# **JVM (Java Virtual Machine) Architecture**

* Java Virtual Machine
* Internal Architecture of JVM

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

### What is JVM

It is:

1. **A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.
2. **An implementation** Its implementation is known as JRE (Java Runtime Environment).
3. **Runtime Instance** Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

### What it does

The JVM performs following operation:

* Loads code
* Verifies code
* Executes code
* Provides runtime environment

JVM provides definitions for the:

* Memory area
* Class file format
* Register set
* Garbage-collected heap
* Fatal error reporting etc.

## JVM Architecture

Let's understand the internal architecture of JVM. It contains classloader, memory area, execution engine etc.



### 1) Classloader

Classloader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader. There are three built-in classloaders in Java.

1. **Bootstrap ClassLoader**: This is the first classloader which is the super class of Extension classloader. It loads the rt.jar file which contains all class files of Java Standard Edition like java.lang package classes, java.net package classes, java.util package classes, java.io package classes, java.sql package classes etc.
2. **Extension ClassLoader**: This is the child classloader of Bootstrap and parent classloader of System classloader. It loades the jar files located inside $JAVA\_HOME/jre/lib/ext directory.
3. **System/Application ClassLoader**: This is the child classloader of Extension classloader. It loads the classfiles from classpath. By default, classpath is set to current directory. You can change the classpath using "-cp" or "-classpath" switch. It is also known as Application classloader.
4. //Let's see an example to print the classloader name
5. **public** **class** ClassLoaderExample
6. {
7. **public** **static** **void** main(String[] args)
8. {
9. // Let's print the classloader name of current class.
10. //Application/System classloader will load this class
11. Class c=ClassLoaderExample.**class**;
12. System.out.println(c.getClassLoader());
13. //If we print the classloader name of String, it will print null because it is an
14. //in-built class which is found in rt.jar, so it is loaded by Bootstrap classloader
15. System.out.println(String.**class**.getClassLoader());
16. }
17. }

Output:

sun.misc.Launcher$AppClassLoader@4e0e2f2a

null

These are the internal classloaders provided by Java. If you want to create your own classloader, you need to extend the ClassLoader class.

### 2) Class(Method) Area

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

### 3) Heap

It is the runtime data area in which objects are allocated.

### 4) Stack

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.

Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

### 5) Program Counter Register

PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.

### 6) Native Method Stack

It contains all the native methods used in the application.

### 7) Execution Engine

It contains:

1. **A virtual processor**
2. **Interpreter:** Read bytecode stream then execute the instructions.
3. **Just-In-Time(JIT) compiler:** It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here, the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

### 8) Java Native Interface

Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc. Java uses JNI framework to send output to the Console or interact with OS libraries.

# **Java Variables**

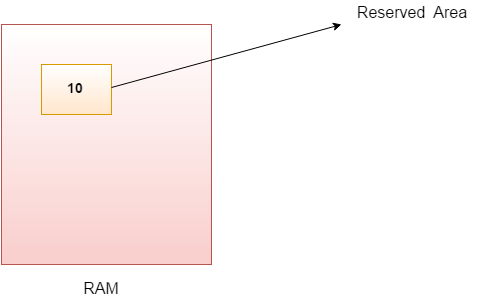
A variable is a container which holds the value while the java program is executed. A variable is assigned with a datatype.

Variable is a name of memory location. There are three types of variables in java: local, instance and static.

There are two types of data types in java: primitive and non-primitive.

## Variable

**Variable** is name of reserved area allocated in memory. In other words, it is a name of memory location. It is a combination of "vary + able" that means its value can be changed.

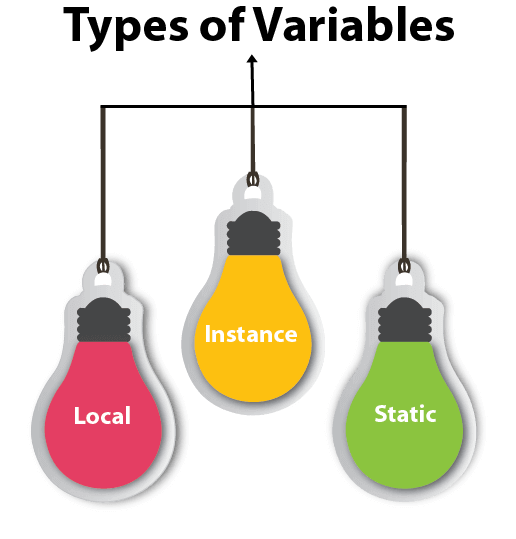


1. **int** data=50;//Here data is variable

### Types of Variables

There are three types of variables in java:

* local variable
* instance variable
* static variable



#### **1) Local Variable**

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

#### **2) Instance Variable**

A variable declared inside the class but outside the body of the method, is called instance variable. It is not declared as static.

It is called instance variable because its value is instance specific and is not shared among instances.

#### **3) Static variable**

A variable which is declared as static is called static variable. It cannot be local. You can create a single copy of static variable and share among all the instances of the class. Memory allocation for static variable happens only once when the class is loaded in the memory.

### Example to understand the types of variables in java

1. **class** A{
2. **int** data=50;//instance variable
3. **static** **int** m=100;//static variable
4. **void** method(){
5. **int** n=90;//local variable
6. }
7. }//end of class

### Java Variable Example: Add Two Numbers

1. **class** Simple{
2. **public** **static** **void** main(String[] args){
3. **int** a=10;
4. **int** b=10;
5. **int** c=a+b;
6. System.out.println(c);
7. }}

Output:

20

### Java Variable Example: Widening

1. **class** Simple{
2. **public** **static** **void** main(String[] args){
3. **int** a=10;
4. **float** f=a;
5. System.out.println(a);
6. System.out.println(f);
7. }}

Output:

10

10.0

### Java Variable Example: Narrowing (Typecasting)

1. **class** Simple{
2. **public** **static** **void** main(String[] args){
3. **float** f=10.5f;
4. //int a=f;//Compile time error
5. **int** a=(**int**)f;
6. System.out.println(f);
7. System.out.println(a);
8. }}

Output:

10.5

10

### Java Variable Example: Overflow

1. **class** Simple{
2. **public** **static** **void** main(String[] args){
3. //Overflow
4. **int** a=130;
5. **byte** b=(**byte**)a;
6. System.out.println(a);
7. System.out.println(b);
8. }}

Output:

130

-126

### Java Variable Example: Adding Lower Type

1. **class** Simple{
2. **public** **static** **void** main(String[] args){
3. **byte** a=10;
4. **byte** b=10;
5. //byte c=a+b;//Compile Time Error: because a+b=20 will be int
6. **byte** c=(**byte**)(a+b);
7. System.out.println(c);
8. }}

Output:

20

# **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:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

### Data Types

Data types are divided into the following types:

-     Integer types - **byte** , **short** , **int** , **long** ;

-     Real floating point types - **float** and **double** ;

-     Boolean type - **boolean** ;

-     Character type - **char** ;

-     Object type - **Object** ;

-     Symbolic strings - **String** .

In the following table, we can see the types of data listed above ( **byte** , **short** , **int** , **long** , **float** , **double** , **boolean** , **char** , **Object,** and **String** ), including their default values ​​and their scope:

|  |  |  |  |
| --- | --- | --- | --- |
| **Data type** | **The default value** | **Minimum value** | **Maximum value** |
| byte | 0 | -128 | 127 |
| short | 0 | -32768 | 32767 |
| int | 0 | -2147483648 | 2147483647 |
| long | 0L | -9.22337E+18 | 9.22337E+18 |
| float | 0.0f | -3.4E + 38 | + 3.4E + 38 |
| double | 0.0d | -1.7E + 308 | + 1.7E + 308 |
| boolean | FALSE | The possible values ​​are two - true or false | |
| char | '\ u0000' | 0 | 65535 |
| Object | null |  |  |
| String | null |  |  |

## 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.

Java is a statically-typed programming language. It means, all variables must be declared before its use. That is why we need to declare variable's type and name.

There are 8 types of primitive data types:

* boolean data type
* byte data type
* char data type
* short data type
* int data type
* long data type
* float data type
* 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 isan 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 - 2,147,483,648and 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 - 9,223,372,036,854,775,808and 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. To get detail explanation about Unicode visit next page.

# **Modifier Types**

Modifiers are keywords that you add to those definitions to change their meanings. Java language has a wide variety of modifiers, including the following −

* Java Access Modifiers
* Non Access Modifiers

To use a modifier, you include its keyword in the definition of a class, method, or variable. The modifier precedes the rest of the statement, as in the following example.

### Example

*public* class className {

// ...

}

*private* boolean myFlag;

*static final* double weeks = 9.5;

*protected static final* int BOXWIDTH = 42;

*public static* void main(String[] arguments) {

// body of method

}

## Access Control Modifiers

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are −

* Visible to the package, the default. No modifiers are needed.
* Visible to the class only (private).
* Visible to the world (public).
* Visible to the package and all subclasses (protected).

## Non-Access Modifiers

Java provides a number of non-access modifiers to achieve many other functionality.

* The *static* modifier for creating class methods and variables.
* The *final* modifier for finalizing the implementations of classes, methods, and variables.
* The *abstract* modifier for creating abstract classes and methods.
* The *synchronized* and *volatile* modifiers, which are used for threads.

# **Operators in java**

**Operator** in java is a symbol that is used to perform operations. For example: +, -, \*, / etc.

There are many types of operators in java which are given below:

* Unary Operator,
* Arithmetic Operator,
* Shift Operator,
* Relational Operator,
* Bitwise Operator,
* Logical Operator,
* Ternary Operator and
* Assignment Operator.

## Java Operator Precedence

|  |  |  |
| --- | --- | --- |
| **Operator Type** | **Category** | **Precedence** |
| Unary | postfix | expr++ expr-- |
| prefix | ++expr --expr +expr -expr ~ ! |
| 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 --

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** x=10;
4. System.out.println(x++);//10 (11)
5. System.out.println(++x);//12
6. System.out.println(x--);//12 (11)
7. System.out.println(--x);//10
8. }}

Output:

10

12

12

10

### Java Unary Operator Example 2: ++ and --

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=10;
5. System.out.println(a++ + ++a);//10+12=22
6. System.out.println(b++ + b++);//10+11=21
8. }}

Output:

22

21

### Java Unary Operator Example: ~ and !

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

Output:

-11

9

false

true

### Java Arithmetic Operators

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

### Java Arithmetic Operator Example

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=5;
5. System.out.println(a+b);//15
6. System.out.println(a-b);//5
7. System.out.println(a\*b);//50
8. System.out.println(a/b);//2
9. System.out.println(a%b);//0
10. }}

Output:

15

5

50

2

0

### Java Arithmetic Operator Example: Expression

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

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

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. System.out.println(10<<2);//10\*2^2=10\*4=40
4. System.out.println(10<<3);//10\*2^3=10\*8=80
5. System.out.println(20<<2);//20\*2^2=20\*4=80
6. System.out.println(15<<4);//15\*2^4=15\*16=240
7. }}

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

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. System.out.println(10>>2);//10/2^2=10/4=2
4. System.out.println(20>>2);//20/2^2=20/4=5
5. System.out.println(20>>3);//20/2^3=20/8=2
6. }}

Output:

2

5

2

### Java Shift Operator Example: >> vs >>>

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. //For positive number, >> and >>> works same
4. System.out.println(20>>2);
5. System.out.println(20>>>2);
6. //For negative number, >>> changes parity bit (MSB) to 0
7. System.out.println(-20>>2);
8. System.out.println(-20>>>2);
9. }}

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.

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=5;
5. **int** c=20;
6. System.out.println(a<b&&a<c);//false && true = false
7. System.out.println(a<b&a<c);//false & true = false
8. }}

Output:

false

false

### Java AND Operator Example: Logical && vs Bitwise &

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=5;
5. **int** c=20;
6. System.out.println(a<b&&a++<c);//false && true = false
7. System.out.println(a);//10 because second condition is not checked
8. System.out.println(a<b&a++<c);//false && true = false
9. System.out.println(a);//11 because second condition is checked
10. }}

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.

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=5;
5. **int** c=20;
6. System.out.println(a>b||a<c);//true || true = true
7. System.out.println(a>b|a<c);//true | true = true
8. //|| vs |
9. System.out.println(a>b||a++<c);//true || true = true
10. System.out.println(a);//10 because second condition is not checked
11. System.out.println(a>b|a++<c);//true | true = true
12. System.out.println(a);//11 because second condition is checked
13. }}

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

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=2;
4. **int** b=5;
5. **int** min=(a<b)?a:b;
6. System.out.println(min);
7. }}

Output:

2

Another Example:

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=5;
5. **int** min=(a<b)?a:b;
6. System.out.println(min);
7. }}

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

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=20;
5. a+=4;//a=a+4 (a=10+4)
6. b-=4;//b=b-4 (b=20-4)
7. System.out.println(a);
8. System.out.println(b);
9. }}

Output:

14

16

### Java Assignment Operator Example

1. **class** OperatorExample{
2. **public** **static** **void** main(String[] args){
3. **int** a=10;
4. a+=3;//10+3
5. System.out.println(a);
6. a-=4;//13-4
7. System.out.println(a);
8. a\*=2;//9\*2
9. System.out.println(a);
10. a/=2;//18/2
11. System.out.println(a);
12. }}

Output:

13

9

18

9

### Java Assignment Operator Example: Adding short

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **short** a=10;
4. **short** b=10;
5. //a+=b;//a=a+b internally so fine
6. a=a+b;//Compile time error because 10+10=20 now int
7. System.out.println(a);
8. }}

Output:

Compile time error

After type cast:

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **short** a=10;
4. **short** b=10;
5. a=(**short**)(a+b);//20 which is int now converted to short
6. System.out.println(a);
7. }}

Output:

20

# **Java Keywords**

**Java keywords** are also known as **reserved words**. Keywords are particular words which acts as a key to a code. These are predefined words by Java so it cannot be used as a variable or object name.

