try
5. {
6. int data=50/0; //may throw exception
7. // if exception occurs, the remaining statement will not execute
8. System.out.println("rest of the code");
9. }
10. // handling the exception
```

```
11. catch(ArithmeticException e)
12. {
13. System.out.println(e);
14. }
15.
16. }
17.
18. }
```

#### Output:

```
java.lang.ArithmeticException: / by zero
```

Here, we can see that if an exception occurs in the try block, the rest of the block code will not execute.

#### Example 4

Here, we handle the exception using the parent class exception.

```
1. public class TryCatchExample4 {
2.
3. public static void main(String[] args) {
4. try
5. {
6. int data=50/0; //may throw exception
7. }
8. // handling the exception by using Exception class
9. catch(Exception e)
10. {
11. System.out.println(e);
12. }
13. System.out.println("rest of the code");
14. }
15.
16. }
```

#### Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

#### Example 5

Let's see an example to print a custom message on exception.

```
1. public class TryCatchExample5 {
```

```

2.
3. public static void main(String[] args) {
4. try
5. {
6. int data=50/0; //may throw exception
7. }
8. // handling the exception
9. catch(Exception e)
10. {
11. // displaying the custom message
12. System.out.println("Can't divided by zero");
13. }
14. }
15.
16. }

```

**Output:**

```
Can't divided by zero
```

#### Example 6

Let's see an example to resolve the exception in a catch block.

```

1. public class TryCatchExample6 {
2.
3. public static void main(String[] args) {
4. int i=50;
5. int j=0;
6. int data;
7. try
8. {
9. data=i/j; //may throw exception
10. }
11. // handling the exception
12. catch(Exception e)
13. {
14. // resolving the exception in catch block
15. System.out.println(i/(j+2));
16. }
17. }
18. }

```

**Output:**

### Example 7

In this example, along with try block, we also enclose exception code in a catch block.

```

1. public class TryCatchExample7 {
2.
3. public static void main(String[] args) {
4.
5. try
6. {
7. int data1=50/0; //may throw exception
8.
9. }
10. // handling the exception
11. catch(Exception e)
12. {
13. // generating the exception in catch block
14. int data2=50/0; //may throw exception
15.
16. }
17. System.out.println("rest of the code");
18. }
19. }

```

#### Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Here, we can see that the catch block didn't contain the exception code. So, enclose exception code within a try block and use catch block only to handle the exceptions.

### Example 8

In this example, we handle the generated exception (Arithmetic Exception) with a different type of exception class (ArrayIndexOutOfBoundsException).

```

1. public class TryCatchExample8 {
2.
3. public static void main(String[] args) {
4.
5. try
6. {
7. int data=50/0; //may throw exception

```

```

8. }
9. // try to handle the ArithmeticException using ArrayIndexOutOfBoundsException
10. catch(ArrayIndexOutOfBoundsException e)
11. {
12. System.out.println(e);
13. }
14. System.out.println("rest of the code");
15. }
16.
17. }

```

#### Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

#### Example 9

Let's see an example to handle another unchecked exception.

```

1. public class TryCatchExample9 {
2.
3. public static void main(String[] args) {
4. try
5. {
6. int arr[] = {1,3,5,7};
7. System.out.println(arr[10]); //may throw exception
8. }
9. // handling the array exception
10. catch(ArrayIndexOutOfBoundsException e)
11. {
12. System.out.println(e);
13. }
14. System.out.println("rest of the code");
15. }
16.
17. }

```

#### Output:

```
java.lang.ArrayIndexOutOfBoundsException: 10
rest of the code
```

#### Example 10

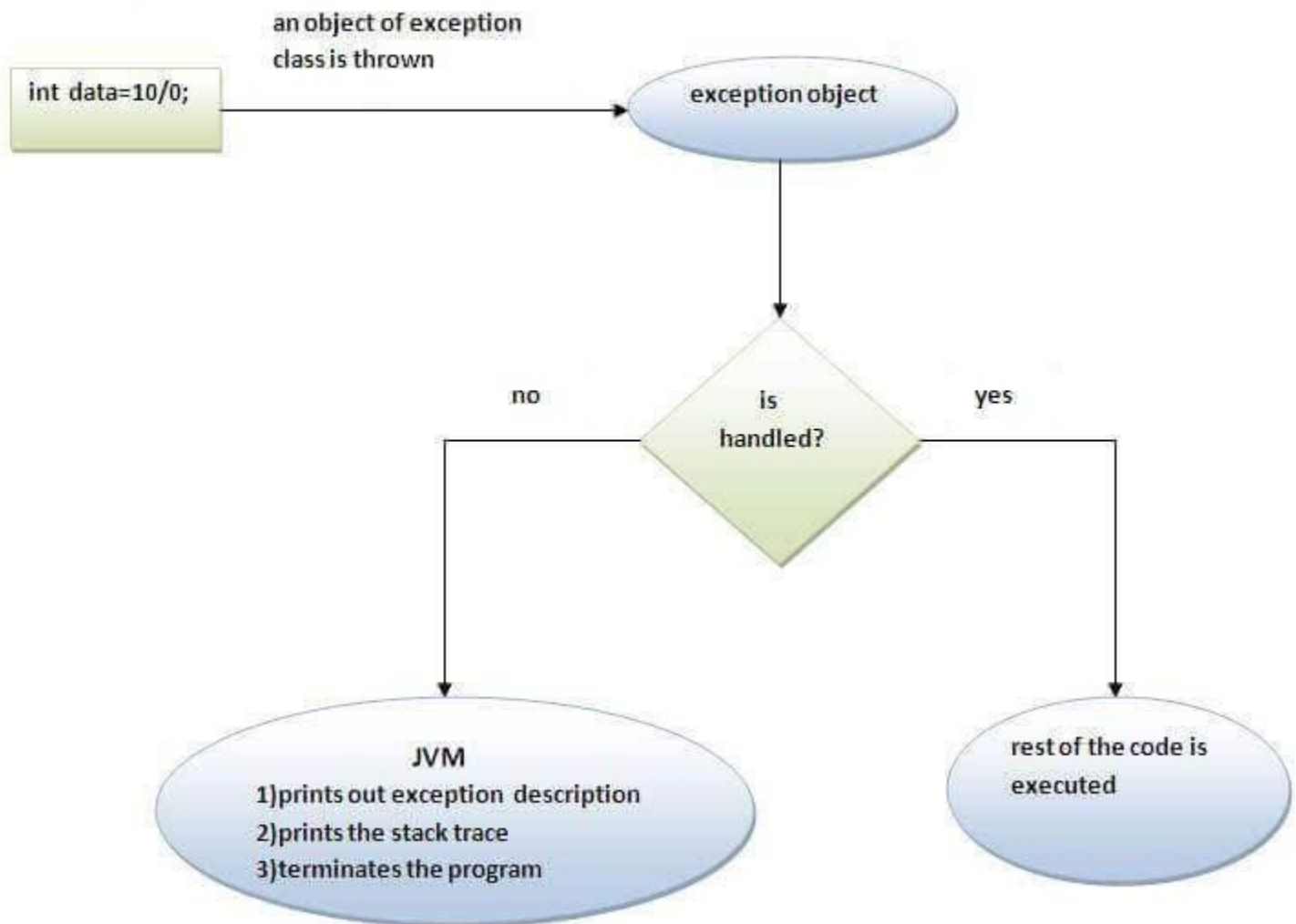
Let's see an example to handle checked exception.

```
1. import java.io.FileNotFoundException;
2. import java.io.PrintWriter;
3.
4. public class TryCatchExample10 {
5.
6. public static void main(String[] args) {
7.
8. PrintWriter pw;
9. try {
10. pw = new PrintWriter("jtp.txt"); //may throw exception
11. pw.println("saved");
12. }
13. }
14. // providing the checked exception handler
15. catch (FileNotFoundException e) {
16.
17. System.out.println(e);
18. }
19. System.out.println("File saved successfully");
20. }
21. }
```

Output:

```
File saved successfully
```

Internal working of java try-catch block



The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

### Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

## Points to remember

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for `ArithmeticException` must come before catch for `Exception`.

## Example 1

Let's see a simple example of java multi-catch block.

```
1. public class MultipleCatchBlock1 {
2.
3. public static void main(String[] args) {
4.
5. try{
6. int a[]=new int[5];
7. a[5]=30/0;
8. }
9. catch(ArithmeticException e)
10. {
11. System.out.println("Arithmetic Exception occurs");
12. }
13. catch(ArrayIndexOutOfBoundsException e)
14. {
15. System.out.println("ArrayIndexOutOfBoundsException occurs");
16. }
17. catch(Exception e)
18. {
19. System.out.println("Parent Exception occurs");
20. }
21. System.out.println("rest of the code");
22. }
23. }
```

## Output:

```
Arithmetic Exception occurs
rest of the code
```

## Example 2

```
1. public class MultipleCatchBlock2 {
2.
3. public static void main(String[] args) {
4.
```



```

5. try{
6. int a[]=new int[5];
7.
8. System.out.println(a[10]);
9. }
10. catch(ArithmeticException e)
11. {
12. System.out.println("Arithmetic Exception occurs");
13. }
14. catch(ArrayIndexOutOfBoundsException e)
15. {
16. System.out.println("ArrayIndexOutOfBoundsException occurs");
17. }
18. catch(Exception e)
19. {
20. System.out.println("Parent Exception occurs");
21. }
22. System.out.println("rest of the code");
23. }
24. }

```

Output:

```

ArrayIndexOutOfBoundsException occurs
rest of the code

```

### Example 3

In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is invoked.

```

1. public class MultipleCatchBlock3 {
2.
3. public static void main(String[] args) {
4.
5. try{
6. int a[]=new int[5];
7. a[5]=30/0;
8. System.out.println(a[10]);
9. }
10. catch(ArithmeticException e)
11. {
12. System.out.println("Arithmetic Exception occurs");
13. }
14. catch(ArrayIndexOutOfBoundsException e)

```

```

15. {
16. System.out.println("ArrayIndexOutOfBoundsException occurs");
17. }
18. catch(Exception e)
19. {
20. System.out.println("Parent Exception occurs");
21. }
22. System.out.println("rest of the code");
23. }
24. }

```

```

Arithmetic Exception occurs
rest of the code

```

#### Example 4

In this example, we generate `NullPointerException`, but didn't provide the corresponding exception type. In such case, the catch block containing the parent exception class **Exception** will invoked.

```

1. public class MultipleCatchBlock4 {
2.
3. public static void main(String[] args) {
4.
5. try{
6. String s=null;
7. System.out.println(s.length());
8. }
9. catch(ArithmeticException e)
10. {
11. System.out.println("Arithmetic Exception occurs");
12. }
13. catch(ArrayIndexOutOfBoundsException e)
14. {
15. System.out.println("ArrayIndexOutOfBoundsException occurs");
16. }
17. catch(Exception e)
18. {
19. System.out.println("Parent Exception occurs");
20. }
21. System.out.println("rest of the code");
22. }
23. }

```

#### Output:

```

Parent Exception occurs

```

rest of the code

### Example 5

Let's see an example, to handle the exception without maintaining the order of exceptions (i.e. from most specific to most general).

```
1. class MultipleCatchBlock5{
2. public static void main(String args[]){
3. try{
4. int a[]=new int[5];
5. a[5]=30/0;
6. }
7. catch(Exception e){System.out.println("common task completed");}
8. catch(ArithmeticException e){System.out.println("task1 is completed");}
9. catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
10. System.out.println("rest of the code...");
11. }
12. }
```

### Output:

Compile-time error

### Java Nested try block

The try block within a try block is known as nested try block in java.

### Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

### Syntax:

```
1.
2. try
3. {
4. statement 1;
5. statement 2;
6. try
7. {
8. statement 1;
9. statement 2;
10. }
11. catch(Exception e)
```

```
12. {
13. }
14. }
15. catch(Exception e)
16. {
17. }
18.
```

### Java nested try example

Let's see a simple example of java nested try block.

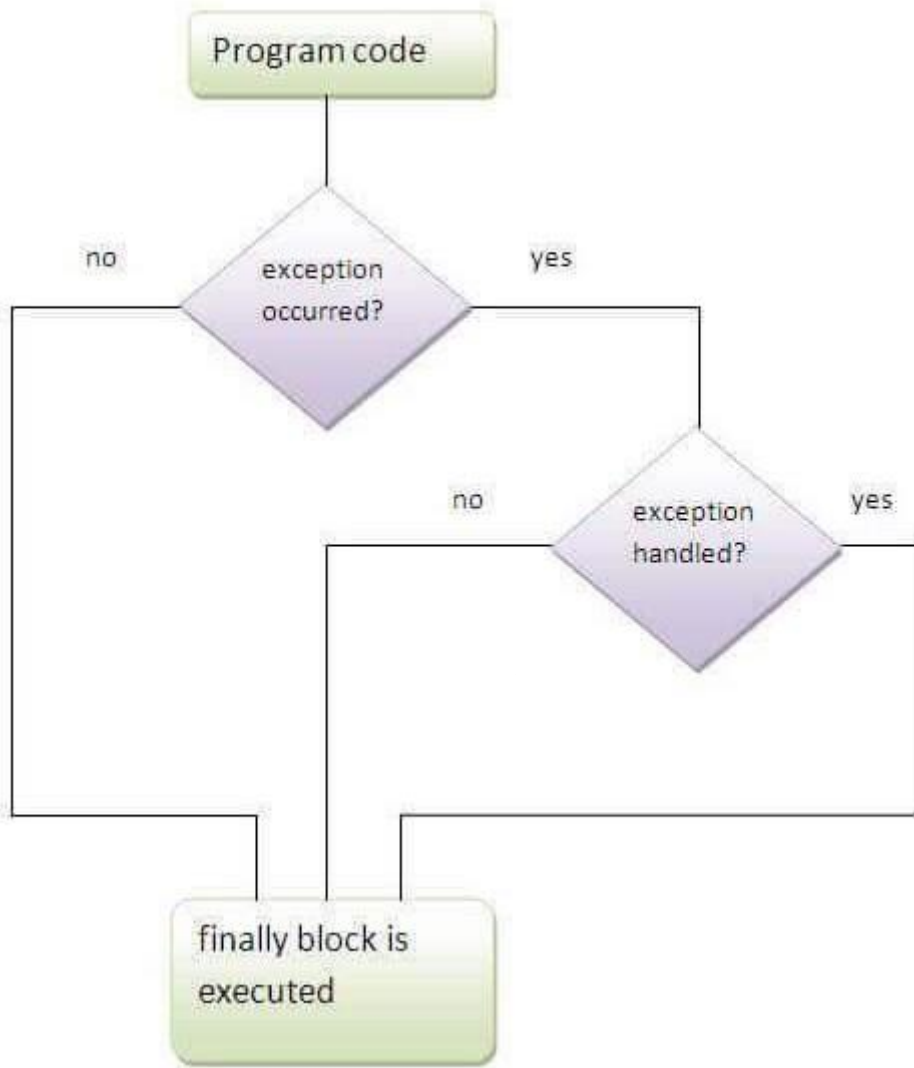
```
1. class Excep6{
2. public static void main(String args[]){
3. try{
4. try{
5. System.out.println("going to divide");
6. int b =39/0;
7. }catch(ArithmeticException e){System.out.println(e);}
8.
9. try{
10. int a[]=new int[5];
11. a[5]=4;
12. }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}
13.
14. System.out.println("other statement);
15. }catch(Exception e){System.out.println("handeled");}
16.
17. System.out.println("normal flow..");
18. }
19. }
```

### Java finally block

**Java finally block** is a block that is used *to execute important code* such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.



*Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).*

### Why use java finally

- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

### Usage of Java finally

Let's see the different cases where java finally block can be used.

#### Case 1

Let's see the java finally example where **exception doesn't occur**.

```

1. class TestFinallyBlock{
2. public static void main(String args[]){
3. try{
4. int data=25/5;
5. System.out.println(data);
6. }
7. catch(NullPointerException e){System.out.println(e);}
8. finally{System.out.println("finally block is always executed");}
9. System.out.println("rest of the code...");
10. }
11. }

```

```

12. Output:5
13. finally block is always executed
14. rest of the code...

```

### 15. Case 2

16. Let's see the java finally example where **exception occurs and not handled**.

```

1. class TestFinallyBlock1{
2. public static void main(String args[]){
3. try{
4. int data=25/0;
5. System.out.println(data);
6. }
7. catch(NullPointerException e){System.out.println(e);}
8. finally{System.out.println("finally block is always executed");}
9. System.out.println("rest of the code...");
10. }
11. }

```

```

Output:finally block is always executed
Exception in thread main java.lang.ArithmeticException:/ by zero

```

### Case 3

Let's see the java finally example where **exception occurs and handled**.

```

1. public class TestFinallyBlock2{
2. public static void main(String args[]){
3. try{
4. int data=25/0;
5. System.out.println(data);
6. }
7. catch(ArithmeticException e){System.out.println(e);}
8. finally{System.out.println("finally block is always executed");}
9. System.out.println("rest of the code...");
10. }

```

11. }

```
Output:Exception in thread main java.lang.ArithmeticException:/ by zero
finally block is always executed
rest of the code...
```

### Java throw keyword

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or unchecked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

1. **throw** exception;

Let's see the example of throw IOException.

1. **throw new** IOException("sorry device error);

### java throw keyword example

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
1. public class TestThrow1{
2. static void validate(int age){
3. if(age<18)
4. throw new ArithmeticException("not valid");
5. else
6. System.out.println("welcome to vote");
7. }
8. public static void main(String args[]){
9. validate(13);
10. System.out.println("rest of the code...");
11. }
12. }
```

# JAVA INPUT/OUTPUT

- **Java I/O** (Input and Output) is used to process the input and produce the output based on the input.
- Java uses the concept of stream to make I/O operation fast. The `java.io` package contains all the classes required for input and output operations.
- We can perform **file handling in java** by `javaIO` API.