## List of Java Keywords

A list of Java keywords or reserved words are given below:

1. abstract: Java abstract keyword is used to declare abstract class. Abstract class can provide the implementation of interface. It can have abstract and non-abstract methods.
2. boolean: Java boolean keyword is used to declare a variable as a boolean type. It can hold True and False values only.
3. break: Java break keyword is used to break loop or switch statement. It breaks the current flow of the program at specified condition.
4. byte: Java byte keyword is used to declare a variable that can hold an 8-bit data values.
5. case: Java case keyword is used to with the switch statements to mark blocks of text.
6. catch: Java catch keyword is used to catch the exceptions generated by try statements. It must be used after the try block only.
7. char: Java char keyword is used to declare a variable that can hold unsigned 16-bit Unicode characters
8. class: Java class keyword is used to declare a class.
9. continue: Java continue keyword is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition.
10. default: Java default keyword is used to specify the default block of code in a switch statement.
11. do: Java do keyword is used in control statement to declare a loop. It can iterate a part of the program several times.
12. double: Java double keyword is used to declare a variable that can hold a 64-bit floating-point numbers.
13. else: Java else keyword is used to indicate the alternative branches in an if statement.
14. enum: Java enum keyword is used to define a fixed set of constants. Enum constructors are always private or default.
15. extends: Java extends keyword is used to indicate that a class is derived from another class or interface.
16. final: Java final keyword is used to indicate that a variable holds a constant value. It is applied with a variable. It is used to restrict the user.
17. finally: Java finally keyword indicates a block of code in a try-catch structure. This block is always executed whether exception is handled or not.
18. float: Java float keyword is used to declare a variable that can hold a 32-bit floating-point number.
19. for: Java for keyword is used to start a for loop. It is used to execute a set of instructions/functions repeatedly when some conditions become true. If the number of iteration is fixed, it is recommended to use for loop.
20. if: Java if keyword tests the condition. It executes the if block if condition is true.
21. implements: Java implements keyword is used to implement an interface.
22. import: Java import keyword makes classes and interfaces available and accessible to the current source code.
23. instanceof: Java instanceof keyword is used to test whether the object is an instance of the specified class or implements an interface.
24. int: Java int keyword is used to declare a variable that can hold a 32-bit signed integer.
25. interface: Java interface keyword is used to declare an interface. It can have only abstract methods.
26. long: Java long keyword is used to declare a variable that can hold a 64-bit integer.
27. native: Java native keyword is used to specify that a method is implemented in native code using JNI (Java Native Interface).
28. new: Java new keyword is used to create new objects.
29. null: Java null keyword is used to indicate that a reference does not refer to anything. It removes the garbage value.
30. package: Java package keyword is used to declare a Java package that includes the classes.
31. private: Java private keyword is an access modifier. It is used to indicate that a method or variable may be accessed only in the class in which it is declared.
32. protected: Java protected keyword is an access modifier. It can be accessible within package and outside the package but through inheritance only. It can't be applied on the class.
33. public: Java public keyword is an access modifier. It is used to indicate that an item is accessible anywhere. It has the widest scope among all other modifiers.
34. return: Java return keyword is used to return from a method when its execution is complete.
35. short: Java short keyword is used to declare a variable that can hold a 16-bit integer.
36. static: Java static keyword is used to indicate that a variable or method is a class method. The static keyword in Java is used for memory management mainly.
37. strictfp: Java strictfp is used to restrict the floating-point calculations to ensure portability.
38. super: Java super keyword is a reference variable that is used to refer parent class object. It can be used to invoke immediate parent class method.
39. switch: The Java switch keyword contains a switch statement that executes code based on test value. The switch statement tests the equality of a variable against multiple values.
40. synchronized: Java synchronized keyword is used to specify the critical sections or methods in multithreaded code.
41. this: Java this keyword can be used to refer the current object in a method or constructor.
42. throw: The Java throw keyword is used to explicitly throw an exception. The throw keyword is mainly used to throw custom exception. It is followed by an instance.
43. throws: The Java throws keyword is used to declare an exception. Checked exception can be propagated with throws.
44. transient: Java transient keyword is used in serialization. If you define any data member as transient, it will not be serialized.
45. try: Java try keyword is used to start a block of code that will be tested for exceptions. The try block must be followed by either catch or finally block.
46. void: Java void keyword is used to specify that a method does not have a return value.
47. volatile: Java volatile keyword is used to indicate that a variable may change asynchronously.
48. while: Java while keyword is used to start a while loop. This loop iterates a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.

**Value and reference types**

There are two types of data types in Java: value and reference.

**The value types ( value types )** are stored in a stack for the execution of the program and contain direct value. The values ​​are primitive numeric types, symbolic type and boolean type: **byte** , **int** , **short** , **long** , **float** , **double** , **char** , **b oolean** . Such variables occupy 1, 2, 4 or 8 bytes in the stack. They are released when out of range.

**Reference types (reference types)** store a reference to dynamic memory (the so-called heap) in the program execution stack, where their real value is stored. The reference is a pointer (memory cell address) indicating the actual location of the value in the dynamic memory. The reference is of type and can only be of value of objects of its type, i.e. it is a typed directory. All reference (object) types can have a value of **null** . This is a special service value, which means that there is no value. Reference types allocate dynamic memory when created and are released at some time by the **garbage collector**when she finds that they are no longer in use by the program. Since memory allocation and release is a slow operation, reference types can be said to be slower than value types.

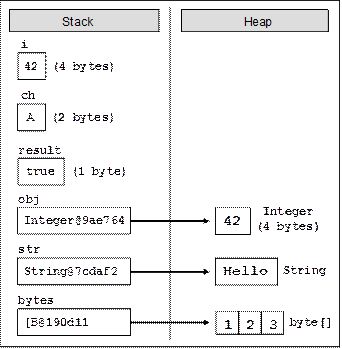
Reference types are all classes, arrays, enumerated types, and interfaces, such as types: **Object** , **String** , **Integer** , **byte []** . Objects, character strings, arrays and interfaces will be introduced in the next chapters of the book. For now, it is enough to know that all non-value types are referenced and stored in dynamic memory.

**Value and reference types and memory**

Let us illustrate with one example how values ​​and reference types are represented in memory. Let the following code be executed:

|  |
| --- |
| **int** i = 42;  **char** ch = 'A' ;  **boolean** result =**true** ;  Object obj = 42;  String str = "Hello" ;  **byte** [] bytes = {1, 2, 3}; |

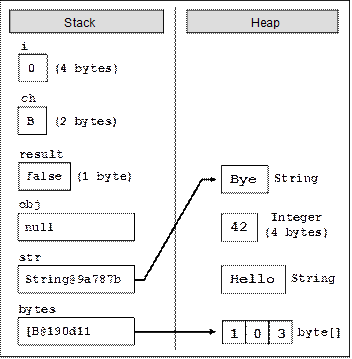
At this point, the variables are stored in memory as follows:

[](https://introprogramming.info/wp-content/uploads/2011/07/clip_image00320.gif)

If we now execute the following code that changes the values ​​of the variables, we will see what happens to the memory when changing the value and reference types:

|  |
| --- |
| i = 0;  ch = 'B' ;  result = **false** ;  obj = **null** ;  str = "Bye" ;  bytes [1] = 0 ; |

After these changes, the variables and their values ​​are stored in memory as follows:

[](https://introprogramming.info/wp-content/uploads/2011/07/clip_image00510.gif)

As you can see from the figure, when a value type changes ( **i = 0** ), its value directly changes to the stack.

When a reference type is changed, its value in the dynamic memory changes directly ( **bytes [1] = 0** ). The reference holding variable remains unchanged ( **B @ 190d11** ). When writing a **null** value in a reference type, the corresponding reference is disconnected from its value and the variable remains worthless ( **obj = null** ).

When assigning a new value to an object (reference type), the new object is allocated to the dynamic value and the old object remains free. The reference is redirected to the new object ( **str = "Bye"** ). Old items, because they are not used, will be cleared at some time by the garbage collector.

Chapter 2

**Priority for operators in Java**

Some operators have priority over others. Higher-priority operators are calculated before lower-priority operators. The operator **()** is used to change the priority of operators and is calculated first, just like in mathematics.

The table shows the priorities of Java operators:

|  |  |
| --- | --- |
| **Priority** | **Operators** |
| the highest | **++, -** (as suffix),**new, (type)** |
|  | **++, -** (as prefix),**+, -** (single argument),**!, ~** |
|  | **\*, /,%** |
|  | **+** (string binding) |
|  | **+, -** |
|  | **<<, >>, >>>** |
|  | **<,>, <=,> =, instanceof** |
|  | **==,! =** |
|  | **&, ^, |** |
|  | **&&** |
|  | **||** |
|  | **?:** |
| the lowest | **=, \* =, / =,% =, + =, - =, << =, >> =, >>> = & =, ^ =, | =** |

#### Arithmetic operators - examples

Here are some examples of arithmetic operators:

|  |
| --- |
| **int** squarePerimeter = 17;  **double** squareSide = squarePerimeter / 4.0;  **double** squareArea = squareSide \* squareSide;  System. *out* .println (squareSide); // 4.25  System. *out* .println (squareArea); // 18.0625    **int** a = 5;  **int** b = 4;  System. *out* .println (a + b);       // 9  System. *out* .println (a + b ++);     // 9  System. *out* .println (a + b);       // 10  System. *out* .println (a + (++ b));   // 11  System. *out* .println (a + b);       // 11  System. *out* .println (14 / a);      // 2  System . *out* . println ( 14 % a );      // 4 |

### Logical operators

Boolean operators accept boolean values ​​and return a boolean result ( **true** or **false** ). The main Boolean operators are AND ( **&&** ), OR ( **||** ), excluding OR ( **^** ), and logical negation ( **!** ).

The following is a table of the logical operators in Java and the operations they perform:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **x** | **y** | **! x** | **x && y** | **x || y** | **x ^ y** |
| **true** | **true** | **false** | **true** | **true** | **false** |
| **true** | **false** | **false** | **false** | **true** | **true** |
| **false** | **true** | **true** | **false** | **true** | **true** |
| **false** | **false** | **true** | **false** | **false** | **false** |

From the table and from the following example, it becomes clear that the logical "AND" returns true only when both variables contain truth. The logical OR returns true when at least one of the operands is true. The logical negation statement changes the value of the argument. For example, if the operand had a value of true and a valid negation operator, the new value would be **false** . The negation statement is put before the argument. The exclusion OR returns a true result when only one of the two operands has a value of **true** . If the two operands have different values, the exclusion OR will return the result true, if they have the same values ​​it will return **false** .

#### Logical operators - an example

Here is an example of using logical operators. The result of the action of the individual logical operators is given as comments:

|  |
| --- |
| **boolean** a =**true** ;  **boolean** b =**false** ;  System. *out* .println (a && b);          // false  System. *out* .println (a || b);          // true  System. *out* .println (! b);              // true  System. *out* .println (b || **true** );       // true  System. *out* .println ((5> 7) ^ (a == b)); // false |

### String join operator

The **+** operator is used to join character strings ( **String** ). What the operator does is simply blind two or more strings and return the result as a new string. If at least one of the arguments in the statement is **String** , and there are other non- **String operands** , then they will be automatically converted to **String** .

#### String join operator - example

Here is an example where we combine several character strings:

|  |
| --- |
| String first = "Star" ;  String second = "Craft" ;  System. *out* .println (first + second); // StarCraft  String output = first + second + "" ;  **int** number = 2;  System. *out* .println (output + number);  // StarCraft 2 |

In the example, we initialize two **String** variables and set values ​​to them. In the third line, we combine the two strings and submit the result of the **println ()** method to print it on the console. In the next line, we join the two strings and add a space at the end. We save the returned result in a variable called **output** . In the last line, we combine the contents of the string **output** with the number 2 (the contents of the variable **number** ) and resubmit the result for printing. The returned result will be automatically converted to a **String** type because the two variables are of a different type.

|  |  |
| --- | --- |
| [clip_image001](https://introprogramming.info/wp-content/uploads/2011/07/clip_image00157.gif) | **String concatenation is a slow operation and should be used with caution. The use of StringBuilder or StringBuffer classes is recommended when iterative (repeated) operations on character strings are required.** |

### Comparison operators

Java comparison operators are used to compare two or more operands. Java supports six comparison operators:

-     larger (>)

-     less (<)

-     greater than or equal to (> =)

-     less than or equal to (<=)

-     equality operator (==)

-     difference (! =)

All comparison operators are two-argument (accept two operands), and the result returned is boolean ( **true** or **false** ). Comparison operators have a lower priority than arithmetic ones, but are higher than the value assignment operators.

#### Comparison operators - an example

The following is an example program that demonstrates the use of comparison operators in Java:

|  |
| --- |
| **public class** RelationalOperatorsDemo {  **public static void** main (String args []) {  **int** x = 10, y = 5;              System. *out* .println ( "x> y:" + (x> y)); // true              System. *out* .println ( "x <y:" + (x <y)); // false              System. *out* .println ( "x> = y:" + (x> = y)); // true              System. *out* .println ( "x <= y:" + (x <= y)); // false              System. *out* .println ( "x == y:" + (x == y)); // false              System. *out* .println ( "x! = y:" + (x! = y)); // true        }  } |

In the example program, we first created the two variables **x** and **y** and assigned them the values ​​10 and 5. In the next line, we print to the console, using the **println ()** method of **System.out** , the result of comparing the two variables **x** and **y** with the operator>. The result returned is **true** because **x** has a greater value than **y** . The next 5 lines print the returned result of using the other 5 operators to compare the **x** and **y** variables .

### Assignment operators

The variable assignment statement is " **=** " (equal). The syntax used to assign values ​​is the following:

|  |
| --- |
| **operand1 = literal or operand2;** |

#### Assignment operators - an example

Here's an example where we use the value assignment operator:

|  |
| --- |
| **int** x = 6;  String helloString = "Hello string." ;  **int** y = x; |

In the example above, we assign a value of 6 to the variable x . In the second line, we assign a text literal to the **helloString** variable , and in the third line we copy the value from the variable **x** to the variable **y** .

#### Cascading assignment

The assignment statement can also be used cascading (to be used more than once in the same expression). In this case, the assignments are made sequentially from right to left. Here is an example:

|  |
| --- |
| **int** x, y, z;  x = y = z = 25; |

On the first line of the example we create three variables, and on the second line we initialize them with value 25 .

|  |  |
| --- | --- |
| [clip_image001 [1]](https://introprogramming.info/wp-content/uploads/2011/07/clip_image001117.gif) | **The Java assignment operator is "**= **", while the comparison operator is "**== **". The exchange of the two operators is a common cause of errors when writing code. Be careful not to confuse the operator with the assignment operator.** |

#### Combined assignment operators

In addition to the Java assignment operator, there are combination assignment operators. They help reduce the amount of code by allowing two operations to be written through one operator. Combined operators have the following syntax:

|  |
| --- |
| **operand1 operator = operand2;** |

The above expression is identical to the following:

|  |
| --- |
| **operand1 = operand1 operand operator2;** |

Here is an example of a combined assignment operator:

|  |
| --- |
| **int** x = 2;  **int** y = 4;    x \* = y; // Same as x = x \* y;  System. *out* .println (x); // 8 |

The most commonly used combination assignment operators are **+ =** (adds the value of **operand2** to **operand1** ), **- =** (subtracts the value of the operand to the right of that of the left). Other constituent assignment operators are **\* =** , **/ =** and **% =** .

The following example gives a good idea of ​​the combined assignment operators and their use:

|  |
| --- |
| **int** x = 6;  **int** y = 4;    System. *out* .println (y \* = 2); // 8  **int** z = y = 3;               // y = 3 and z = 3    System. *out* .println (z);       // 3  System. *out* .println (x | = 1); // 7  System. *out* .println (x + = 3); // 10  System. *out* .println (x / = 2); // 5 |

In the example, we first create the variables **x** and **y** and assign values ​​to them 6 and 4. In the next line, we print to the console **y** , after assigning a new value with the operator **\* =** and literal **2. The** result of the operation is 8.   Then in the example we apply other composite assignment operators and we derive the resulting console result.

### Conditional Operator ?:

Conditional Operator ?: Uses the boolean value of one expression to determine which of the other two expressions to count and return as a result. The operator is working on 3 operands. The symbol " **?** " Is placed between the first and second operands, and " **:** " is placed between the second and third operands. The first operand (or expression) must be Boolean.

The syntax of the operator is the following:

|  |
| --- |
| **operand1? operand2: operand3** |

If **operand1** is **true** , the operator returns result **operand2** . If **operand1** is **false** , the operator returns an **operand3** result .

At run time, the value of the first argument is calculated. If it is **true** , then the second (middle) argument is calculated and returned as a result. However, if the calculated result from the first argument is **false** , then the third (last) argument is calculated and returned as a result.

#### Conditional statement?: - example

Here's an example of using the " **?:** " Operator :

|  |
| --- |
| **int** a = 6;  **int** b = 4;  System. *out* .println (a> b? "a> b" : "b <= a" ); // a> b |

### Other operators

So far we have considered arithmetic operators, logical and bitwise operators, the string concatenation operator, and the conditional operator **?:** . There are several other operators in Java:

-       The access operator " **.** " Is used to access member variables of an object.

-       Square brackets **[]** are used to access array elements.

-       Brackets **()** are used to redefine the priority of expressions and operators.

-       The **type** conversion operator is used to convert a variable from one compatible type to another.

-       The **new** operator is used to create and initialize new objects.

-       The **instanceof** operator is used to check if an object is compatible with a given type.

#### Other operators - examples

Here are some examples of the operators we looked at in this section:

|  |
| --- |
| **int** a = 6;  **int** b = 3;  **int** c = 3;    System. *out* .println (c);          // 3  System. *out* .println ((a + b) / 2); // 4    String s = "Beer" ;  System. *out* .println (s **instanceof** String); // true    **int** d = 0;  System. *out* .println (d);          // 0  System. *out* .println ((a + b) / d); // ArithmeticException |

# **Numbers Class**

Normally, when we work with Numbers, we use primitive data types such as byte, int, long, double, etc.

### Example

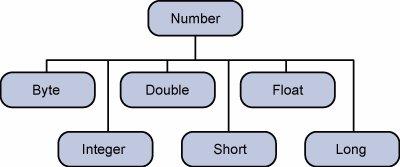
int i = 5000;

float gpa = 13.65;

double mask = 0xaf;

However, in development, we come across situations where we need to use objects instead of primitive data types. In order to achieve this, Java provides **wrapper classes**.

All the wrapper classes (Integer, Long, Byte, Double, Float, Short) are subclasses of the abstract class Number.



The object of the wrapper class contains or wraps its respective primitive data type. Converting primitive data types into object is called **boxing**, and this is taken care by the compiler. Therefore, while using a wrapper class you just need to pass the value of the primitive data type to the constructor of the Wrapper class.

And the Wrapper object will be converted back to a primitive data type, and this process is called unboxing. The **Number** class is part of the java.lang package.

Following is an example of boxing and unboxing −

### Example

public class Test {

public static void main(String args[]) {

Integer x = 5; // boxes int to an Integer object

x = x + 10; // unboxes the Integer to a int

System.out.println(x);

}

}

This will produce the following result −

### Output

15

When x is assigned an integer value, the compiler boxes the integer because x is integer object. Later, x is unboxed so that they can be added as an integer.

## Number Methods

Following is the list of the instance methods that all the subclasses of the Number class implements −

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | xxxValue() |
| 2 | Converts the value of this Number object to the xxx data type and returns it. |
| 3 | compareTo() |
| 4 | Compares this Number object to the argument. |
| 5 | equals() |
| 6 | Determines whether this number object is equal to the argument. |
| 7 | valueOf() |
| 8 | Returns an Integer object holding the value of the specified primitive. |
| 9 | toString() |
| 10 | Returns a String object representing the value of a specified int or Integer. |
| 11 | parseInt() |
| 12 | This method is used to get the primitive data type of a certain String. |
| 13 | abs() |
| 14 | Returns the absolute value of the argument. |
| 15 | ceil() |
| 16 | Returns the smallest integer that is greater than or equal to the argument. Returned as a double. |
| 17 | floor() |
| 18 | Returns the largest integer that is less than or equal to the argument. Returned as a double. |
| 19 | rint() |
| 20 | Returns the integer that is closest in value to the argument. Returned as a double. |
| 21 | round() |
| 22 | Returns the closest long or int, as indicated by the method's return type to the argument. |
| 23 | min() |
| 24 | Returns the smaller of the two arguments. |
| 25 | max() |
| 26 | Returns the larger of the two arguments. |
| 27 | exp() |

# **Character Class**

Normally, when we work with characters, we use primitive data types char.

### Example

char ch = 'a';

// Unicode for uppercase Greek omega character

char uniChar = '\u039A';

// an array of chars

char[] charArray ={ 'a', 'b', 'c', 'd', 'e' };

However in development, we come across situations where we need to use objects instead of primitive data types. In order to achieve this, Java provides wrapper class **Character** for primitive data type char.

The Character class offers a number of useful class (i.e., static) methods for manipulating characters. You can create a Character object with the Character constructor −

Character ch = new Character('a');

The Java compiler will also create a Character object for you under some circumstances. For example, if you pass a primitive char into a method that expects an object, the compiler automatically converts the char to a Character for you. This feature is called autoboxing or unboxing, if the conversion goes the other way.

### Example

// Here following primitive char 'a'

// is boxed into the Character object ch

Character ch = 'a';

// Here primitive 'x' is boxed for method test,

// return is unboxed to char 'c'

char c = test('x');

## Escape Sequences

A character preceded by a backslash (\) is an escape sequence and has a special meaning to the compiler.

The newline character (\n) has been used frequently in this tutorial in System.out.println() statements to advance to the next line after the string is printed.

Following table shows the Java escape sequences −

|  |  |
| --- | --- |
| **Escape Sequence** | **Description** |
| \t | Inserts a tab in the text at this point. |
| \b | Inserts a backspace in the text at this point. |
| \n | Inserts a newline in the text at this point. |
| \r | Inserts a carriage return in the text at this point. |
| \f | Inserts a form feed in the text at this point. |
| \' | Inserts a single quote character in the text at this point. |
| \" | Inserts a double quote character in the text at this point. |
| \\ | Inserts a backslash character in the text at this point. |

When an escape sequence is encountered in a print statement, the compiler interprets it accordingly.

### Example

If you want to put quotes within quotes, you must use the escape sequence, \", on the interior quotes −

public class Test {

public static void main(String args[]) {

System.out.println("She said \"Hello!\" to me.");

}

}

This will produce the following result −

### Output

She said "Hello!" to me.

## Character Methods

Following is the list of the important instance methods that all the subclasses of the Character class implement −

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | isLetter() |
| 2 | Determines whether the specified char value is a letter. |
| 3 | isDigit() |
| 4 | Determines whether the specified char value is a digit. |
| 5 | isWhitespace() |
| 6 | Determines whether the specified char value is white space. |
| 7 | isUpperCase() |
| 8 | Determines whether the specified char value is uppercase. |

For a complete list of methods, please refer to the java.lang.Character API specification.

# **Strings Class**

Strings, which are widely used in Java programming, are a sequence of characters. In Java programming language, strings are treated as objects.

The Java platform provides the String class to create and manipulate strings.

## Creating Strings

The most direct way to create a string is to write −

String greeting = "Hello world!";

Whenever it encounters a string literal in your code, the compiler creates a String object with its value in this case, "Hello world!'.

As with any other object, you can create String objects by using the new keyword and a constructor. The String class has 11 constructors that allow you to provide the initial value of the string using different sources, such as an array of characters.

### Example

public class StringDemo {

public static void main(String args[]) {

char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };

String helloString = new String(helloArray);

System.out.println( helloString );

}

}

This will produce the following result −

### Output

hello.

**Note** − The String class is immutable, so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters, then you should use String Buffer & String Builder Classes.

## String Length

Methods used to obtain information about an object are known as **accessor methods**. One accessor method that you can use with strings is the length() method, which returns the number of characters contained in the string object.

The following program is an example of **length()**, method String class.

### Example

public class StringDemo {

public static void main(String args[]) {

String palindrome = "Dot saw I was Tod";

int len = palindrome.length();

System.out.println( "String Length is : " + len );

}

}

This will produce the following result −

### Output

String Length is : 17

## Concatenating Strings

The String class includes a method for concatenating two strings −

string1.concat(string2);

This returns a new string that is string1 with string2 added to it at the end. You can also use the concat() method with string literals, as in −

"My name is ".concat("Zara");

Strings are more commonly concatenated with the + operator, as in −

"Hello," + " world" + "!"

which results in −

"Hello, world!"

Let us look at the following example −

### Example

public class StringDemo {

public static void main(String args[]) {

String string1 = "saw I was ";

System.out.println("Dot " + string1 + "Tod");

}

}

This will produce the following result −

### Output

Dot saw I was Tod

## Creating Format Strings

You have printf() and format() methods to print output with formatted numbers. The String class has an equivalent class method, format(), that returns a String object rather than a PrintStream object.

Using String's static format() method allows you to create a formatted string that you can reuse, as opposed to a one-time print statement. For example, instead of −

### Example

System.out.printf("The value of the float variable is " +

"%f, while the value of the integer " +

"variable is %d, and the string " +

"is %s", floatVar, intVar, stringVar);

You can write −

String fs;

fs = String.format("The value of the float variable is " +

"%f, while the value of the integer " +

"variable is %d, and the string " +

"is %s", floatVar, intVar, stringVar);

System.out.println(fs);

## String Methods

Here is the list of methods supported by String class −

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | char charAt(int index) |
| 2 | Returns the character at the specified index. |
| 3 | int compareTo(Object o) |
| 4 | Compares this String to another Object. |
| 5 | int compareTo(String anotherString) |
| 6 | Compares two strings lexicographically. |
| 7 | int compareToIgnoreCase(String str) |
| 8 | Compares two strings lexicographically, ignoring case differences. |
| 9 | String concat(String str) |
| 10 | Concatenates the specified string to the end of this string. |
| 11 | boolean contentEquals(StringBuffer sb) |
| 12 | Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer. |
| 13 | static String copyValueOf(char[] data) |
| 14 | Returns a String that represents the character sequence in the array specified. |
| 15 | static String copyValueOf(char[] data, int offset, int count) |
| 16 | Returns a String that represents the character sequence in the array specified. |
| 17 | boolean endsWith(String suffix) |
| 18 | Tests if this string ends with the specified suffix. |
| 19 | boolean equals(Object anObject) |
| 20 | Compares this string to the specified object. |
| 21 | boolean equalsIgnoreCase(String anotherString) |
| 22 | Compares this String to another String, ignoring case considerations. |
| 23 | byte getBytes() |
| 24 | Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array. |
| 25 | byte[] getBytes(String charsetName) |
| 26 | Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array. |
| 27 | void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) |
| 28 | Copies characters from this string into the destination character array. |
| 29 | int hashCode() |
| 30 | Returns a hash code for this string. |
| 31 | int indexOf(int ch) |
| 32 | Returns the index within this string of the first occurrence of the specified character. |
| 33 | int indexOf(int ch, int fromIndex) |
| 34 | Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index. |
| 35 | int indexOf(String str) |
| 36 | Returns the index within this string of the first occurrence of the specified substring. |
| 37 | int indexOf(String str, int fromIndex) |
| 38 | Returns the index within this string of the first occurrence of the specified substring, starting at the specified index. |
| 39 | String intern() |
| 40 | Returns a canonical representation for the string object. |
| 41 | int lastIndexOf(int ch) |
| 42 | Returns the index within this string of the last occurrence of the specified character. |
| 43 | int lastIndexOf(int ch, int fromIndex) |
| 44 | Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index. |
| 45 | int lastIndexOf(String str) |
| 46 | Returns the index within this string of the rightmost occurrence of the specified substring. |

# **Date and Time**

Java provides the **Date** class available in **java.util** package, this class encapsulates the current date and time.

The Date class supports two constructors as shown in the following table.

|  |  |
| --- | --- |
| **Sr.No.** | **Constructor & Description** |
| 1 | **Date( )**  This constructor initializes the object with the current date and time. |
| 2 | **Date(long millisec)**  This constructor accepts an argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970. |

Following are the methods of the date class.

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **boolean after(Date date)**  Returns true if the invoking Date object contains a date that is later than the one specified by date, otherwise, it returns false. |
| 2 | **boolean before(Date date)**  Returns true if the invoking Date object contains a date that is earlier than the one specified by date, otherwise, it returns false. |
| 3 | **Object clone( )**  Duplicates the invoking Date object. |
| 4 | **int compareTo(Date date)**  Compares the value of the invoking object with that of date. Returns 0 if the values are equal. Returns a negative value if the invoking object is earlier than date. Returns a positive value if the invoking object is later than date. |
| 5 | **int compareTo(Object obj)**  Operates identically to compareTo(Date) if obj is of class Date. Otherwise, it throws a ClassCastException. |
| 6 | **boolean equals(Object date)**  Returns true if the invoking Date object contains the same time and date as the one specified by date, otherwise, it returns false. |
| 7 | **long getTime( )**  Returns the number of milliseconds that have elapsed since January 1, 1970. |
| 8 | **int hashCode( )**  Returns a hash code for the invoking object. |
| 9 | **void setTime(long time)**  Sets the time and date as specified by time, which represents an elapsed time in milliseconds from midnight, January 1, 1970. |
| 10 | **String toString( )**  Converts the invoking Date object into a string and returns the result. |

## Getting Current Date and Time

This is a very easy method to get current date and time in Java. You can use a simple Date object with *toString()* method to print the current date and time as follows −

### Example

import java.util.Date;

public class DateDemo {

public static void main(String args[]) {

// Instantiate a Date object

Date date = new Date();

// display time and date using toString()

System.out.println(date.toString());

}

}

This will produce the following result −

### Output

on May 04 09:51:52 CDT 2009

## Date Comparison

Following are the three ways to compare two dates −

* You can use getTime( ) to obtain the number of milliseconds that have elapsed since midnight, January 1, 1970, for both objects and then compare these two values.
* You can use the methods before( ), after( ), and equals( ). Because the 12th of the month comes before the 18th, for example, new Date(99, 2, 12).before(new Date (99, 2, 18)) returns true.
* You can use the compareTo( ) method, which is defined by the Comparable interface and implemented by Date.

## Date Formatting Using SimpleDateFormat

SimpleDateFormat is a concrete class for formatting and parsing dates in a locale-sensitive manner. SimpleDateFormat allows you to start by choosing any user-defined patterns for date-time formatting.

### Example

import java.util.\*;

import java.text.\*;

public class DateDemo {

public static void main(String args[]) {

Date dNow = new Date( );

SimpleDateFormat ft =

new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");

System.out.println("Current Date: " + ft.format(dNow));

}

}

This will produce the following result −

### Output

Current Date: Sun 2004.07.18 at 04:14:09 PM PDT

## Simple DateFormat Format Codes

To specify the time format, use a time pattern string. In this pattern, all ASCII letters are reserved as pattern letters, which are defined as the following −

|  |  |  |
| --- | --- | --- |
| **Character** | **Description** | **Example** |
| G | Era designator | AD |
| y | Year in four digits | 2001 |
| M | Month in year | July or 07 |
| d | Day in month | 10 |
| h | Hour in A.M./P.M. (1~12) | 12 |
| H | Hour in day (0~23) | 22 |
| m | Minute in hour | 30 |
| s | Second in minute | 55 |
| S | Millisecond | 234 |
| E | Day in week | Tuesday |
| D | Day in year | 360 |
| F | Day of week in month | 2 (second Wed. in July) |
| w | Week in year | 40 |
| W | Week in month | 1 |
| a | A.M./P.M. marker | PM |
| k | Hour in day (1~24) | 24 |
| K | Hour in A.M./P.M. (0~11) | 10 |
| z | Time zone | Eastern Standard Time |
| ' | Escape for text | Delimiter |
| " | Single quote | ` |

## Date Formatting Using printf

Date and time formatting can be done very easily using **printf** method. You use a two-letter format, starting with **t** and ending in one of the letters of the table as shown in the following code.

### Example

import java.util.Date;

public class DateDemo {

public static void main(String args[]) {

// Instantiate a Date object

Date date = new Date();

// display time and date

String str = String.format("Current Date/Time : %tc", date );

System.out.printf(str);

}

}

This will produce the following result −

### Output

Current Date/Time : Sat Dec 15 16:37:57 MST 2012

It would be a bit silly if you had to supply the date multiple times to format each part. For that reason, a format string can indicate the index of the argument to be formatted.

The index must immediately follow the % and it must be terminated by a $.

### Example

import java.util.Date;

public class DateDemo {

public static void main(String args[]) {

// Instantiate a Date object

Date date = new Date();

// display time and date

System.out.printf("%1$s %2$tB %2$td, %2$tY", "Due date:", date);

}

}

This will produce the following result −

### Output

Due date: February 09, 2004

Alternatively, you can use the < flag. It indicates that the same argument as in the preceding format specification should be used again.