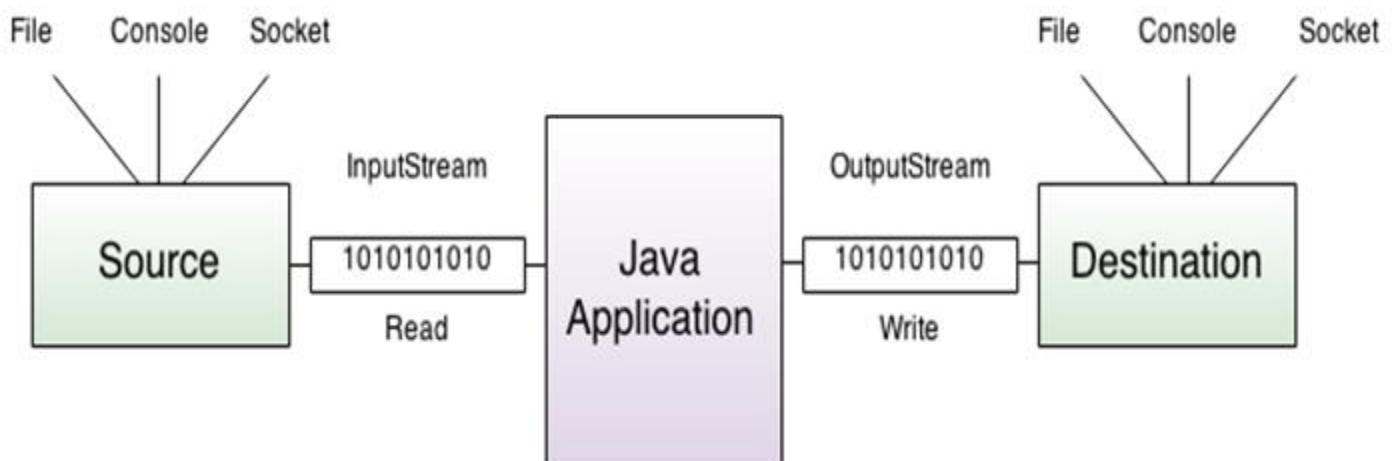
## STREAM

- A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it's like a stream of water that continues to flow.
- In Java, 3 streams are created for us automatically. All these streams are attached with console.
- **1) System.out:** standard output stream
- **2) System.in:** standard input stream
- **3) System.err:** standard error stream
- Let's see the code to print **output** and **error** message to the console.
  1. `System.out.println("simple message");`
  2. `System.err.println("error message");`



- Let's see the code to get **input** from console.
  - `int i=System.in.read();`//returns ASCII code of 1st character
  - `System.out.println((char)i);`//will print the character
- The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data types such as primitives, Object, localized characters, etc.
- Stream
- A stream can be defined as a sequence of data. there are two kinds of Streams
- InPutStream:** The InputStream is used to read data from a source.
- OutPutStream:** the OutputStream is used for writing data to a destination.

**LET'S UNDERSTAND WORKING OF JAVA OUTPUTSTREAM AND INPUTSTREAM BY THE FIGURE GIVEN BELOW.**

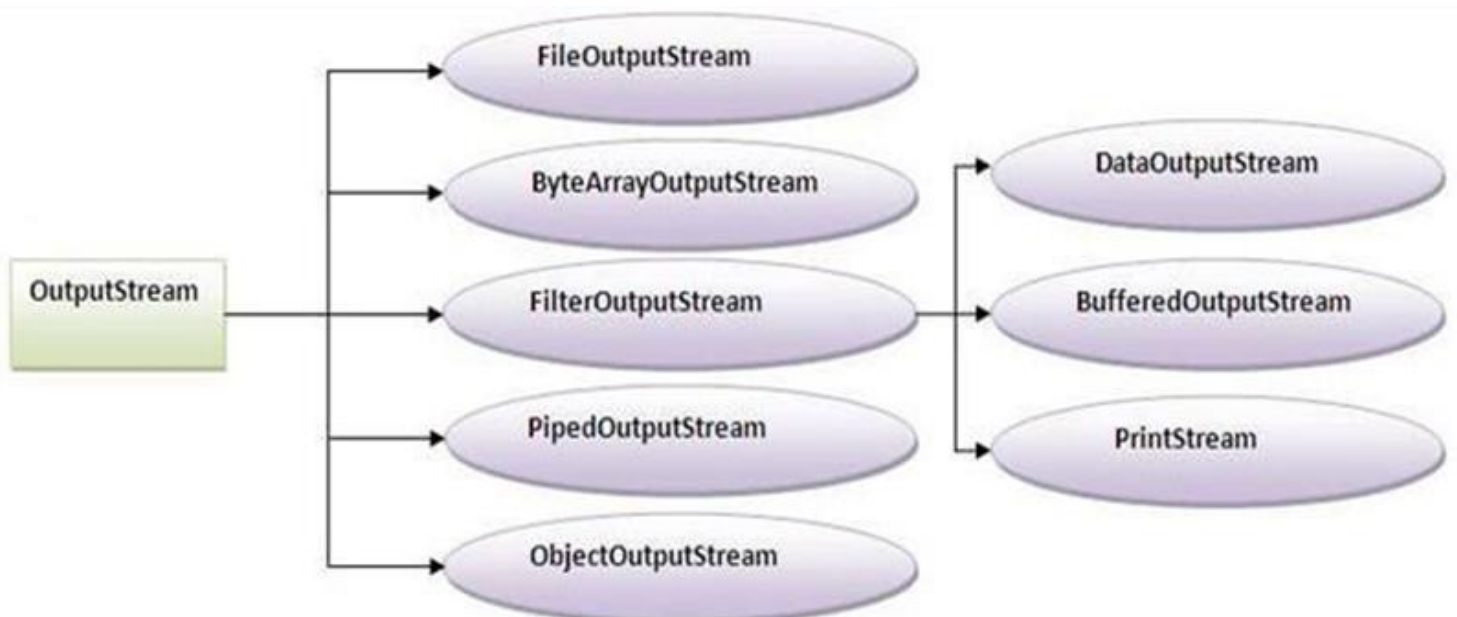


# OUTPUTSTREAM CLASS

- OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.



| Method                                                  | Description                                                     |
|---------------------------------------------------------|-----------------------------------------------------------------|
| 1) <b>public void write(int) throws IOException:</b>    | is used to write a byte to the current output stream.           |
| 2) <b>public void write(byte[]) throws IOException:</b> | is used to write an array of byte to the current output stream. |
| 3) <b>public void flush() throws IOException:</b>       | flushes the current output stream.                              |
| 4) <b>public void close() throws IOException:</b>       | is used to close the current output stream.                     |