### Example

import java.util.Date;

public class DateDemo {

public static void main(String args[]) {

// Instantiate a Date object

Date date = new Date();

// display formatted date

System.out.printf("%s %tB %<te, %<tY", "Due date:", date);

}

}

This will produce the following result −

### Output

Due date: February 09, 2004

## Date and Time Conversion Characters

|  |  |  |
| --- | --- | --- |
| **Character** | **Description** | **Example** |
| c | Complete date and time | Mon May 04 09:51:52 CDT 2009 |
| F | ISO 8601 date | 2004-02-09 |
| D | U.S. formatted date (month/day/year) | 02/09/2004 |
| T | 24-hour time | 18:05:19 |
| r | 12-hour time | 06:05:19 pm |
| R | 24-hour time, no seconds | 18:05 |
| Y | Four-digit year (with leading zeroes) | 2004 |
| y | Last two digits of the year (with leading zeroes) | 04 |
| C | First two digits of the year (with leading zeroes) | 20 |
| B | Full month name | February |
| b | Abbreviated month name | Feb |
| m | Two-digit month (with leading zeroes) | 02 |
| d | Two-digit day (with leading zeroes) | 03 |
| e | Two-digit day (without leading zeroes) | 9 |
| A | Full weekday name | Monday |
| a | Abbreviated weekday name | Mon |
| j | Three-digit day of year (with leading zeroes) | 069 |
| H | Two-digit hour (with leading zeroes), between 00 and 23 | 18 |
| k | Two-digit hour (without leading zeroes), between 0 and 23 | 18 |
| I | Two-digit hour (with leading zeroes), between 01 and 12 | 06 |
| l | Two-digit hour (without leading zeroes), between 1 and 12 | 6 |
| M | Two-digit minutes (with leading zeroes) | 05 |
| S | Two-digit seconds (with leading zeroes) | 19 |
| L | Three-digit milliseconds (with leading zeroes) | 047 |
| N | Nine-digit nanoseconds (with leading zeroes) | 047000000 |
| P | Uppercase morning or afternoon marker | PM |
| p | Lowercase morning or afternoon marker | pm |
| z | RFC 822 numeric offset from GMT | -0800 |
| Z | Time zone | PST |
| s | Seconds since 1970-01-01 00:00:00 GMT | 1078884319 |
| Q | Milliseconds since 1970-01-01 00:00:00 GMT | 1078884319047 |

There are other useful classes related to Date and time. For more details, you can refer to Java Standard documentation.

## Parsing Strings into Dates

The SimpleDateFormat class has some additional methods, notably parse( ), which tries to parse a string according to the format stored in the given SimpleDateFormat object.

### Example

import java.util.\*;

import java.text.\*;

public class DateDemo {

public static void main(String args[]) {

SimpleDateFormat ft = new SimpleDateFormat ("yyyy-MM-dd");

String input = args.length == 0 ? "1818-11-11" : args[0];

System.out.print(input + " Parses as ");

Date t;

try {

t = ft.parse(input);

System.out.println(t);

} catch (ParseException e) {

System.out.println("Unparseable using " + ft);

}

}

}

A sample run of the above program would produce the following result −

### Output

1818-11-11 Parses as Wed Nov 11 00:00:00 EST 1818

## Sleeping for a While

You can sleep for any period of time from one millisecond up to the lifetime of your computer. For example, the following program would sleep for 3 seconds −

### Example

import java.util.\*;

public class SleepDemo {

public static void main(String args[]) {

try {

System.out.println(new Date( ) + "\n");

Thread.sleep(5\*60\*10);

System.out.println(new Date( ) + "\n");

} catch (Exception e) {

System.out.println("Got an exception!");

}

}

}

This will produce the following result −

### Output

Sun May 03 18:04:41 GMT 2009

Sun May 03 18:04:51 GMT 2009

## Measuring Elapsed Time

Sometimes, you may need to measure point in time in milliseconds. So let's re-write the above example once again −

### Example

import java.util.\*;

public class DiffDemo {

public static void main(String args[]) {

try {

long start = System.currentTimeMillis( );

System.out.println(new Date( ) + "\n");

Thread.sleep(5\*60\*10);

System.out.println(new Date( ) + "\n");

long end = System.currentTimeMillis( );

long diff = end - start;

System.out.println("Difference is : " + diff);

} catch (Exception e) {

System.out.println("Got an exception!");

}

}

}

This will produce the following result −

### Output

Sun May 03 18:16:51 GMT 2009

Sun May 03 18:16:57 GMT 2009

Difference is : 5993

## GregorianCalendar Class

GregorianCalendar is a concrete implementation of a Calendar class that implements the normal Gregorian calendar with which you are familiar. We did not discuss Calendar class in this tutorial, you can look up standard Java documentation for this.

The **getInstance( )** method of Calendar returns a GregorianCalendar initialized with the current date and time in the default locale and time zone. GregorianCalendar defines two fields: AD and BC. These represent the two eras defined by the Gregorian calendar.

There are also several constructors for GregorianCalendar objects −

|  |  |
| --- | --- |
| **Sr.No.** | **Constructor & Description** |
| 1 | **GregorianCalendar()**  Constructs a default GregorianCalendar using the current time in the default time zone with the default locale. |
| 2 | **GregorianCalendar(int year, int month, int date)**  Constructs a GregorianCalendar with the given date set in the default time zone with the default locale. |
| 3 | **GregorianCalendar(int year, int month, int date, int hour, int minute)**  Constructs a GregorianCalendar with the given date and time set for the default time zone with the default locale. |
| 4 | **GregorianCalendar(int year, int month, int date, int hour, int minute, int second)**  Constructs a GregorianCalendar with the given date and time set for the default time zone with the default locale. |
| 5 | **GregorianCalendar(Locale aLocale)**  Constructs a GregorianCalendar based on the current time in the default time zone with the given locale. |
| 6 | **GregorianCalendar(TimeZone zone)**  Constructs a GregorianCalendar based on the current time in the given time zone with the default locale. |
| 7 | **GregorianCalendar(TimeZone zone, Locale aLocale)**  Constructs a GregorianCalendar based on the current time in the given time zone with the given locale. |

Here is the list of few useful support methods provided by GregorianCalendar class −

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **void add(int field, int amount)**  Adds the specified (signed) amount of time to the given time field, based on the calendar's rules. |
| 2 | **protected void computeFields()**  Converts UTC as milliseconds to time field values. |
| 3 | **protected void computeTime()**  Overrides Calendar Converts time field values to UTC as milliseconds. |
| 4 | **boolean equals(Object obj)**  Compares this GregorianCalendar to an object reference. |
| 5 | **int get(int field)**  Gets the value for a given time field. |
| 6 | **int getActualMaximum(int field)**  Returns the maximum value that this field could have, given the current date. |
| 7 | **int getActualMinimum(int field)**  Returns the minimum value that this field could have, given the current date. |
| 8 | **int getGreatestMinimum(int field)**  Returns highest minimum value for the given field if varies. |
| 9 | **Date getGregorianChange()**  Gets the Gregorian Calendar change date. |
| 10 | **int getLeastMaximum(int field)**  Returns lowest maximum value for the given field if varies. |
| 11 | **int getMaximum(int field)**  Returns maximum value for the given field. |
| 12 | **Date getTime()**  Gets this Calendar's current time. |
| 13 | **long getTimeInMillis()**  Gets this Calendar's current time as a long. |
| 14 | **TimeZone getTimeZone()**  Gets the time zone. |
| 15 | **int getMinimum(int field)**  Returns minimum value for the given field. |
| 16 | **int hashCode()**  Overrides hashCode. |
| 17 | **boolean isLeapYear(int year)**  Determines if the given year is a leap year. |
| 18 | **void roll(int field, boolean up)**  Adds or subtracts (up/down) a single unit of time on the given time field without changing larger fields. |
| 19 | **void set(int field, int value)**  Sets the time field with the given value. |
| 20 | **void set(int year, int month, int date)**  Sets the values for the fields year, month, and date. |
| 21 | **void set(int year, int month, int date, int hour, int minute)**  Sets the values for the fields year, month, date, hour, and minute. |
| 22 | **void set(int year, int month, int date, int hour, int minute, int second)**  Sets the values for the fields year, month, date, hour, minute, and second. |
| 23 | **void setGregorianChange(Date date)**  Sets the GregorianCalendar change date. |
| 24 | **void setTime(Date date)**  Sets this Calendar's current time with the given Date. |
| 25 | **void setTimeInMillis(long millis)**  Sets this Calendar's current time from the given long value. |
| 26 | **void setTimeZone(TimeZone value)**  Sets the time zone with the given time zone value. |
| 27 | **String toString()**  Returns a string representation of this calendar. |

### Example

import java.util.\*;

public class GregorianCalendarDemo {

public static void main(String args[]) {

String months[] = {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep",

"Oct", "Nov", "Dec"};

int year;

// Create a Gregorian calendar initialized

// with the current date and time in the

// default locale and timezone.

GregorianCalendar gcalendar = new GregorianCalendar();

// Display current time and date information.

System.out.print("Date: ");

System.out.print(months[gcalendar.get(Calendar.MONTH)]);

System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");

System.out.println(year = gcalendar.get(Calendar.YEAR));

System.out.print("Time: ");

System.out.print(gcalendar.get(Calendar.HOUR) + ":");

System.out.print(gcalendar.get(Calendar.MINUTE) + ":");

System.out.println(gcalendar.get(Calendar.SECOND));

// Test if the current year is a leap year

if(gcalendar.isLeapYear(year)) {

System.out.println("The current year is a leap year");

}else {

System.out.println("The current year is not a leap year");

}

}

}

This will produce the following result −

### Output

Date: Apr 22 2009

Time: 11:25:27

The current year is not a leap year

For a complete list of constant available in Calendar class, you can refer the standard Java documentation.

# **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.

* if statement
* if-else statement
* if-else-if ladder
* nested if statement

## Java if Statement

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

**Syntax:**

1. **if**(condition){
2. //code to be executed
3. }



**Example:**

1. //Java Program to demonstate the use of if statement.
2. **public** **class** IfExample {
3. **public** **static** **void** main(String[] args) {
4. //defining an 'age' variable
5. **int** age=20;
6. //checking the age
7. **if**(age>18){
8. System.out.print("Age is greater than 18");
9. }
10. }
11. }

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 otherwise else block is executed.

**Syntax:**

1. **if**(condition){
2. //code if condition is true
3. }**else**{
4. //code if condition is false
5. }



**Example:**

1. //A Java Program to demonstrate the use of if-else statement.
2. //It is a program of odd and even number.
3. **public** **class** IfElseExample {
4. **public** **static** **void** main(String[] args) {
5. //defining a variable
6. **int** number=13;
7. //Check if the number is divisible by 2 or not
8. **if**(number%2==0){
9. System.out.println("even number");
10. }**else**{
11. System.out.println("odd number");
12. }
13. }
14. }

Output:

odd number

**Leap Year Example:**

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

1. **public** **class** LeapYearExample {
2. **public** **static** **void** main(String[] args) {
3. **int** year=2020;
4. **if**(((year % 4 ==0) && (year % 100 !=0)) || (year % 400==0)){
5. System.out.println("LEAP YEAR");
6. }
7. **else**{
8. System.out.println("COMMON YEAR");
9. }
10. }
11. }

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:**

1. **public** **class** IfElseTernaryExample {
2. **public** **static** **void** main(String[] args) {
3. **int** number=13;
4. //Using ternary operator
5. String output=(number%2==0)?"even number":"odd number";
6. System.out.println(output);
7. }
8. }

Output:

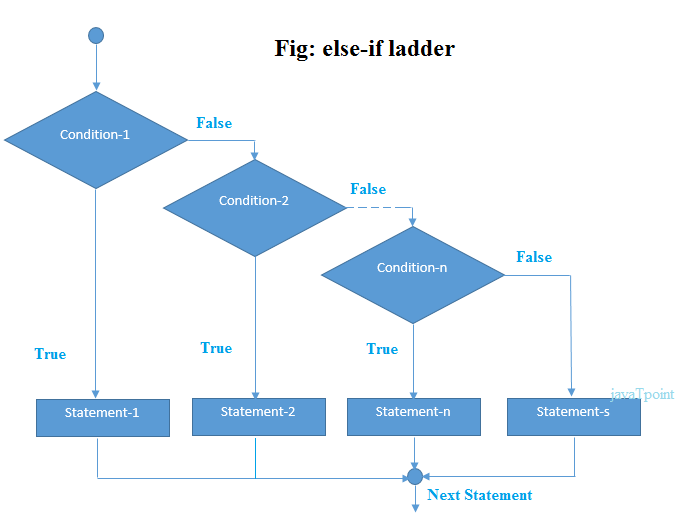
odd number

## Java if-else-if ladder Statement

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

**Syntax:**

1. **if**(condition1){
2. //code to be executed if condition1 is true
3. }**else** **if**(condition2){
4. //code to be executed if condition2 is true
5. }
6. **else** **if**(condition3){
7. //code to be executed if condition3 is true
8. }
9. ...
10. **else**{
11. //code to be executed if all the conditions are false
12. }



**Example:**

1. //Java Program to demonstrate the use of If else-if ladder.
2. //It is a program of grading system for fail, D grade, C grade, B grade, A grade and A+.
3. **public** **class** IfElseIfExample {
4. **public** **static** **void** main(String[] args) {
5. **int** marks=65;
7. **if**(marks<50){
8. System.out.println("fail");
9. }
10. **else** **if**(marks>=50 && marks<60){
11. System.out.println("D grade");
12. }
13. **else** **if**(marks>=60 && marks<70){
14. System.out.println("C grade");
15. }
16. **else** **if**(marks>=70 && marks<80){
17. System.out.println("B grade");
18. }
19. **else** **if**(marks>=80 && marks<90){
20. System.out.println("A grade");
21. }**else** **if**(marks>=90 && marks<100){
22. System.out.println("A+ grade");
23. }**else**{
24. System.out.println("Invalid!");
25. }
26. }
27. }

Output:

C grade

**Program to check POSITIVE, NEGATIVE or ZERO:**

1. **public** **class** PositiveNegativeExample {
2. **public** **static** **void** main(String[] args) {
3. **int** number=-13;
4. **if**(number>0){
5. System.out.println("POSITIVE");
6. }**else** **if**(number<0){
7. System.out.println("NEGATIVE");
8. }**else**{
9. System.out.println("ZERO");
10. }
11. }
12. }

Output:

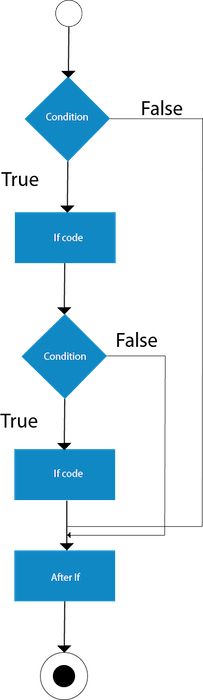
NEGATIVE

## Java Nested if statement

The nested if statement represents the if block within another if block. Here, the inner if block condition executes only when outer if block condition is true.

**Syntax:**

1. **if**(condition){
2. //code to be executed
3. **if**(condition){
4. //code to be executed
5. }
6. }



**Example:**

1. //Java Program to demonstrate the use of Nested If Statement.
2. **public** **class** JavaNestedIfExample {
3. **public** **static** **void** main(String[] args) {
4. //Creating two variables for age and weight
5. **int** age=20;
6. **int** weight=80;
7. //applying condition on age and weight
8. **if**(age>=18){
9. **if**(weight>50){
10. System.out.println("You are eligible to donate blood");
11. }
12. }
13. }}

Output:

You are eligible to donate blood

**Example 2:**

1. //Java Program to demonstrate the use of Nested If Statement.
2. **public** **class** JavaNestedIfExample2 {
3. **public** **static** **void** main(String[] args) {
4. //Creating two variables for age and weight
5. **int** age=25;
6. **int** weight=48;
7. //applying condition on age and weight
8. **if**(age>=18){
9. **if**(weight>50){
10. System.out.println("You are eligible to donate blood");
11. } **else**{
12. System.out.println("You are not eligible to donate blood");
13. }
14. } **else**{
15. System.out.println("Age must be greater than 18");
16. }
17. }  }

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.

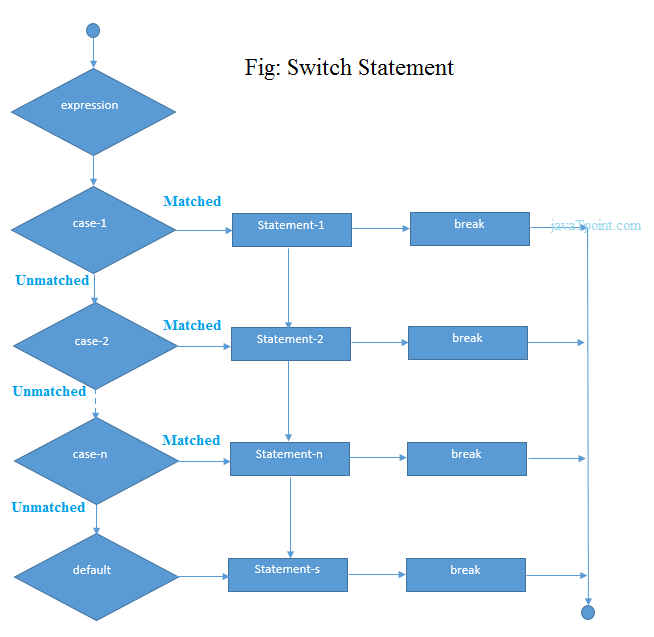
In other words, the switch statement tests the equality of a variable against multiple values.

#### **Points to Remember**

* There can be one or N number of case values for a switch expression.
* The case value must be of switch expression type only. The case value must be literal or constant. It doesn't allow variables.
* The case values must be unique. In case of duplicate value, it renders compile-time error.
* The Java switch expression must be of byte, short, int, long (with its Wrapper type), enums and string.
* Each case statement can have a break statement which is optional. When control reaches to the break statement, it jumps the control after the switch expression. If a break statement is not found, it executes the next case.
* The case value can have a default label which is optional.

**Syntax:**

1. **switch**(expression){
2. **case** value1:
3. //code to be executed;
4. **break**;  //optional
5. **case** value2:
6. //code to be executed;
7. **break**;  //optional
8. ......
10. **default**:
11. code to be executed **if** all cases are not matched;
12. }



**Example:**

1. **public** **class** SwitchExample {
2. **public** **static** **void** main(String[] args) {
3. //Declaring a variable for switch expression
4. **int** number=20;
5. //Switch expression
6. **switch**(number){
7. //Case statements
8. **case** 10: System.out.println("10");
9. **break**;
10. **case** 20: System.out.println("20");
11. **break**;
12. **case** 30: System.out.println("30");
13. **break**;
14. //Default case statement
15. **default**:System.out.println("Not in 10, 20 or 30");
16. }
17. }
18. }

Output:

20

**Finding Month Example:**

1. //Java Program to demonstrate the example of Switch statement
2. //where we are printing month name for the given number
3. **public** **class** SwitchMonthExample {
4. **public** **static** **void** main(String[] args) {
5. //Specifying month number
6. **int** month=7;
7. String monthString="";
8. //Switch statement
9. **switch**(month){
10. //case statements within the switch block
11. **case** 1: monthString="1 - January";
12. **break**;
13. **case** 2: monthString="2 - February";
14. **break**;
15. **case** 3: monthString="3 - March";
16. **break**;
17. **case** 4: monthString="4 - April";
18. **break**;
19. **case** 5: monthString="5 - May";
20. **break**;
21. **case** 6: monthString="6 - June";
22. **break**;
23. **case** 7: monthString="7 - July";
24. **break**;
25. **case** 8: monthString="8 - August";
26. **break**;
27. **case** 9: monthString="9 - September";
28. **break**;
29. **case** 10: monthString="10 - October";
30. **break**;
31. **case** 11: monthString="11 - November";
32. **break**;
33. **case** 12: monthString="12 - December";
34. **break**;
35. **default**:System.out.println("Invalid Month!");
36. }
37. //Printing month of the given number
38. System.out.println(monthString);
39. }
40. }

Output:

7 - July

**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.

1. **public** **class** SwitchVowelExample {
2. **public** **static** **void** main(String[] args) {
3. **char** ch='O';
4. **switch**(ch)
5. {
6. **case** 'a':
7. System.out.println("Vowel");
8. **break**;
9. **case** 'e':
10. System.out.println("Vowel");
11. **break**;
12. **case** 'i':
13. System.out.println("Vowel");
14. **break**;
15. **case** 'o':
16. System.out.println("Vowel");
17. **break**;
18. **case** 'u':
19. System.out.println("Vowel");
20. **break**;
21. **case** 'A':
22. System.out.println("Vowel");
23. **break**;
24. **case** 'E':
25. System.out.println("Vowel");
26. **break**;
27. **case** 'I':
28. System.out.println("Vowel");
29. **break**;
30. **case** 'O':
31. System.out.println("Vowel");
32. **break**;
33. **case** 'U':
34. System.out.println("Vowel");
35. **break**;
36. **default**:
37. System.out.println("Consonant");
38. }
39. }
40. }

Output:

20

## 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:**

1. //Java Switch Example where we are omitting the
2. //break statement
3. **public** **class** SwitchExample2 {
4. **public** **static** **void** main(String[] args) {
5. **int** number=20;
6. //switch expression with int value
7. **switch**(number){
8. //switch cases without break statements
9. **case** 10: System.out.println("10");
10. **case** 20: System.out.println("20");
11. **case** 30: System.out.println("30");
12. **default**:System.out.println("Not in 10, 20 or 30");
13. }
14. }
15. }

Output:

20

30

Not in 10, 20 or 30

## Java Switch Statement with String

Java allows us to use strings in switch expression since Java SE 7. The case statement should be string literal.

**Example:**

1. //Java Program to demonstrate the use of Java Switch
2. //statement with String
3. **public** **class** SwitchStringExample {
4. **public** **static** **void** main(String[] args) {
5. //Declaring String variable
6. String levelString="Expert";
7. **int** level=0;
8. //Using String in Switch expression
9. **switch**(levelString){
10. //Using String Literal in Switch case
11. **case** "Beginner": level=1;
12. **break**;
13. **case** "Intermediate": level=2;
14. **break**;
15. **case** "Expert": level=3;
16. **break**;
17. **default**: level=0;
18. **break**;
19. }
20. System.out.println("Your Level is: "+level);
21. }
22. }

Output:

Your Level is: 3

## Java Nested Switch Statement

We can use switch statement inside other switch statement in Java. It is known as nested switch statement.

**Example:**

1. //Java Program to demonstrate the use of Java Nested Switch
2. **public** **class** NestedSwitchExample {
3. **public** **static** **void** main(String args[])
4. {
5. //C - CSE, E - ECE, M - Mechanical
6. **char** branch = 'C';
7. **int** collegeYear = 4;
8. **switch**( collegeYear )
9. {
10. **case** 1:
11. System.out.println("English, Maths, Science");
12. **break**;
13. **case** 2:
14. **switch**( branch )
15. {
16. **case** 'C':
17. System.out.println("Operating System, Java, Data Structure");
18. **break**;
19. **case** 'E':
20. System.out.println("Micro processors, Logic switching theory");
21. **break**;
22. **case** 'M':
23. System.out.println("Drawing, Manufacturing Machines");
24. **break**;
25. }
26. **break**;
27. **case** 3:
28. **switch**( branch )
29. {
30. **case** 'C':
31. System.out.println("Computer Organization, MultiMedia");
32. **break**;
33. **case** 'E':
34. System.out.println("Fundamentals of Logic Design, Microelectronics");
35. **break**;
36. **case** 'M':
37. System.out.println("Internal Combustion Engines, Mechanical Vibration");
38. **break**;
39. }
40. **break**;
41. **case** 4:
42. **switch**( branch )
43. {
44. **case** 'C':
45. System.out.println("Data Communication and Networks, MultiMedia");
46. **break**;
47. **case** 'E':
48. System.out.println("Embedded System, Image Processing");
49. **break**;
50. **case** 'M':
51. System.out.println("Production Technology, Thermal Engineering");
52. **break**;
53. }
54. **break**;
55. }
56. }
57. }

Output:

Data Communication and Networks, MultiMedia

## Java Enum in Switch Statement

Java allows us to use enum in switch statement.

**Example:**

1. //Java Program to demonstrate the use of Enum
2. //in switch statement
3. **public** **class** JavaSwitchEnumExample {
4. **public** **enum** Day {  Sun, Mon, Tue, Wed, Thu, Fri, Sat  }
5. **public** **static** **void** main(String args[])
6. {
7. Day[] DayNow = Day.values();
8. **for** (Day Now : DayNow)
9. {
10. **switch** (Now)
11. {
12. **case** Sun:
13. System.out.println("Sunday");
14. **break**;
15. **case** Mon:
16. System.out.println("Monday");
17. **break**;
18. **case** Tue:
19. System.out.println("Tuesday");
20. **break**;
21. **case** Wed:
22. System.out.println("Wednesday");
23. **break**;
24. **case** Thu:
25. System.out.println("Thursday");
26. **break**;
27. **case** Fri:
28. System.out.println("Friday");
29. **break**;
30. **case** Sat:
31. System.out.println("Saturday");
32. **break**;
33. }
34. }
35. }
36. }

Output:

Sunday

Monday

Twesday

Wednesday

Thursday

Friday

Saturday

## Java Wrapper in Switch Statement

Java allows us to use four wrapper classes: Byte, Short, Integer and Long in switch statement.

**Example:**

1. //Java Program to demonstrate the use of Wrapper class
2. //in switch statement
3. **public** **class** WrapperInSwitchCaseExample {
4. **public** **static** **void** main(String args[])
5. {
6. Integer age = 18;
7. **switch** (age)
8. {
9. **case** (16):
10. System.out.println("You are under 18.");
11. **break**;
12. **case** (18):
13. System.out.println("You are eligible for vote.");
14. **break**;
15. **case** (65):
16. System.out.println("You are senior citizen.");
17. **break**;
18. **default**:
19. System.out.println("Please give the valid age.");
20. **break**;
21. }
22. }
23. }

Output:

You are eligible for vote.

# **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.

* for loop
* while loop
* 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 | for(init;condition;incr/decr){  // code to be executed  } | while(condition){  //code to be executed  } | do{  //code to be executed  }while(condition); |
| Example | //for loop  for(int i=1;i<=10;i++){  System.out.println(i);  } | //while loop  int i=1;  while(i<=10){  System.out.println(i);  i++;  } | //do-while loop  int i=1;  do{  System.out.println(i);  i++;  }while(i<=10); |
| Syntax for infinitive loop | for(;;){  //code to be executed  } | while(true){  //code to be executed  } | do{  //code to be executed  }while(true); |

# **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

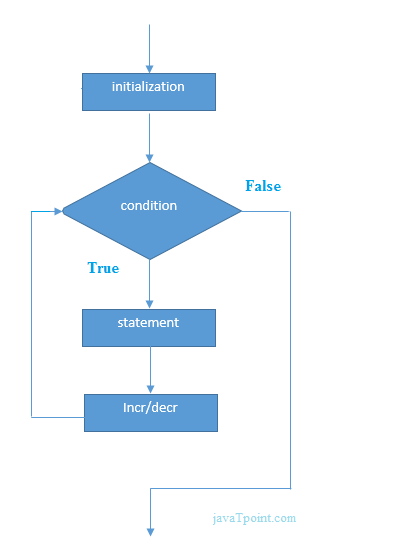
A simple for loop is the same as C/C++. We can initialize the variable, check condition and increment/decrement value. It consists of four parts:

1. **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.
2. **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.
3. **Statement**: The statement of the loop is executed each time until the second condition is false.
4. **Increment/Decrement**: It increments or decrements the variable value. It is an optional condition.

**Syntax:**

1. **for**(initialization;condition;incr/decr){
2. //statement or code to be executed
3. }

**Flowchart:**



**Example:**

1. //Java Program to demonstrate the example of for loop
2. //which prints table of 1
3. **public** **class** ForExample {
4. **public** **static** **void** main(String[] args) {
5. //Code of Java for loop
6. **for**(**int** i=1;i<=10;i++){
7. System.out.println(i);
8. }
9. }
10. }

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:**

1. **public** **class** NestedForExample {
2. **public** **static** **void** main(String[] args) {
3. //loop of i
4. **for**(**int** i=1;i<=3;i++){
5. //loop of j
6. **for**(**int** j=1;j<=3;j++){
7. System.out.println(i+" "+j);
8. }//end of i
9. }//end of j
10. }
11. }

Output:

1 1

1 2

1 3

2 1

2 2

2 3

3 1

3 2

3 3

**Pyramid Example 1:**

1. **public** **class** PyramidExample {
2. **public** **static** **void** main(String[] args) {
3. **for**(**int** i=1;i<=5;i++){
4. **for**(**int** j=1;j<=i;j++){
5. System.out.print("\* ");
6. }
7. System.out.println();//new line
8. }
9. }
10. }

Output:

\*

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

**Pyramid Example 2:**

1. **public** **class** PyramidExample2 {
2. **public** **static** **void** main(String[] args) {
3. **int** term=6;
4. **for**(**int** i=1;i<=term;i++){
5. **for**(**int** j=term;j>=i;j--){
6. System.out.print("\* ");
7. }
8. System.out.println();//new line
9. }
10. }
11. }

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:**

1. **for**(Type var:array){
2. //code to be executed
3. }

**Example:**

1. //Java For-each loop example which prints the
2. //elements of the array
3. **public** **class** ForEachExample {
4. **public** **static** **void** main(String[] args) {
5. //Declaring an array
6. **int** arr[]={12,23,44,56,78};
7. //Printing array using for-each loop
8. **for**(**int** i:arr){
9. System.out.println(i);
10. }
11. }
12. }

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:**

1. labelname:
2. **for**(initialization;condition;incr/decr){
3. //code to be executed
4. }

**Example:**

1. //A Java program to demonstrate the use of labeled for loop
2. **public** **class** LabeledForExample {
3. **public** **static** **void** main(String[] args) {
4. //Using Label for outer and for loop
5. aa:
6. **for**(**int** i=1;i<=3;i++){
7. bb:
8. **for**(**int** j=1;j<=3;j++){
9. **if**(i==2&&j==2){
10. **break** aa;
11. }
12. System.out.println(i+" "+j);
13. }
14. }
15. }
16. }

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.

1. **public** **class** LabeledForExample2 {
2. **public** **static** **void** main(String[] args) {
3. aa:
4. **for**(**int** i=1;i<=3;i++){
5. bb:
6. **for**(**int** j=1;j<=3;j++){
7. **if**(i==2&&j==2){
8. **break** bb;
9. }
10. System.out.println(i+" "+j);
11. }
12. }
13. }
14. }

Output:

1 1

1 2

1 3

2 1

3 1

3 2

3 3

## Java Infinitive For Loop

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

**Syntax:**

1. **for**(;;){
2. //code to be executed
3. }

**Example:**

1. //Java program to demonstrate the use of infinite for loop
2. //which prints an statement
3. **public** **class** ForExample {
4. **public** **static** **void** main(String[] args) {
5. //Using no condition in for loop
6. **for**(;;){
7. System.out.println("infinitive loop");
8. }
9. }
10. }

Output:

infinitive loop

infinitive loop

infinitive loop

infinitive loop

infinitive loop

ctrl+c

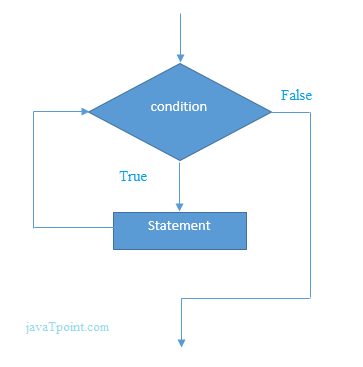
Now, you need to press ctrl+c to exit from the program.