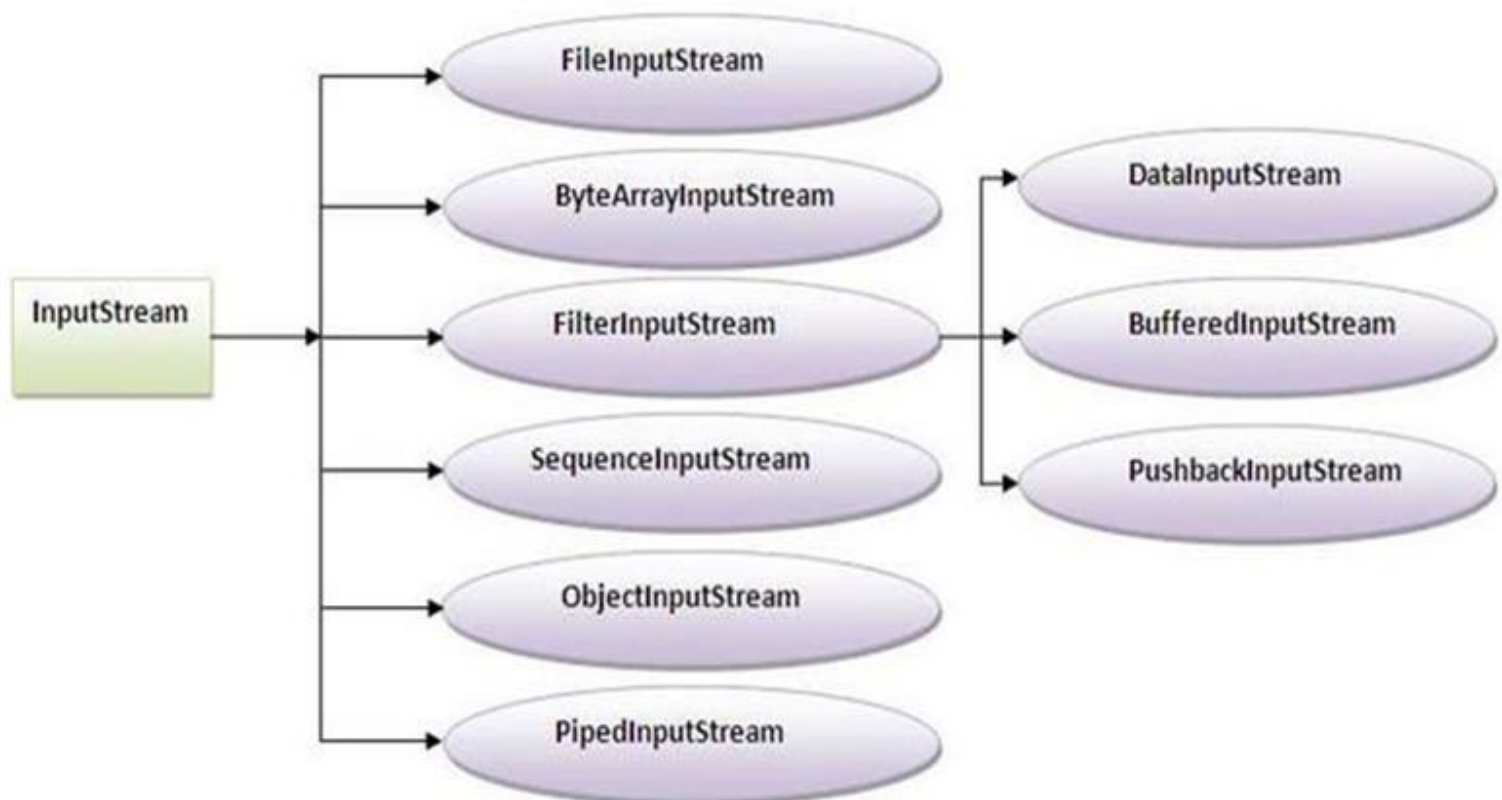


## ◉ InputStream class

- ◉ Input Stream class is an abstract class. It is the superclass of all classes representing an inputstream of bytes.

| Method                                           | Description                                                                                |
|--------------------------------------------------|--------------------------------------------------------------------------------------------|
| 1) public abstract int read()throws IOException: | reads the next byte of data from the input stream. It returns -1 at the end of file.       |
| 2) public int available()throws IOException:     | returns an estimate of the number of bytes that can be read from the current input stream. |
| 3) public void close()throws IOException:        | is used to close the current input stream.                                                 |

### Commonly used methods of InputStream class



# FILEINPUTSTREAM AND FILEOUTPUTSTREAM(FILE HANDLING)

- In Java, FileInputStream and FileOutputStream classes are used to read and write data in file. In another words, they are used for file handling in java.

## JAVA FILEOUTPUTSTREAM CLASS

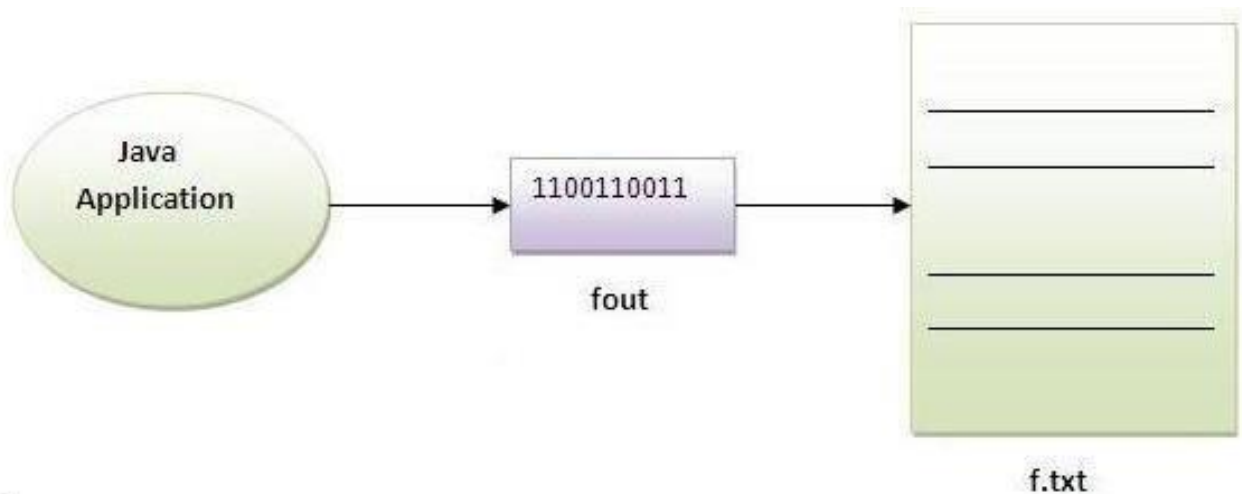
- Java FileOutputStream is an output stream for writing data to a file.
- If you have to write primitive values then use FileOutputStream. Instead, for character-oriented data, prefer FileWriter. But you can write byte-oriented as well as character-oriented data.

# EXAMPLE OF JAVA FILEOUTPUTSTREAM CLASS

- **import** java.io.\*;
- **class** Test{
  - **public static void** main(String args[]){
  - **try**{
- FileOutputStream fout=**new** FileOutputStream("abc.txt");

String s="Sachin Tendulkar is my favourite player";

- **byte** b[]=s.getBytes();//converting string into byte array
- fout.write(b);
- fout.close();
- System.out.println("success...");
- }**catch**(Exception e){system.out.println(e);}
- }
- }



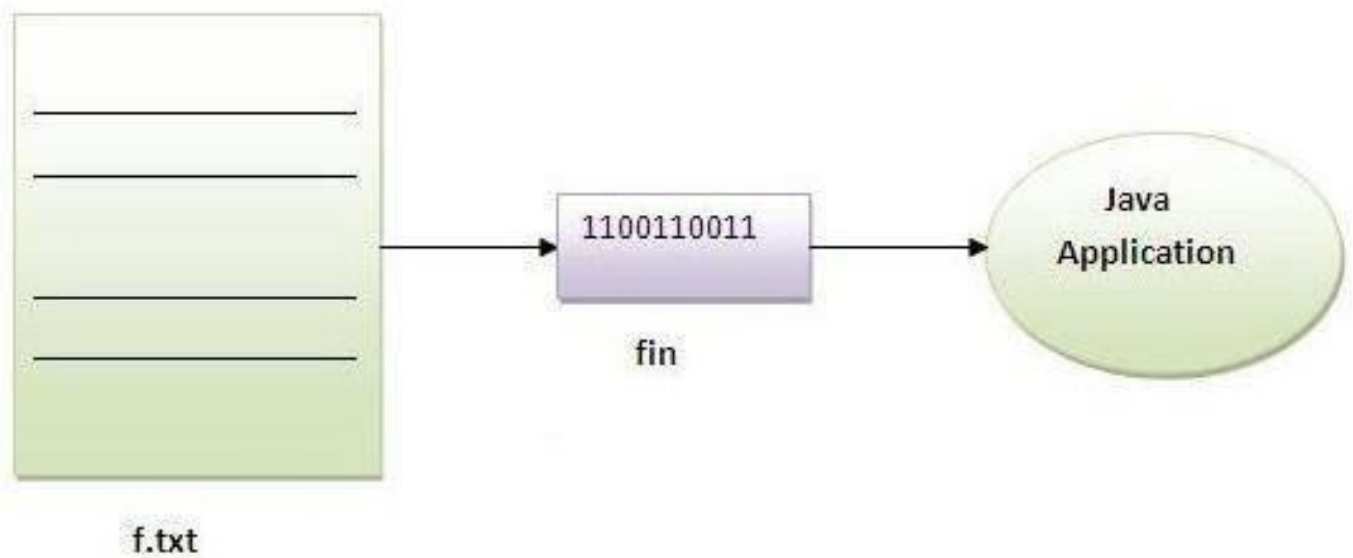
I

## JAVA FILEINPUTSTREAM CLASS

- Java FileInputStream class obtains input bytes from a file. It is used for reading streams of raw bytes such as image data. For reading streams of characters, consider using FileReader.
- It should be used to read byte-oriented data for example to read image, audio, video etc.

# EXAMPLE OF FILEINPUTSTREAM CLASS

- **import** java.io.\*;
- **class** SimpleRead{
  - **public static void** main(String args[]){
  - **try**{
  - FileInputStream fin=**new** FileInputStream("abc.txt");
  - **int** i=0;
  - **while**((i=fin.read())!=-1){
  - System.out.println(**(char)**i);
  - }
  - fin.close();
  - }**catch**(Exception e){system.out.println(e);}
  - }
  - }
- Output:Sachin is my favourite player.