# **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:**

1. **while**(condition){
2. //code to be executed
3. }



**Example:**

1. **public** **class** WhileExample {
2. **public** **static** **void** main(String[] args) {
3. **int** i=1;
4. **while**(i<=10){
5. System.out.println(i);
6. i++;
7. }
8. }
9. }

Output:

1

2

3

4

5

6

7

8

9

10

## Java Infinitive While Loop

If you pass **true** in the while loop, it will be infinitive while loop.

**Syntax:**

1. **while**(**true**){
2. //code to be executed
3. }

**Example:**

1. **public** **class** WhileExample2 {
2. **public** **static** **void** main(String[] args) {
3. **while**(**true**){
4. System.out.println("infinitive while loop");
5. }
6. }
7. }

Output:

infinitive while loop

infinitive while loop

infinitive while loop

infinitive while loop

infinitive while loop

ctrl+c

Now, you need to press ctrl+c to exit from the program.

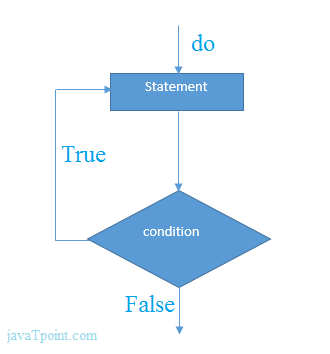
# **Java do-while Loop**

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:**

1. **do**{
2. //code to be executed
3. }**while**(condition);



**Example:**

1. **public** **class** DoWhileExample {
2. **public** **static** **void** main(String[] args) {
3. **int** i=1;
4. **do**{
5. System.out.println(i);
6. i++;
7. }**while**(i<=10);
8. }
9. }

Output:

1

2

3

4

5

6

7

8

9

10

## Java Infinitive do-while Loop

If you pass **true** in the do-while loop, it will be infinitive do-while loop.

**Syntax:**

1. **do**{
2. //code to be executed
3. }**while**(**true**);

**Example:**

1. **public** **class** DoWhileExample2 {
2. **public** **static** **void** main(String[] args) {
3. **do**{
4. System.out.println("infinitive do while loop");
5. }**while**(**true**);
6. }
7. }

Output:

infinitive do while loop

infinitive do while loop

infinitive do while loop

ctrl+c

Now, you need to press ctrl+c to exit from the program.

# **Java Break Statement**

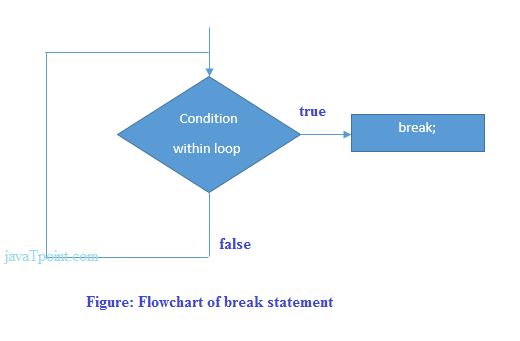
When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

The Java break 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:**

1. jump-statement;
2. **break**;



## Java Break Statement with Loop

**Example:**

1. //Java Program to demonstrate the use of break statement
2. //inside the for loop.
3. **public** **class** BreakExample {
4. **public** **static** **void** main(String[] args) {
5. //using for loop
6. **for**(**int** i=1;i<=10;i++){
7. **if**(i==5){
8. //breaking the loop
9. **break**;
10. }
11. System.out.println(i);
12. }
13. }
14. }

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:**

1. //Java Program to illustrate the use of break statement
2. //inside an inner loop
3. **public** **class** BreakExample2 {
4. **public** **static** **void** main(String[] args) {
5. //outer loop
6. **for**(**int** i=1;i<=3;i++){
7. //inner loop
8. **for**(**int** j=1;j<=3;j++){
9. **if**(i==2&&j==2){
10. //using break statement inside the inner loop
11. **break**;
12. }
13. System.out.println(i+" "+j);
14. }
15. }
16. }
17. }

Output:

1 1

1 2

1 3

2 1

3 1

3 2

3 3

## Java Break Statement with Labeled For Loop

We can use break statement with a label. This feature is introduced since JDK 1.5. So, we can break any loop in Java now whether it is outer loop or inner.

**Example:**

1. //Java Program to illustrate the use of continue statement
2. //with label inside an inner loop to break outer loop
3. **public** **class** BreakExample3 {
4. **public** **static** **void** main(String[] args) {
5. aa:
6. **for**(**int** i=1;i<=3;i++){
7. bb:
8. **for**(**int** j=1;j<=3;j++){
9. **if**(i==2&&j==2){
10. //using break statement with label
11. **break** aa;
12. }
13. System.out.println(i+" "+j);
14. }
15. }
16. }
17. }

Output:

1 1

1 2

1 3

2 1

## Java Break Statement in while loop

**Example:**

1. //Java Program to demonstrate the use of break statement
2. //inside the while loop.
3. **public** **class** BreakWhileExample {
4. **public** **static** **void** main(String[] args) {
5. //while loop
6. **int** i=1;
7. **while**(i<=10){
8. **if**(i==5){
9. //using break statement
10. i++;
11. **break**;//it will break the loop
12. }
13. System.out.println(i);
14. i++;
15. }
16. }
17. }

Output:

1

2

3

4

## Java Break Statement in do-while loop

**Example:**

1. //Java Program to demonstrate the use of break statement
2. //inside the Java do-while loop.
3. **public** **class** BreakDoWhileExample {
4. **public** **static** **void** main(String[] args) {
5. //declaring variable
6. **int** i=1;
7. //do-while loop
8. **do**{
9. **if**(i==5){
10. //using break statement
11. i++;
12. **break**;//it will break the loop
13. }
14. System.out.println(i);
15. i++;
16. }**while**(i<=10);
17. }
18. }

Output:

1

2

3

4

# **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:**

1. jump-statement;
2. **continue**;

## Java Continue Statement Example

**Example:**

1. //Java Program to demonstrate the use of continue statement
2. //inside the for loop.
3. **public** **class** ContinueExample {
4. **public** **static** **void** main(String[] args) {
5. //for loop
6. **for**(**int** i=1;i<=10;i++){
7. **if**(i==5){
8. //using continue statement
9. **continue**;//it will skip the rest statement
10. }
11. System.out.println(i);
12. }
13. }
14. }

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 Continue Statement with Inner Loop

It continues inner loop only if you use the continue statement inside the inner loop.

**Example:**

1. //Java Program to illustrate the use of continue statement
2. //inside an inner loop
3. **public** **class** ContinueExample2 {
4. **public** **static** **void** main(String[] args) {
5. //outer loop
6. **for**(**int** i=1;i<=3;i++){
7. //inner loop
8. **for**(**int** j=1;j<=3;j++){
9. **if**(i==2&&j==2){
10. //using continue statement inside inner loop
11. **continue**;
12. }
13. System.out.println(i+" "+j);
14. }
15. }
16. }
17. }

Output:

1 1

1 2

1 3

2 1

2 3

3 1

3 2

3 3

## Java Continue Statement with Labeled For Loop

We can use continute statement with a label. This feature is introduced since JDK 1.5. So, we can continue any loop in Java now whether it is outer loop or inner.

**Example:**

1. //Java Program to illustrate the use of continue statement
2. //with label inside an inner loop to continue outer loop
3. **public** **class** ContinueExample3 {
4. **public** **static** **void** main(String[] args) {
5. aa:
6. **for**(**int** i=1;i<=3;i++){
7. bb:
8. **for**(**int** j=1;j<=3;j++){
9. **if**(i==2&&j==2){
10. //using continue statement with label
11. **continue** aa;
12. }
13. System.out.println(i+" "+j);
14. }
15. }
16. }
17. }

Output:

1 1

1 2

1 3

2 1

3 1

3 2

3 3

## Java Continue Statement in while loop

**Example:**

1. //Java Program to demonstrate the use of continue statement
2. //inside the while loop.
3. **public** **class** ContinueWhileExample {
4. **public** **static** **void** main(String[] args) {
5. //while loop
6. **int** i=1;
7. **while**(i<=10){
8. **if**(i==5){
9. //using continue statement
10. i++;
11. **continue**;//it will skip the rest statement
12. }
13. System.out.println(i);
14. i++;
15. }
16. }
17. }

Output:

1

2

3

4

6

7

8

9

10

## Java Continue Statement in do-while loop

**Example:**

1. //Java Program to demonstrate the use of continue statement
2. //inside the Java do-while loop.
3. **public** **class** ContinueDoWhileExample {
4. **public** **static** **void** main(String[] args) {
5. //declaring variable
6. **int** i=1;
7. //do-while loop
8. **do**{
9. **if**(i==5){
10. //using continue statement
11. i++;
12. **continue**;//it will skip the rest statement
13. }
14. System.out.println(i);
15. i++;
16. }**while**(i<=10);
17. }
18. }

Output:

1

2

3

4

6

7

8

9

10

# **Java OOPs Concepts**

* Object-Oriented Programming
* Advantage of OOPs over Procedure-oriented programming language
* Difference between Object-oriented and Object-based programming language.

In this page, we will learn about the basics of OOPs. Object-Oriented Programming is a paradigm that provides many concepts, such as **inheritance**, **data binding**, **polymorphism**, etc.

**Simula** is considered the first object-oriented programming language. The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language.

**Smalltalk** is considered the first truly object-oriented programming language.

The popular object-oriented languages are Java, C#, PHP, Python, C++, etc.

The main aim of object-oriented programming is to implement real-world entities, for example, object, classes, abstraction, inheritance, polymorphism, etc.

## 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

Apart from these concepts, there are some other terms which are used in Object-Oriented design:

* Coupling
* Cohesion
* Association
* Aggregation
* Composition



## 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.



### 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.

### Coupling

Coupling refers to the knowledge or information or dependency of another class. It arises when classes are aware of each other. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can use interfaces for the weaker coupling because there is no concrete implementation.

### Cohesion

Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.

### Association

Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:

* One to One
* One to Many
* Many to One, and
* Many to Many

Let's understand the relationship with real-time examples. For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).

Association can be undirectional or bidirectional.

### Aggregation

Aggregation is a way to achieve Association. Aggregation represents the relationship where one object contains other objects as a part of its state. It represents the weak relationship between objects. It is also termed as a has-a relationship in Java. Like, inheritance represents the is-a relationship. It is another way to reuse objects.

### Composition

The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state. There is a strong relationship between the containing object and the dependent object. It is the state where containing objects do not have an independent existence. If you delete the parent object, all the child objects will be deleted automatically.

## Advantage of OOPs over Procedure-oriented programming language

1) OOPs makes development and maintenance easier, whereas, in a procedure-oriented programming language, it is not easy to manage if code grows as project size increases.

2) OOPs provides data hiding, whereas, in a procedure-oriented programming language, global data can be accessed from anywhere.

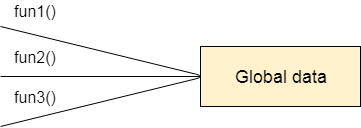


Figure: Data Representation in Procedure-Oriented Programming

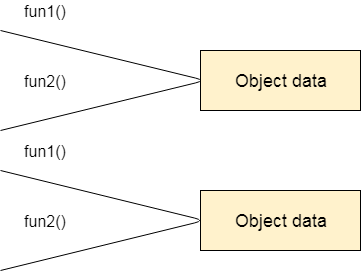


Figure: Data Representation in Object-Oriented Programming

3) OOPs provides the ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

## What is the difference between an object-oriented programming language and object-based programming language?

Object-based programming language follows all the features of OOPs except Inheritance. JavaScript and VBScript are examples of object-based programming languages.

Do You Know?

* Can we overload the main method?
* A Java Constructor returns a value but, what?
* Can we create a program without main method?
* What are the six ways to use this keyword?
* Why is multiple inheritance not supported in Java?
* Why use aggregation?
* Can we override the static method?
* What is the covariant return type?
* What are the three usages of Java super keyword?
* Why use instance initializer block?
* What is the usage of a blank final variable?
* What is a marker or tagged interface?
* What is runtime polymorphism or dynamic method dispatch?
* What is the difference between static and dynamic binding?
* How downcasting is possible in Java?
* What is the purpose of a private constructor?
* What is object cloning?

What will we learn in OOPs Concepts?

* Advantage of OOPs
* Naming Convention
* Object and class
* Method overloading
* Constructor
* static keyword
* this keyword with six usage
* Inheritance
* Aggregation
* Method Overriding
* Covariant Return Type
* super keyword
* Instance Initializer block
* final keyword
* Abstract class
* Interface
* Runtime Polymorphism
* Static and Dynamic Binding
* Downcasting with instanceof operator
* Package
* Access Modifiers
* Encapsulation
* Object Cloning

# **Java Naming conventions**

Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.

But, it is not forced to follow. So, it is known as convention not rule. These conventions are suggested by several Java communities such as Sun Microsystems and Netscape.

All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention. If you fail to follow these conventions, it may generate confusion or erroneous code.

## Advantage of naming conventions in java

By using standard Java naming conventions, you make your code easier to read for yourself and other programmers. Readability of Java program is very important. It indicates that less time is spent to figure out what the code does.

The following are the key rules that must be followed by every identifier:

* The name must not contain any white spaces.
* The name should not start with special characters like & (ampersand), $ (dollar), \_ (underscore).

Let's see some other rules that should be followed by identifiers.

### Class

* It should start with the uppercase letter.
* It should be a noun such as Color, Button, System, Thread, etc.
* Use appropriate words, instead of acronyms.
* **Example: -**

1. **public** **class** Employee
2. {
3. //code snippet
4. }

### Interface

* It should start with the uppercase letter.
* It should be an adjective such as Runnable, Remote, ActionListener.
* Use appropriate words, instead of acronyms.
* **Example: -**

1. **interface** Printable
2. {
3. //code snippet
4. }

### Method

* It should start with lowercase letter.
* It should be a verb such as main(), print(), println().
* If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed().
* **Example:-**

1. **class** Employee
2. {
3. //method
4. **void** draw()
5. {
6. //code snippet
7. }
8. }

### Variable

* It should start with a lowercase letter such as id, name.
* It should not start with the special characters like & (ampersand), $ (dollar), \_ (underscore).
* If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName.
* Avoid using one-character variables such as x, y, z.
* **Example :-**

1. **class** Employee
2. {
3. //variable
4. **int** id;
5. //code snippet
6. }

### Package

* It should be a lowercase letter such as java, lang.
* If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.
* **Example :-**

1. **package** com.javatpoint; //package
2. **class** Employee
3. {
4. //code snippet
5. }

### Constant

* It should be in uppercase letters such as RED, YELLOW.
* If the name contains multiple words, it should be separated by an underscore(\_) such as MAX\_PRIORITY.
* It may contain digits but not as the first letter.
* **Example :-**

1. **class** Employee
2. {
3. //constant
4. **static** **final** **int** MIN\_AGE = 18;
5. //code snippet
6. }

## CamelCase in java naming conventions

Java follows camel-case syntax for naming the class, interface, method, and variable.

If the name is combined with two words, the second word will start with uppercase letter always such as actionPerformed(), firstName, ActionEvent, ActionListener, etc.

# **Objects and Classes in Java**

* Object in Java
* Class in Java
* Instance Variable in Java
* Method in Java
* Example of Object and class that maintains the records of student
* Anonymous Object

In this page, we will learn about Java objects and classes. In object-oriented programming technique, we design a program using objects and classes.