## EXAMPLE OF READING THE DATA OF CURRENT JAVA FILE AND WRITING IT INTO ANOTHER FILE

- We can read the data of any file using the `FileInputStream` class whether it is java file, image file, video file etc. In this example, we are reading the data of C.java file and writing it into another file M.java.



- **import** java.io.\*;
- **class** C{
- **public static void** main(String args[])**throws** Exception{
- FileInputStream fin=**new** FileInputStream("C.java");
- FileOutputStream fout=**new** FileOutputStream("M.java");
- **int** i=0;
- **while**((i=fin.read())!=-1){
- fout.write(**(byte)**i);
- }
- fin.close();
- }
- }



## JAVA BYTEARRAYOUTPUTSTREAM CLASS

- Java ByteArrayOutputStream class is used to write data into multiple files. In this stream, the data is written into a byte array that can be written to multiple stream.
- The ByteArrayOutputStream holds a copy of data and forwards it to multiple streams.
- The buffer of ByteArrayOutputStream automatically grows according to data.
- **Closing the ByteArrayOutputStream has no effect.**

# CONSTRUCTORS OF BYTEARRAYOUTPUTSTREAM CLASS

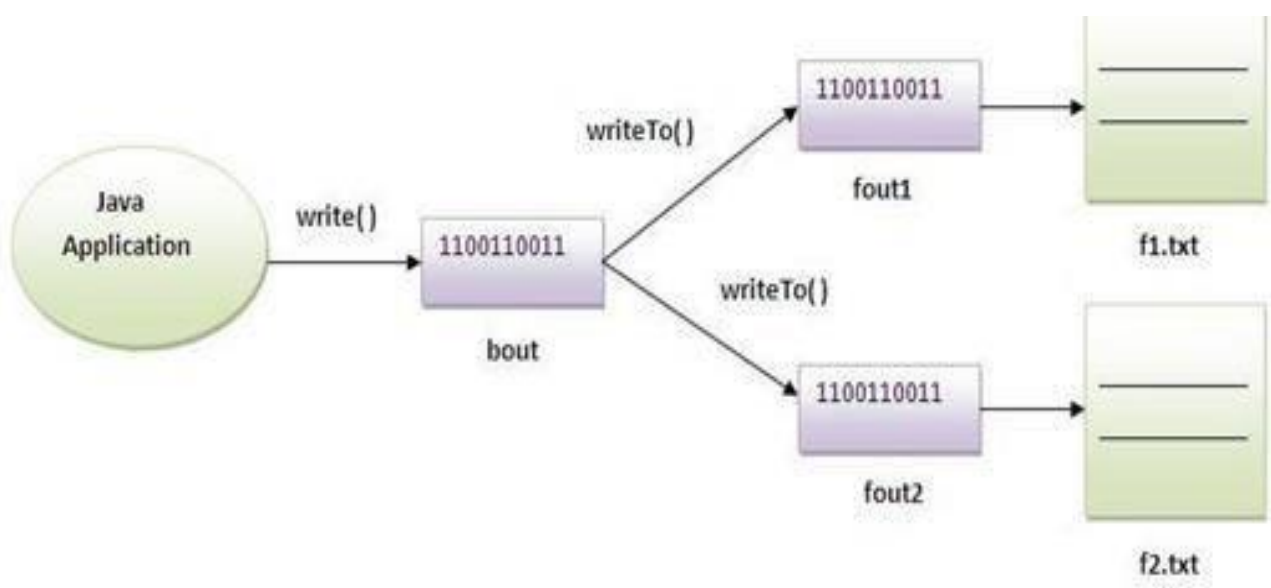
| Constructor                                  | Description                                                                                                           |
|----------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| <code>ByteArrayOutputStream()</code>         | creates a new byte array output stream with the initial capacity of 32 bytes, though its size increases if necessary. |
| <code>ByteArrayOutputStream(int size)</code> | creates a new byte array output stream, with a buffer capacity of the specified size, in bytes.                       |

## METHODS OF BYTEARRAYOUTPUTSTREAM CLASS

| Method                                                                                             | Description                                                                                   |
|----------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------|
| 1) public synchronized void <code>writeTo(OutputStream out)</code> throws <code>IOException</code> | writes the complete contents of this byte array output stream to the specified output stream. |
| 2) public void <code>write(byte b)</code> throws <code>IOException</code>                          | writes byte into this stream.                                                                 |
| 3) public void <code>write(byte[] b)</code> throws <code>IOException</code>                        | writes byte array into this stream.                                                           |
| 4) public void <code>flush()</code>                                                                | flushes this stream.                                                                          |
| 5) public void <code>close()</code>                                                                | has no affect, it doesn't closes the <code>bytearrayoutputstream</code> .                     |

# JAVA BYTEARRAYOUTPUTSTREAM EXAMPLE

- Let's see a simple example of java  
ByteArrayOutputStream class to write data into 2 files.
  
- **import** java.io.\*;
- **class** S{
  - **public static void** main(String args[])**throws**  
Exception{
  - FileOutputStream fout1=**new**  
FileOutputStream("f1.txt");
  - FileOutputStream fout2=**new**  
FileOutputStream("f2.txt");
  - ByteArrayOutputStream bout=**new**  
ByteArrayOutputStream();
  - bout.write(139);
  - bout.writeTo(fout1);
  - bout.writeTo(fout2);
  - bout.flush();
  - bout.close();//has no effect
  - System.out.println("success...");
  - }
- }
- success...



# JAVA SEQUENCEINPUTSTREAM CLASS

- Java SequenceInputStream class is used to read data from multiple streams. It reads data of streams one by one.

## CONSTRUCTORS OF SEQUENCEINPUTSTREAM CLASS:

| Constructor                                                   | Description                                                                                      |
|---------------------------------------------------------------|--------------------------------------------------------------------------------------------------|
| 1) <u>SequenceInputStream(InputStream s1, InputStream s2)</u> | creates a new input stream by reading the data of input stream in order, first s1 and then s2.   |
| 2) <u>SequenceInputStream(Enumeration e)</u>                  | creates a new input stream by reading the data of enumeration whose type is <u>InputStream</u> . |

- Constructors of `SequenceInputStream` class:
- Simple example of `SequenceInputStream` class
- In this example, we are printing the data of two files `f1.txt` and `f2.txt`.

```
○ import java.io.*;
○ class Simple{
 ○ public static void main(String args[])throws
 Exception{
 ○ FileInputStream fin1=new
 FileInputStream("f1.txt");
 ○ FileInputStream fin2=new
 FileInputStream("f2.txt");
 ○ SequenceInputStream sis=new
 SequenceInputStream(fin1,fin2);
 ○ int i;
 ○ while((i=sis.read())!=-1){
 ○ System.out.println((char)i);
 ○ }
 ○ sis.close();
 ○ fin1.close();
 ○ fin2.close();
 ○ }
 ○ }
```

# EXAMPLE OF SEQUENCEINPUTSTREAM THAT READS THE DATA FROM TWO FILES

- In this example, we are writing the data of two files f1.txt and f2.txt into another file named f3.txt.
- //reading data of 2 files and writing it into one file
- **import** java.io.\*;
- **class** Simple{
  - **public static void** main(String args[]) **throws** Exception{
  - **FileInputStream** fin1=**new** **FileInputStream**("f1.txt");
  - **FileInputStream** fin2=**new** **FileInputStream**("f2.txt");
  - **FileOutputStream** fout=**new** **FileOutputStream**("f3.txt");
  - **SequenceInputStream** sis=**new** **SequenceInputStream**(fin1,fin2);
  - **int** i;
  - **while**((i.sisread())!=-1)
  - {
  - fout.write(i);
  - }
  - sis.close();
  - fout.close();
  - fin.close();
  - fin.close();
  - }
  - }

# EXAMPLE OF SEQUENCEINPUTSTREAM CLASS THAT READS THE DATA FROM MULTIPLE FILES USING ENUMERATION

- If we need to read the data from more than two files, we need to have these information in the Enumeration object. Enumeration object can be get by calling elements method of the Vector class. Let's see the simple example where we are reading the data from the 4 files.

- **import** java.io.\*;
- **import** java.util.\*;
- **class** B{
- **public static void** main(String args[]) **throws** IOException{
- //creating the FileInputStream objects for all the files
- FileInputStream fin=**new** FileInputStream("A.java");
- FileInputStream fin2=**new** FileInputStream("abc2.txt");
- FileInputStream fin3=**new** FileInputStream("abc.txt");
- FileInputStream fin4=**new** FileInputStream("B.java");
- //creating Vector object to all the stream
- Vector v=**new** Vector();
- v.add(fin);
- v.add(fin2);
- v.add(fin3);
- v.add(fin4);
- //creating enumeration object by calling the elements method
- Enumeration e=v.elements();
- //passing the enumeration object in the constructor
- SequenceInputStream bin=**new** SequenceInputStream(e);
- **int** i=0;
- **while**((i=bin.read())!=-1){
- System.out.print((**char**)i);
- }
- bin.close();
- fin.close();
- fin2.close();
- }
- }

# JAVA BUFFEREDOUTPUTSTREAM AND BUFFEREDINPUTSTREAM

- Java BufferedOutputStream class
- Java BufferedOutputStream class uses an internal buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.
- Example of BufferedOutputStream class:
- In this example, we are writing the textual information in the BufferedOutputStream object which is connected to the FileOutputStream object. The flush() flushes the data of one stream and send it into another. It is required if you have connected the one stream with another.



- **import** java.io.\*;
- **class** Test{
  - **public static void** main(String args[])**throws** Exception{
  - `FileOutputStream fout=new FileOutputStream("f1.txt");`
  - `BufferedOutputStream bout=new BufferedOutputStream(fout);`
  - `String s="Sachin is my favourite player";`
  - `byte b[]=s.getBytes();`
  - `bout.write(b);`
  - `bout.flush();`
  - `bout.close();`
  - `fout.close();`
  - `System.out.println("success");`
  - `}`
- `}`
- Output:
- success...



## JAVA BUFFEREDINPUTSTREAM CLASS

- Java BufferedInputStream class is used to read information from stream. It internally uses buffer mechanism to make the performance fast.
- Example of Java BufferedInputStream
- Let's see the simple example to read data of file using BufferedInputStream.

- **import** java.io.\*;
- **class** SimpleRead{
  - **public static void** main(String args[]){
  - **try**{
  - FileInputStream fin=**new** FileInputStream("f1.txt");
  - BufferedInputStream bin=**new** BufferedInputStream(fin);
  - **int** i;
  - **while**((i=bin.read())!=-1){
  - System.out.println((**char**)i);
  - }
  - bin.close();
  - fin.close();
  - }**catch**(Exception e){system.out.println(e);}
  - }
- }
- Output:
- Sachin is my favourite player



## JAVA FILEWRITER AND FILEREADER (FILEHANDLING IN JAVA)

- Java FileWriter and FileReader classes are used to write and read data from text files. These are character-oriented classes, used for file handling in java.
- Java has suggested not to use the FileInputStream and FileOutputStream classes if you have to read and write the textual information.

- Java FileWriter class
- Java FileWriter class is used to write character-oriented data to the file.

## CONSTRUCTORS OF FILEWRITER CLASS

| Constructor                          | Description                                           |
|--------------------------------------|-------------------------------------------------------|
| <code>FileWriter(String file)</code> | creates a new file. It gets file name in string.      |
| <code>FileWriter(File file)</code>   | creates a new file. It gets file name in File object. |

## METHODS OF FILEWRITER CLASS

| Method                                         | Description                                      |
|------------------------------------------------|--------------------------------------------------|
| 1) <code>public void write(String text)</code> | writes the string into <code>FileWriter</code> . |
| 2) <code>public void write(char c)</code>      | writes the char into <code>FileWriter</code> .   |
| 3) <code>public void write(char[] c)</code>    | writes char array into <code>FileWriter</code> . |
| 4) <code>public void flush()</code>            | flushes the data of <code>FileWriter</code> .    |
| 5) <code>public void close()</code>            | closes <code>FileWriter</code> .                 |

- Java FileWriter Example
- In this example, we are writing the data in the file abc.txt.

- **import** java.io.\*;
- **class** Simple{
  - **public static void** main(String args[]){
  - **try**{
  - **FileWriter** fw=**new** **FileWriter**("abc.txt");
  - fw.write("my name is sachin");
  - fw.close();
  - }**catch**(Exception e){System.out.println(e);}
  - System.out.println("success");
  - }
- }
- Output:
- success...

## JAVA FILE READER CLASS

- Java FileReader class is used to read data from the file. It returns data in byte format like FileInputStream class.

## CONSTRUCTORS OF FILE WRITER CLASS

| Constructor                          | Description                                                                                                                                    |
|--------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>FileReader(String file)</code> | It gets filename in string. It opens the given file in read mode. If file doesn't exist, it throws <code>FileNotFoundException</code> .        |
| <code>FileReader(File file)</code>   | It gets filename in file instance. It opens the given file in read mode. If file doesn't exist, it throws <code>FileNotFoundException</code> . |

# METHODS OF FILEREADER CLASS

## Methods of FileReader class

| Method                      | Description                                                                 |
|-----------------------------|-----------------------------------------------------------------------------|
| 1) public <u>int</u> read() | <u>returns</u> a character in ASCII form. It returns -1 at the end of file. |
| 2) public void close()      | <u>closes</u> <u>FileReader</u> .                                           |

### ○ Java FileReader Example

- In this example, we are reading the data from the file abc.txt file.

- **import** java.io.\*;

- **class** Simple{

- **public static void** main(String args[]) **throws** Exception{

- **FileReader** fr=**new** **FileReader**("abc.txt");

- **int** i;

- **while**((i=fr.read())!=-1)

- **System.out.println**((**char**)i);

- fr.close();

- }

- }

- Output:

- my name is sachin

# CHARARRAYWRITER CLASS:

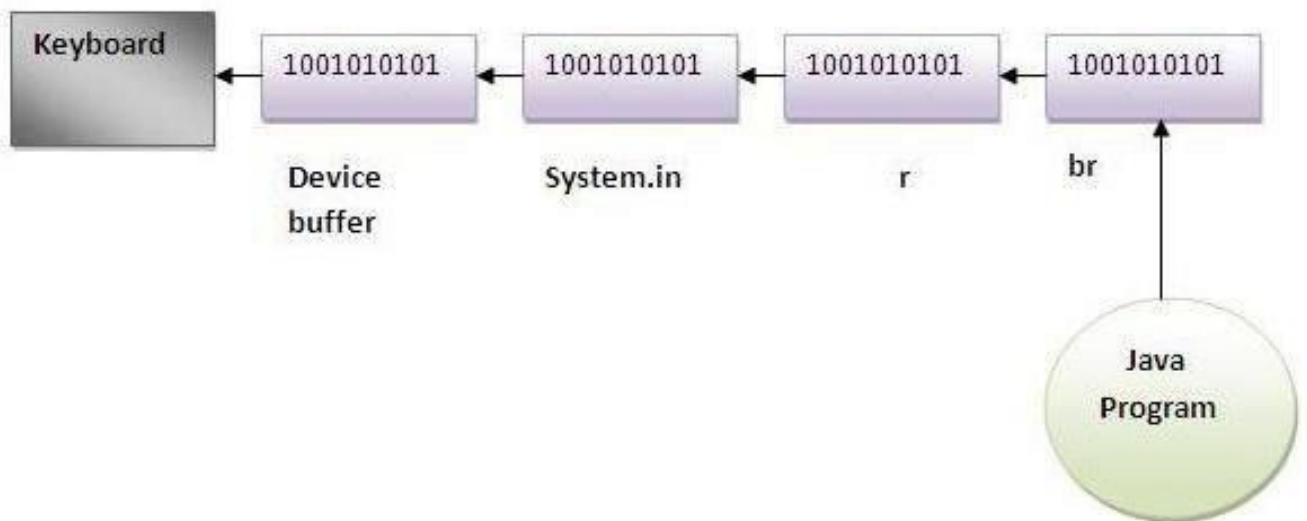
- The CharArrayWriter class can be used to write data to multiple files. This class implements the Appendable interface. Its buffer automatically grows when data is written in this stream. Calling the close() method on this object has no effect.
- Example of CharArrayWriter class:
- In this example, we are writing a common data to 4 files a.txt, b.txt, c.txt and d.txt.

```
import java.io.*;
class Simple{
 public static void main(String args[])throws
 Exception{
 CharArrayWriter out=new CharArrayWriter();
 out.write("my name is");
 FileWriter f1=new FileWriter("a.txt");
 FileWriter f2=new FileWriter("b.txt");
 FileWriter f3=new FileWriter("c.txt");
 FileWriter f4=new FileWriter("d.txt");
 out.writeTo(f1);
 out.writeTo(f2);
 out.writeTo(f3);
 out.writeTo(f4);
 f1.close();
 f2.close();
 f3.close();
 f4.close();
 }
}
```

# READING DATA FROM KEYBOARD

- There are many ways to read data from the keyboard. For example:
  1. InputStreamReader
  2. Console
  3. Scanner
  4. DataInputStream etc.
  
- InputStreamReader class
- InputStreamReader class can be used to read data from keyboard. It performs two tasks:
  - connects to input stream of keyboard
  - converts the byte-oriented stream into character-oriented stream
- BufferedReader class
- BufferedReader class can be used to read dataline by line by readLine() method.
  