An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

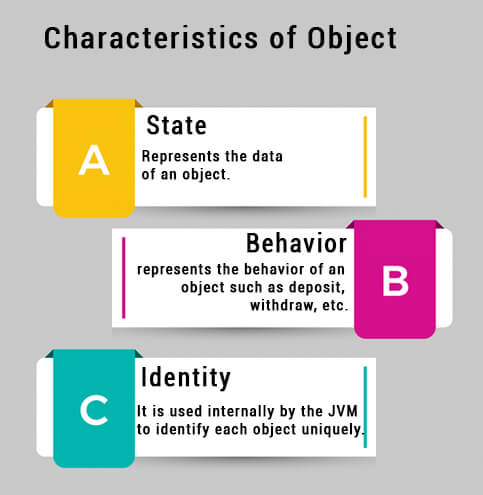
### What is an object in Java



An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

* **State:** represents the data (value) of an object.
* **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
* **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.



For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

**An object is an instance of a class.** A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

**Object Definitions:**

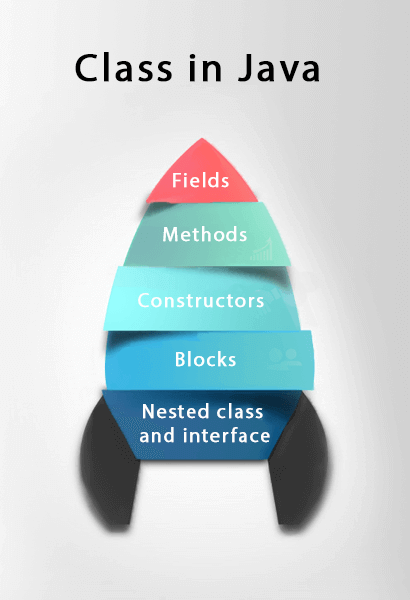
* An object is a real-world entity.
* An object is a runtime entity.
* The object is an entity which has state and behavior.
* The object is an instance of a class.

## What is a class in Java

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

* **Fields**
* **Methods**
* **Constructors**
* **Blocks**
* **Nested class and interface**



### Syntax to declare a class:

1. **class** <class\_name>{
2. field;
3. method;
4. }

### Instance variable in Java

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

### Method in Java

In Java, a method is like a function which is used to expose the behavior of an object.

#### **Advantage of Method**

* Code Reusability
* Code Optimization

### new keyword in Java

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

### Object and Class Example: main within the class

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

*File: Student.java*

1. //Java Program to illustrate how to define a class and fields
2. //Defining a Student class.
3. **class** Student{
4. //defining fields
5. **int** id;//field or data member or instance variable
6. String name;
7. //creating main method inside the Student class
8. **public** **static** **void** main(String args[]){
9. //Creating an object or instance
10. Student s1=**new** Student();//creating an object of Student
11. //Printing values of the object
12. System.out.println(s1.id);//accessing member through reference variable
13. System.out.println(s1.name);
14. }
15. }

Output:

0

null

### Object and Class Example: main outside the class

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different Java files or single Java file. If you define multiple classes in a single Java source file, it is a good idea to save the file name with the class name which has main() method.

*File: TestStudent1.java*

1. //Java Program to demonstrate having the main method in
2. //another class
3. //Creating Student class.
4. **class** Student{
5. **int** id;
6. String name;
7. }
8. //Creating another class TestStudent1 which contains the main method
9. **class** TestStudent1{
10. **public** **static** **void** main(String args[]){
11. Student s1=**new** Student();
12. System.out.println(s1.id);
13. System.out.println(s1.name);
14. }
15. }

Output:

0

null

## 3 Ways to initialize object

There are 3 ways to initialize object in Java.

1. By reference variable
2. By method
3. By constructor

### 1) Object and Class Example: Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

*File: TestStudent2.java*

1. **class** Student{
2. **int** id;
3. String name;
4. }
5. **class** TestStudent2{
6. **public** **static** **void** main(String args[]){
7. Student s1=**new** Student();
8. s1.id=101;
9. s1.name="Sonoo";
10. System.out.println(s1.id+" "+s1.name);//printing members with a white space
11. }
12. }

Output:

101 Sonoo

We can also create multiple objects and store information in it through reference variable.

*File: TestStudent3.java*

1. **class** Student{
2. **int** id;
3. String name;
4. }
5. **class** TestStudent3{
6. **public** **static** **void** main(String args[]){
7. //Creating objects
8. Student s1=**new** Student();
9. Student s2=**new** Student();
10. //Initializing objects
11. s1.id=101;
12. s1.name="Sonoo";
13. s2.id=102;
14. s2.name="Amit";
15. //Printing data
16. System.out.println(s1.id+" "+s1.name);
17. System.out.println(s2.id+" "+s2.name);
18. }
19. }

Output:

101 Sonoo

102 Amit

### 2) Object and Class Example: Initialization through method

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

*File: TestStudent4.java*

1. **class** Student{
2. **int** rollno;
3. String name;
4. **void** insertRecord(**int** r, String n){
5. rollno=r;
6. name=n;
7. }
8. **void** displayInformation(){System.out.println(rollno+" "+name);}
9. }
10. **class** TestStudent4{
11. **public** **static** **void** main(String args[]){
12. Student s1=**new** Student();
13. Student s2=**new** Student();
14. s1.insertRecord(111,"Karan");
15. s2.insertRecord(222,"Aryan");
16. s1.displayInformation();
17. s2.displayInformation();
18. }
19. }

Output:

111 Karan

222 Aryan



As you can see in the above figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

### 3) Object and Class Example: Initialization through a constructor

We will learn about constructors in Java later.

### Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.

*File: TestEmployee.java*

1. **class** Employee{
2. **int** id;
3. String name;
4. **float** salary;
5. **void** insert(**int** i, String n, **float** s) {
6. id=i;
7. name=n;
8. salary=s;
9. }
10. **void** display(){System.out.println(id+" "+name+" "+salary);}
11. }
12. **public** **class** TestEmployee {
13. **public** **static** **void** main(String[] args) {
14. Employee e1=**new** Employee();
15. Employee e2=**new** Employee();
16. Employee e3=**new** Employee();
17. e1.insert(101,"ajeet",45000);
18. e2.insert(102,"irfan",25000);
19. e3.insert(103,"nakul",55000);
20. e1.display();
21. e2.display();
22. e3.display();
23. }
24. }

Output:

101 ajeet 45000.0

102 irfan 25000.0

103 nakul 55000.0

### Object and Class Example: Rectangle

There is given another example that maintains the records of Rectangle class.

*File: TestRectangle1.java*

1. **class** Rectangle{
2. **int** length;
3. **int** width;
4. **void** insert(**int** l, **int** w){
5. length=l;
6. width=w;
7. }
8. **void** calculateArea(){System.out.println(length\*width);}
9. }
10. **class** TestRectangle1{
11. **public** **static** **void** main(String args[]){
12. Rectangle r1=**new** Rectangle();
13. Rectangle r2=**new** Rectangle();
14. r1.insert(11,5);
15. r2.insert(3,15);
16. r1.calculateArea();
17. r2.calculateArea();
18. }
19. }

Output:

55

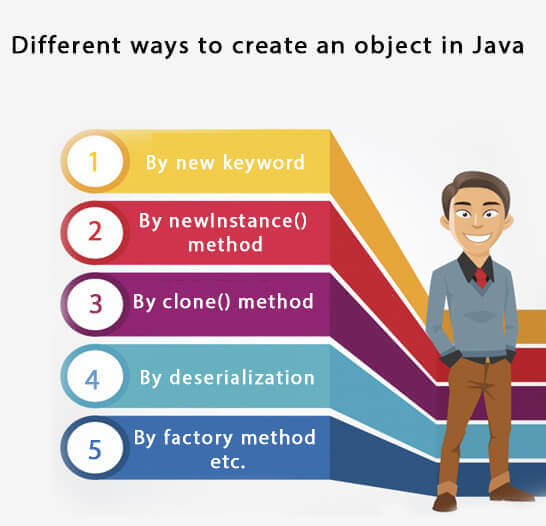
45

## What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

* By new keyword
* By newInstance() method
* By clone() method
* By deserialization
* By factory method etc.

We will learn these ways to create object later.



## Anonymous object

Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only.

If you have to use an object only once, an anonymous object is a good approach. For example:

1. **new** Calculation();//anonymous object

Calling method through a reference:

1. Calculation c=**new** Calculation();
2. c.fact(5);

Calling method through an anonymous object

1. **new** Calculation().fact(5);

Let's see the full example of an anonymous object in Java.

1. **class** Calculation{
2. **void** fact(**int**  n){
3. **int** fact=1;
4. **for**(**int** i=1;i<=n;i++){
5. fact=fact\*i;
6. }
7. System.out.println("factorial is "+fact);
8. }
9. **public** **static** **void** main(String args[]){
10. **new** Calculation().fact(5);//calling method with anonymous object
11. }
12. }

Output:

Factorial is 120

### Creating multiple objects by one type only

We can create multiple objects by one type only as we do in case of primitives.

Initialization of primitive variables:

1. **int** a=10, b=20;

Initialization of refernce variables:

1. Rectangle r1=**new** Rectangle(), r2=**new** Rectangle();//creating two objects

Let's see the example:

1. //Java Program to illustrate the use of Rectangle class which
2. //has length and width data members
3. **class** Rectangle{
4. **int** length;
5. **int** width;
6. **void** insert(**int** l,**int** w){
7. length=l;
8. width=w;
9. }
10. **void** calculateArea(){System.out.println(length\*width);}
11. }
12. **class** TestRectangle2{
13. **public** **static** **void** main(String args[]){
14. Rectangle r1=**new** Rectangle(),r2=**new** Rectangle();//creating two objects
15. r1.insert(11,5);
16. r2.insert(3,15);
17. r1.calculateArea();
18. r2.calculateArea();
19. }
20. }

Output:

55

45

### Real World Example: Account

*File: TestAccount.java*

1. //Java Program to demonstrate the working of a banking-system
2. //where we deposit and withdraw amount from our account.
3. //Creating an Account class which has deposit() and withdraw() methods
4. **class** Account{
5. **int** acc\_no;
6. String name;
7. **float** amount;
8. //Method to initialize object
9. **void** insert(**int** a,String n,**float** amt){
10. acc\_no=a;
11. name=n;
12. amount=amt;
13. }
14. //deposit method
15. **void** deposit(**float** amt){
16. amount=amount+amt;
17. System.out.println(amt+" deposited");
18. }
19. //withdraw method
20. **void** withdraw(**float** amt){
21. **if**(amount<amt){
22. System.out.println("Insufficient Balance");
23. }**else**{
24. amount=amount-amt;
25. System.out.println(amt+" withdrawn");
26. }
27. }
28. //method to check the balance of the account
29. **void** checkBalance(){System.out.println("Balance is: "+amount);}
30. //method to display the values of an object
31. **void** display(){System.out.println(acc\_no+" "+name+" "+amount);}
32. }
33. //Creating a test class to deposit and withdraw amount
34. **class** TestAccount{
35. **public** **static** **void** main(String[] args){
36. Account a1=**new** Account();
37. a1.insert(832345,"Ankit",1000);
38. a1.display();
39. a1.checkBalance();
40. a1.deposit(40000);
41. a1.checkBalance();
42. a1.withdraw(15000);
43. a1.checkBalance();
44. }}

Output:

832345 Ankit 1000.0

Balance is: 1000.0

40000.0 deposited

Balance is: 41000.0

15000.0 withdrawn

Balance is: 26000.0

# **Constructors in Java**

* Types of constructors
* Default Constructor
* Parameterized Constructor
* Constructor Overloading
* Does constructor return any value?
* Copying the values of one object into another
* Does constructor perform other tasks instead of the initialization

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

**Note:** It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

### Rules for creating Java constructor

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

#### **Note: We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.**

## Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor



## Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

### Syntax of default constructor:

1. <class\_name>(){}

## Example of default constructor

|  |
| --- |
| In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation. |

1. //Java Program to create and call a default constructor
2. **class** Bike1{
3. //creating a default constructor
4. Bike1(){System.out.println("Bike is created");}
5. //main method
6. **public** **static** **void** main(String args[]){
7. //calling a default constructor
8. Bike1 b=**new** Bike1();
9. }
10. }

Output:

Bike is created

#### **Rule: If there is no constructor in a class, compiler automatically creates a default constructor.**



### Q) What is the purpose of a default constructor?

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

### Example of default constructor that displays the default values

1. //Let us see another example of default constructor
2. //which displays the default values
3. **class** Student3{
4. **int** id;
5. String name;
6. //method to display the value of id and name
7. **void** display(){System.out.println(id+" "+name);}
9. **public** **static** **void** main(String args[]){
10. //creating objects
11. Student3 s1=**new** Student3();
12. Student3 s2=**new** Student3();
13. //displaying values of the object
14. s1.display();
15. s2.display();
16. }
17. }

Output:

0 null

0 null

**Explanation:**In the above class,you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

### Java Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor.

### Why use the parameterized constructor?

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

### Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

1. //Java Program to demonstrate the use of the parameterized constructor.
2. **class** Student4{
3. **int** id;
4. String name;
5. //creating a parameterized constructor
6. Student4(**int** i,String n){
7. id = i;
8. name = n;
9. }
10. //method to display the values
11. **void** display(){System.out.println(id+" "+name);}
13. **public** **static** **void** main(String args[]){
14. //creating objects and passing values
15. Student4 s1 = **new** Student4(111,"Karan");
16. Student4 s2 = **new** Student4(222,"Aryan");
17. //calling method to display the values of object
18. s1.display();
19. s2.display();
20. }
21. }

Output:

111 Karan

222 Aryan

## Constructor Overloading in Java

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

### Example of Constructor Overloading

1. //Java program to overload constructors
2. **class** Student5{
3. **int** id;
4. String name;
5. **int** age;
6. //creating two arg constructor
7. Student5(**int** i,String n){
8. id = i;
9. name = n;
10. }
11. //creating three arg constructor
12. Student5(**int** i,String n,**int** a){
13. id = i;
14. name = n;
15. age=a;
16. }
17. **void** display(){System.out.println(id+" "+name+" "+age);}
19. **public** **static** **void** main(String args[]){
20. Student5 s1 = **new** Student5(111,"Karan");
21. Student5 s2 = **new** Student5(222,"Aryan",25);
22. s1.display();
23. s2.display();
24. }
25. }

Output:

111 Karan 0

222 Aryan 25

## Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.

|  |  |
| --- | --- |
| **Java Constructor** | **Java Method** |
| A constructor is used to initialize the state of an object. | A method is used to expose the behavior of an object. |
| A constructor must not have a return type. | A method must have a return type. |
| The constructor is invoked implicitly. | The method is invoked explicitly. |
| The Java compiler provides a default constructor if you don't have any constructor in a class. | The method is not provided by the compiler in any case. |
| The constructor name must be same as the class name. | The method name may or may not be same as the class name. |



## Java Copy Constructor

There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in Java. They are:

* By constructor
* By assigning the values of one object into another
* By clone() method of Object class

In this example, we are going to copy the values of one object into another using Java constructor.

1. //Java program to initialize the values from one object to another object.
2. **class** Student6{
3. **int** id;
4. String name;
5. //constructor to initialize integer and string
6. Student6(**int** i,String n){
7. id = i;
8. name = n;
9. }
10. //constructor to initialize another object
11. Student6(Student6 s){
12. id = s.id;
13. name =s.name;
14. }
15. **void** display(){System.out.println(id+" "+name);}
17. **public** **static** **void** main(String args[]){
18. Student6 s1 = **new** Student6(111,"Karan");
19. Student6 s2 = **new** Student6(s1);
20. s1.display();
21. s2.display();
22. }
23. }

Output:

111 Karan

111 Karan

## Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

1. **class** Student7{
2. **int** id;
3. String name;
4. Student7(**int** i,String n){
5. id = i;
6. name = n;
7. }
8. Student7(){}
9. **void** display(){System.out.println(id+" "+name);}
11. **public** **static** **void** main(String args[]){
12. Student7 s1 = **new** Student7(111,"Karan");
13. Student7 s2 = **new** Student7();
14. s2.id=s1.id;
15. s2.name=s1.name;
16. s1.display();
17. s2.display();
18. }
19. }

Output:

111 Karan

111 Karan

### Q) Does constructor return any value?

Yes, it is the current class instance (You cannot use return type yet it returns a value).