- Example of reading data from keyboard by InputStreamReader and BufferedReader class
  
- In this example, we are connecting the BufferedReader stream with the InputStreamReader stream for reading the line by line data from the keyboard.

- **import** java.io.\*;
- **class** G5{
- **public static void** main(String args[])**throws** Exception{
- 
- **InputStreamReader** r=**new** **InputStreamReader**(System.in);
- **BufferedReader** br=**new** **BufferedReader**(r);
- 
- System.out.println("Enter your name");
- String name=br.readLine();
- System.out.println("Welcome "+name);
- }
- }
- Output:Enter your name
- Arjun
- Welcome Arjun





- Another Example of reading data from keyboard by InputStreamReader and BufferedReader class until the user writes stop
- In this example, we are reading and printing the data until the user prints stop.

- **import** java.io.\*;
- **class** G5{
- **public static void** main(String args[]) **throws** Exception{
- **InputStreamReader** r=**new** InputStreamReader(System.in);
- **BufferedReader** br=**new** BufferedReader(r);
- **String** name="";
- **while**(!name.equals("stop")){
- System.out.println("Enter data: ");
- name=br.readLine();
- System.out.println("data is: "+name);
- }
- br.close();
- r.close();
- }
- }
- **Output:**Enter data: Arjun
- data is: Arjun
- Enter data: 10
- data is: 10
- Enter data: stop
- data is: stop

- Java Console class
- The Java Console class is used to get input from console. It provides methods to read text and password.
- If you read password using Console class, it will not be displayed to the user.
- The `java.io.Console` class is attached with system console internally. The Console class is introduced since 1.5.
- Let's see a simple example to read text from console.
- `String text=System.console().readLine();`
- `System.out.println("Text is: "+text);`

## METHODS OF CONSOLE CLASS

| Method                                                                 | Description                                                                                        |
|------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| 1) <code>public String readLine()</code>                               | is used to read a single line of text from the console.                                            |
| 2) <code>public String readLine(String fmt, Object... args)</code>     | it provides a formatted prompt then reads the single line of text from the console.                |
| 3) <code>public char[] readPassword()</code>                           | is used to read password that is not being displayed on the console.                               |
| 4) <code>public char[] readPassword(String fmt, Object... args)</code> | it provides a formatted prompt then reads the password that is not being displayed on the console. |

# HOW TO GET THE OBJECT OF CONSOLE

- System class provides a static method `console()` that returns the unique instance of Console class.
- **public static** Console `console()`{}
- Let's see the code to get the instance of Console class.
- `Console c=System.console();`

## JAVA CONSOLE EXAMPLE

- **import** java.io.\*;
- **class** ReadStringTest{
- **public static void** main(String args[]){
- `Console c=System.console();`
- `System.out.println("Enter your name: ");`
- `String n=c.readLine();`
- `System.out.println("Welcome "+n);`
- }
- }
- Output:
- Enter your name: Arjun
- Welcome Arjun

## JAVA CONSOLE EXAMPLE TO READ PASSWORD

- **import** java.io.\*;
- **class** ReadPasswordTest{
- **public static void** main(String args[]){
- Console c=System.console();
- System.out.println("Enter password: ");
- **char**[] ch=c.readPassword();
- String pass=String.valueOf(ch);//converting char array into string
- System.out.println("Password is: "+pass);
- }
- }
- Output:
- Enter password:
- Password is: sonoo

## JAVA SCANNER CLASS

- There are various ways to read input from the keyboard, the java.util.Scanner class is one of them.
- The **Java Scanner** class breaks the input into tokens using a delimiter that is whitespace by default. It provides many methods to read and parse various primitive values.
- Java Scanner class is widely used to parse text for string and primitive types using regular expression.
- Java Scanner class extends Object class and implements Iterator and Closeable interfaces.

- Commonly used methods of Scanner class
- There is a list of commonly used Scanner class methods:

| Method                                         | Description                                                                       |
|------------------------------------------------|-----------------------------------------------------------------------------------|
| <code>public String next()</code>              | it returns the next token from the scanner.                                       |
| <code>public String <u>nextLine()</u></code>   | it moves the scanner position to the next line and returns the value as a string. |
| <code>public byte <u>nextByte()</u></code>     | it scans the next token as a byte.                                                |
| <code>public short <u>nextShort()</u></code>   | it scans the next token as a short value.                                         |
| <code>public int <u>nextInt()</u></code>       | it scans the next token as an int value.                                          |
| <code>public long <u>nextLong()</u></code>     | it scans the next token as a long value.                                          |
| <code>public float <u>nextFloat()</u></code>   | it scans the next token as a float value.                                         |
| <code>public double <u>nextDouble()</u></code> | it scans the next token as a double value.                                        |

- The **Java Scanner** class breaks the input into tokens using a delimiter that is whitespace by default. It provides many methods to read and parse various primitive values.
- Java Scanner class is widely used to parse text for string and primitive types using regular expression.
- Java Scanner class extends Object class and implements Iterator and Closeable interfaces.

- Commonly used methods of Scanner class
- There is a list of commonly used Scanner class methods:
- Java Scanner Example to get input from console
- Let's see the simple example of the Java Scanner class which reads the int, string and double values as an input:

```

○ import java.util.Scanner;
○ class ScannerTest{
 ○ public static void main(String args[]){
 ○ Scanner sc=new Scanner(System.in);
 ○
 ○ System.out.println("Enter your rollno");
 ○ int rollno=sc.nextInt();
 ○ System.out.println("Enter your name");
 ○ String name=sc.next();
 ○ System.out.println("Enter your fee");
 ○ double fee=sc.nextDouble();
 ○ System.out.println("Rollno:"+rollno+"
name:"+name+" fee:"+fee);
 ○ sc.close();
 ○ }
 ○ }

```

```

○ Enter your rollno 111
Enter your name Ratan Enter 450000
Rollno:111 name:Ratan fee:450000

```

- Java Scanner Example with delimiter
- Let's see the example of Scanner class with delimiter. The \s represents whitespace.

- **import** java.util.\*;
- **public class** ScannerTest2{
- **public static void** main(String args[]){
  - String input = "10 tea 20 coffee 30 tea biscuits";
  - Scanner s = **new** Scanner(input).useDelimiter("\\s");
  - System.out.println(s.nextInt());
  - System.out.println(s.next());
  - System.out.println(s.nextInt());
  - System.out.println(s.next());
  - s.close();
- }}
- Output:
- 10tea20coffee

- java.io.PrintStream class:
- The PrintStream class provides methods to write data to another stream. The PrintStream class automatically flushes the data so there is no need to call flush() method. Moreover, its methods don't throw IOException.

- There are many methods in `PrintStream` class. Let's see commonly used methods of `PrintStream` class:
- **`public void print(boolean b)`**: it prints the specified boolean value.
- **`public void print(char c)`**: it prints the specified char value.
- **`public void print(char[] c)`**: it prints the specified character array values.
- **`public void print(int i)`**: it prints the specified int value.
- **`public void print(long l)`**: it prints the specified long value.
- **`public void print(float f)`**: it prints the specified float value.
- **`public void print(double d)`**: it prints the specified double value.
- **`public void print(String s)`**: it prints the specified string value.
- **`public void print(Object obj)`**: it prints the specified object value.
- **`public void println(boolean b)`**: it prints the specified boolean value and terminates the line.
- **`public void println(char c)`**: it prints the specified char value and terminates the line.
- **`public void println(char[] c)`**: it prints the specified character array values and terminates the line.
- **`public void println(int i)`**: it prints the specified int value and terminates the line.
- **`public void println(long l)`**: it prints the specified long value and terminates the line.
- **`public void println(float f)`**: it prints the specified float value and terminates the line.
- **`public void println(double d)`**: it prints the specified double value and terminates the line.
- **`public void println(String s)`**: it prints the specified string value and terminates the line.
- **`public void println(Object obj)`**: it prints the specified object value and terminates the line.



- **public void println():** it terminates the line only.
- **public void printf(Object format, Object... args):** it writes the formatted string to the current stream.
- **public void printf(Locale l, Object format, Object... args):** it writes the formatted string to the current stream.
- **public void format(Object format, Object... args):** it writes the formatted string to the current stream using specified format.
- **public void format(Locale l, Object format, Object... args):** it writes the formatted string to the current stream using specified format.

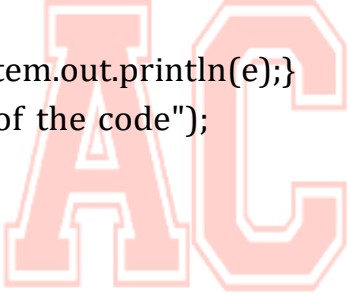
## EXAMPLE OF

## JAVA.IO.PRINTSTREAMCLASS:

- In this example, we are simply printing integer and string values.
- **import java.io.\*;**
- **class PrintStreamTest{**
  - **public static void main(String args[]) throws Exception{**
    - **FileOutputStream fout=new**  
**FileOutputStream("mfile.txt");**
    - **PrintStream pout=new PrintStream(fout);**
    - **pout.println(1900);**
    - **pout.println("Hello Java");**
    - **pout.println("Welcome to Java");**
    - **pout.close();**
    - **fout.close();**
    - 
    - **}**
    - **}**

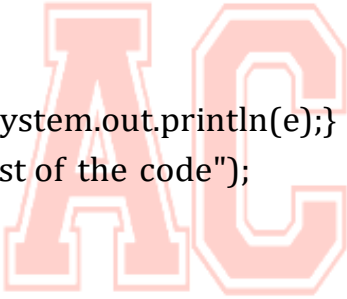
- Example of printf() method of java.io.OutputStream class:
- Let's see the simple example of printing integer value by format specifier.
- **class** OutputStreamTest{
  - **public static void** main(String args[]){
  - **int** a=10;
  - System.out.printf("%d",a); //Note, out is the object of OutputStream class
  - }
  - }
- Output:10
- Compressing and Uncompressing File
- The DeflaterOutputStream and InflaterInputStream classes provide mechanism to compress and uncompress the data in the **deflate compression format**.
- DeflaterOutputStream class:
- The DeflaterOutputStream class is used to compress the data in the deflate compression format. It provides facility to the other compression filters, such as GZIPOutputStream.
- Example of Compressing file using DeflaterOutputStream.
- In this example, we are reading data of a file and compressing it into another file using DeflaterOutputStream class. You can compress any file, here we are compressing the Deflater.java file

- **import** java.io.\*;
- **import** java.util.zip.\*;
- **class** Compress{
- **public static void** main(String args[]){
- **try**{
- FileInputStream fin=**new** FileInputStream("Deflater.java");
- FileOutputStream fout=**new** FileOutputStream("def.txt");
- DeflaterOutputStream out=**new** DeflaterOutputStream(fout);
- **int** i;
- **while**((i=fin.read())!=-1){
- out.write(**(byte)**i);
- out.flush();
- }
- fin.close();
- out.close();
- }**catch**(Exception e){System.out.println(e);}
- System.out.println("rest of the code");
- }
- }



## ○ InflaterInputStream class:

- The InflaterInputStream class is used to uncompress the file in the deflate compression format. It provides facility to the other uncompression filters, such as GZIPInputStream class.
- Example of uncompressing file using InflaterInputStream class
- In this example, we are decompressing the compressed file def.txt into D.java .



```
○ import java.io.*;
○ import java.util.zip.*;
○ class UnCompress{
○ public static void main(String args[]){
○ try{
○ FileInputStream fin=new FileInputStream("def.txt");
○ InflaterInputStream in=new InflaterInputStream(fin);
○ FileOutputStream fout=new FileOutputStream("D.java");
○ int i;
○ while((i=in.read())!=-1){
○ fout.write((byte)i);
○ fout.flush();
○ }
○ fin.close();
○ fout.close();
○ in.close();
○ }catch(Exception e){System.out.println(e);}
○ System.out.println("rest of the code");
○ }
○ }
```

- Java provides strong but flexible support for I/O related to Files and networks but this tutorial covers very basic functionality related to streams and I/O. We would see most commonly used example one by one:

- Byte Streams

- Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are

, **FileInputStream** and **FileOutputStream**.

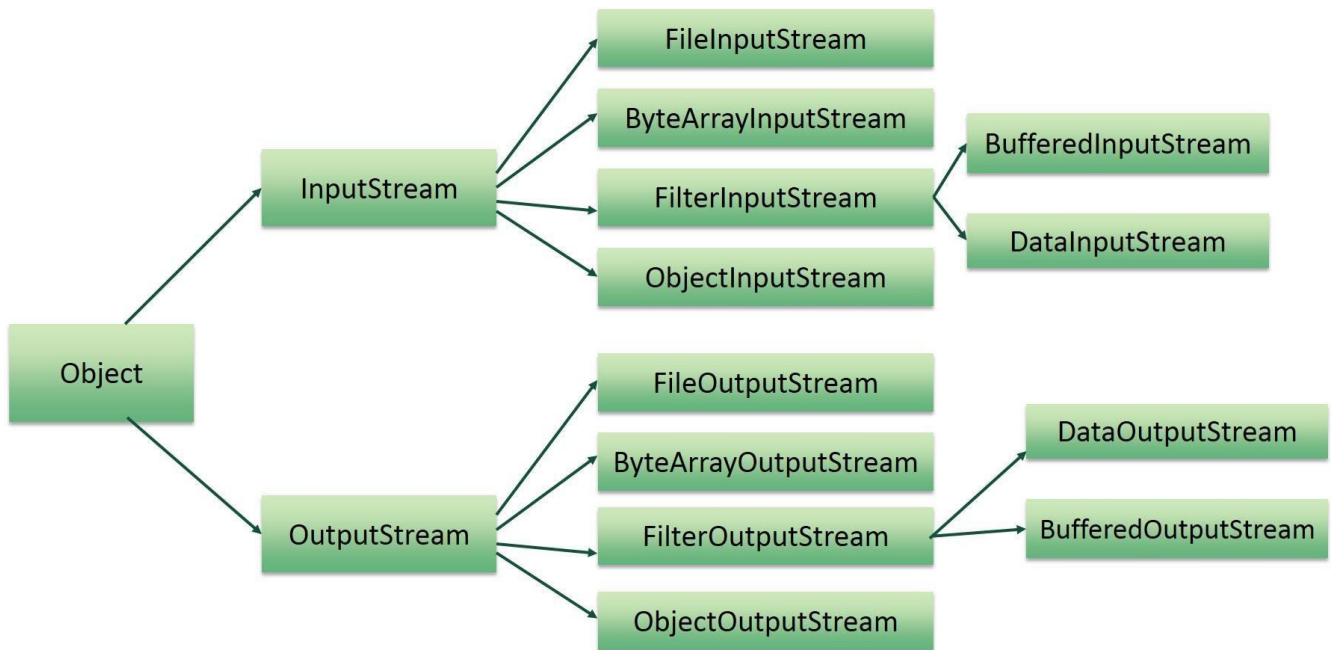
Following is an example which makes use of these two classes to copy an input file into an output file:



○ import  
java.io.\*;  
public  
class  
CopyFil  
e {  
public static void main(String args[]) throws IOException {  
FileReader in = null;  
FileWri  
ter out  
=  
null;try  
{  
in = new  
FileReader("input.txt")  
; out = new  
FileWriter("output.txt"  
);int c;  
while ((c = in.read())  
!= -1) {out.write(c);  
}}  
finally {  
if (in != null) {  
in.close();}  
if (out  
!= null)  
{  
out.clos  
e();  
}}}}}



- ◉ Reading and Writing Files:
- ◉ As described earlier, A stream can be defined as a sequence of data. The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination.
- ◉ Here is a hierarchy of classes to deal with Input and Output streams.



- ◉ The two important streams are **FileInputStream** and **FileOutputStream**, which would be discussed in this tutorial:
- ◉ **FileInputStream**:
- ◉ This stream is used for reading data from the files. Objects can be created using the keyword `new` and there are several types of constructors available.
- ◉ Following constructor takes a file name as a string to create an input stream object to read the file.:
- ◉ `InputStream f = new  
FileInputStream("C:/java/hello");`

- Following constructor takes a file object to create an input stream object to read the file. First we create a file object using `File()` method as follows:
- `File f = new File("C:/java/hello");`  
`InputStream f = new FileInputStream(f);`  
Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

| SN | Methods with Description                                                                                                                                                                                                                                                                       |
|----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1  | <b><code>public void close() throws IOException{}</code></b><br>This method closes the file output stream. Releases any system resources associated with the file. Throws an <code>IOException</code> .                                                                                        |
| 2  | <b><code>protected void finalize()throws IOException {}</code></b><br>This method cleans up the connection to the file. Ensures that the <code>close</code> method of this file output stream is called when there are no more references to this stream. Throws an <code>IOException</code> . |
| 3  | <b><code>public int read(int r)throws IOException{}</code></b><br>This method reads the specified byte of data from the <code>InputStream</code> . Returns an <code>int</code> . Returns the next byte of data and -1 will be returned if it's end of file.                                    |
| 4  | <b><code>public int read(byte[] r) throws IOException{}</code></b><br>This method reads <code>r.length</code> bytes from the input stream into an array. Returns the total number of bytes read. If end of file -1 will be returned.                                                           |
| 5  | <b><code>public int available() throws IOException{}</code></b><br>Gives the number of bytes that can be read from this file input stream. Returns an <code>int</code> .                                                                                                                       |



AC

THE END