### Can constructor perform other tasks instead of initialization?

Yes, like object creation, starting a thread, calling a method, etc. You can perform any operation in the constructor as you perform in the method.

### Is there Constructor class in Java?

Yes.

### What is the purpose of Constructor class?

Java provides a Constructor class which can be used to get the internal information of a constructor in the class. It is found in the java.lang.reflect package.

# **Java static keyword**

* Static variable
* Program of the counter without static variable
* Program of the counter with static variable
* Static method
* Restrictions for the static method
* Why is the main method static?
* Static block
* Can we execute a program without main method?

The **static keyword** in Java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than an instance of the class.

The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class



## 1) Java static variable

If you declare any variable as static, it is known as a static variable.

* The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
* The static variable gets memory only once in the class area at the time of class loading.

### Advantages of static variable

It makes your program **memory efficient** (i.e., it saves memory).

#### **Understanding the problem without static variable**

1. **class** Student{
2. **int** rollno;
3. String name;
4. String college="ITS";
5. }

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all objects. If we make it static, this field will get the memory only once.

#### **Java static property is shared to all objects.**

### Example of static variable

1. //Java Program to demonstrate the use of static variable
2. **class** Student{
3. **int** rollno;//instance variable
4. String name;
5. **static** String college ="ITS";//static variable
6. //constructor
7. Student(**int** r, String n){
8. rollno = r;
9. name = n;
10. }
11. //method to display the values
12. **void** display (){System.out.println(rollno+" "+name+" "+college);}
13. }
14. //Test class to show the values of objects
15. **public** **class** TestStaticVariable1{
16. **public** **static** **void** main(String args[]){
17. Student s1 = **new** Student(111,"Karan");
18. Student s2 = **new** Student(222,"Aryan");
19. //we can change the college of all objects by the single line of code
20. //Student.college="BBDIT";
21. s1.display();
22. s2.display();
23. }
24. }

Output:

111 Karan ITS

222 Aryan ITS



### Program of the counter without static variable

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable. If it is incremented, it won't reflect other objects. So each object will have the value 1 in the count variable.

1. //Java Program to demonstrate the use of an instance variable
2. //which get memory each time when we create an object of the class.
3. **class** Counter{
4. **int** count=0;//will get memory each time when the instance is created
6. Counter(){
7. count++;//incrementing value
8. System.out.println(count);
9. }
11. **public** **static** **void** main(String args[]){
12. //Creating objects
13. Counter c1=**new** Counter();
14. Counter c2=**new** Counter();
15. Counter c3=**new** Counter();
16. }
17. }

Output:

1

1

1

### Program of counter by static variable

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

1. //Java Program to illustrate the use of static variable which
2. //is shared with all objects.
3. **class** Counter2{
4. **static** **int** count=0;//will get memory only once and retain its value
6. Counter2(){
7. count++;//incrementing the value of static variable
8. System.out.println(count);
9. }
11. **public** **static** **void** main(String args[]){
12. //creating objects
13. Counter2 c1=**new** Counter2();
14. Counter2 c2=**new** Counter2();
15. Counter2 c3=**new** Counter2();
16. }
17. }

Output:

1

2

3

## 2) Java static method

If you apply static keyword with any method, it is known as static method.

* A static method belongs to the class rather than the object of a class.
* A static method can be invoked without the need for creating an instance of a class.
* A static method can access static data member and can change the value of it.

### Example of static method

1. //Java Program to demonstrate the use of a static method.
2. **class** Student{
3. **int** rollno;
4. String name;
5. **static** String college = "ITS";
6. //static method to change the value of static variable
7. **static** **void** change(){
8. college = "BBDIT";
9. }
10. //constructor to initialize the variable
11. Student(**int** r, String n){
12. rollno = r;
13. name = n;
14. }
15. //method to display values
16. **void** display(){System.out.println(rollno+" "+name+" "+college);}
17. }
18. //Test class to create and display the values of object
19. **public** **class** TestStaticMethod{
20. **public** **static** **void** main(String args[]){
21. Student.change();//calling change method
22. //creating objects
23. Student s1 = **new** Student(111,"Karan");
24. Student s2 = **new** Student(222,"Aryan");
25. Student s3 = **new** Student(333,"Sonoo");
26. //calling display method
27. s1.display();
28. s2.display();
29. s3.display();
30. }
31. }

Output:111 Karan BBDIT

222 Aryan BBDIT

333 Sonoo BBDIT

### Another example of a static method that performs a normal calculation

1. //Java Program to get the cube of a given number using the static method
3. **class** Calculate{
4. **static** **int** cube(**int** x){
5. **return** x\*x\*x;
6. }
8. **public** **static** **void** main(String args[]){
9. **int** result=Calculate.cube(5);
10. System.out.println(result);
11. }
12. }

Output:125

### Restrictions for the static method

There are two main restrictions for the static method. They are:

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.
3. **class** A{
4. **int** a=40;//non static
6. **public** **static** **void** main(String args[]){
7. System.out.println(a);
8. }
9. }

Output:Compile Time Error

### Q) Why is the Java main method static?

Ans) It is because the object is not required to call a static method. If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory allocation.

## 3) Java static block

* Is used to initialize the static data member.
* It is executed before the main method at the time of classloading.

### Example of static block

1. **class** A2{
2. **static**{System.out.println("static block is invoked");}
3. **public** **static** **void** main(String args[]){
4. System.out.println("Hello main");
5. }
6. }

Output:static block is invoked

Hello main

### Q) Can we execute a program without main() method?

Ans) No, one of the ways was the static block, but it was possible till JDK 1.6. Since JDK 1.7, it is not possible to execute a java class without the main method.

1. **class** A3{
2. **static**{
3. System.out.println("static block is invoked");
4. System.exit(0);
5. }
6. }

Output:

static block is invoked

Since JDK 1.7 and above, output would be:

Error: Main method not found in class A3, please define the main method as:

public static void main(String[] args)

or a JavaFX application class must extend javafx.application.Application

# **this keyword in java**

There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.

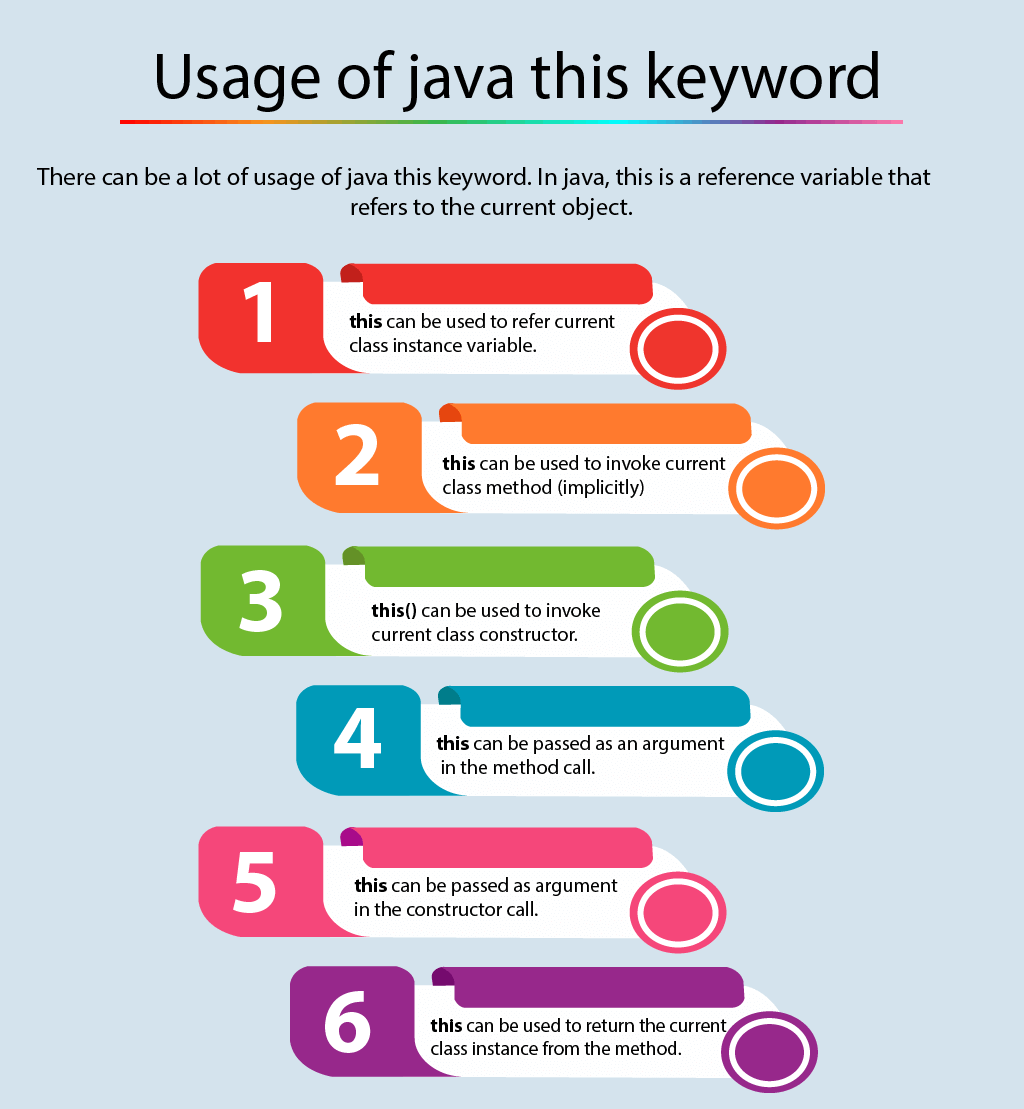


## Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

**Suggestion:** If you are beginner to java, lookup only three usage of this keyword.



### 1) this: to refer current class instance variable

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

#### **Understanding the problem without this keyword**

|  |
| --- |
| Let's understand the problem if we don't use this keyword by the example given below: |

1. **class** Student{
2. **int** rollno;
3. String name;
4. **float** fee;
5. Student(**int** rollno,String name,**float** fee){
6. rollno=rollno;
7. name=name;
8. fee=fee;
9. }
10. **void** display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
12. **class** TestThis1{
13. **public** **static** **void** main(String args[]){
14. Student s1=**new** Student(111,"ankit",5000f);
15. Student s2=**new** Student(112,"sumit",6000f);
16. s1.display();
17. s2.display();
18. }}

Output:

0 null 0.0

0 null 0.0

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

#### **Solution of the above problem by this keyword**

1. **class** Student{
2. **int** rollno;
3. String name;
4. **float** fee;
5. Student(**int** rollno,String name,**float** fee){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.fee=fee;
9. }
10. **void** display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
13. **class** TestThis2{
14. **public** **static** **void** main(String args[]){
15. Student s1=**new** Student(111,"ankit",5000f);
16. Student s2=**new** Student(112,"sumit",6000f);
17. s1.display();
18. s2.display();
19. }}

Output:

111 ankit 5000

112 sumit 6000

If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

#### **Program where this keyword is not required**

1. **class** Student{
2. **int** rollno;
3. String name;
4. **float** fee;
5. Student(**int** r,String n,**float** f){
6. rollno=r;
7. name=n;
8. fee=f;
9. }
10. **void** display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
13. **class** TestThis3{
14. **public** **static** **void** main(String args[]){
15. Student s1=**new** Student(111,"ankit",5000f);
16. Student s2=**new** Student(112,"sumit",6000f);
17. s1.display();
18. s2.display();
19. }}

Output:

111 ankit 5000

112 sumit 6000

#### **It is better approach to use meaningful names for variables. So we use same name for instance variables and parameters in real time, and always use this keyword.**

### 2) this: to invoke current class method

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example



1. **class** A{
2. **void** m(){System.out.println("hello m");}
3. **void** n(){
4. System.out.println("hello n");
5. //m();//same as this.m()
6. **this**.m();
7. }
8. }
9. **class** TestThis4{
10. **public** **static** **void** main(String args[]){
11. A a=**new** A();
12. a.n();
13. }}

Output:

hello n

hello m

### 3) this() : to invoke current class constructor

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

**Calling default constructor from parameterized constructor:**

1. **class** A{
2. A(){System.out.println("hello a");}
3. A(**int** x){
4. **this**();
5. System.out.println(x);
6. }
7. }
8. **class** TestThis5{
9. **public** **static** **void** main(String args[]){
10. A a=**new** A(10);
11. }}

Output:

hello a

10

**Calling parameterized constructor from default constructor:**

1. **class** A{
2. A(){
3. **this**(5);
4. System.out.println("hello a");
5. }
6. A(**int** x){
7. System.out.println(x);
8. }
9. }
10. **class** TestThis6{
11. **public** **static** **void** main(String args[]){
12. A a=**new** A();
13. }}

Output:

5

hello a

### Real usage of this() constructor call

The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

1. **class** Student{
2. **int** rollno;
3. String name,course;
4. **float** fee;
5. Student(**int** rollno,String name,String course){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.course=course;
9. }
10. Student(**int** rollno,String name,String course,**float** fee){
11. **this**(rollno,name,course);//reusing constructor
12. **this**.fee=fee;
13. }
14. **void** display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
15. }
16. **class** TestThis7{
17. **public** **static** **void** main(String args[]){
18. Student s1=**new** Student(111,"ankit","java");
19. Student s2=**new** Student(112,"sumit","java",6000f);
20. s1.display();
21. s2.display();
22. }}

Output:

111 ankit java null

112 sumit java 6000

#### **Rule: Call to this() must be the first statement in constructor.**

1. **class** Student{
2. **int** rollno;
3. String name,course;
4. **float** fee;
5. Student(**int** rollno,String name,String course){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.course=course;
9. }
10. Student(**int** rollno,String name,String course,**float** fee){
11. **this**.fee=fee;
12. **this**(rollno,name,course);//C.T.Error
13. }
14. **void** display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
15. }
16. **class** TestThis8{
17. **public** **static** **void** main(String args[]){
18. Student s1=**new** Student(111,"ankit","java");
19. Student s2=**new** Student(112,"sumit","java",6000f);
20. s1.display();
21. s2.display();
22. }}

Compile Time Error: Call to this must be first statement in constructor

### 4) this: to pass as an argument in the method

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

1. **class** S2{
2. **void** m(S2 obj){
3. System.out.println("method is invoked");
4. }
5. **void** p(){
6. m(**this**);
7. }
8. **public** **static** **void** main(String args[]){
9. S2 s1 = **new** S2();
10. s1.p();
11. }
12. }

Output:

method is invoked

### Application of this that can be passed as an argument:

In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods.

### 5) this: to pass as argument in the constructor call

We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:

1. **class** B{
2. A4 obj;
3. B(A4 obj){
4. **this**.obj=obj;
5. }
6. **void** display(){
7. System.out.println(obj.data);//using data member of A4 class
8. }
9. }
11. **class** A4{
12. **int** data=10;
13. A4(){
14. B b=**new** B(**this**);
15. b.display();
16. }
17. **public** **static** **void** main(String args[]){
18. A4 a=**new** A4();
19. }
20. }

Output:10

### 6) this keyword can be used to return current class instance

We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

### Syntax of this that can be returned as a statement

1. return\_type method\_name(){
2. **return** **this**;
3. }

### Example of this keyword that you return as a statement from the method

1. **class** A{
2. A getA(){
3. **return** **this**;
4. }
5. **void** msg(){System.out.println("Hello java");}
6. }
7. **class** Test1{
8. **public** **static** **void** main(String args[]){
9. **new** A().getA().msg();
10. }
11. }

Output:

Hello java

### Proving this keyword

|  |
| --- |
| Let's prove that this keyword refers to the current class instance variable. In this program, we are printing the reference variable and this, output of both variables are same. |

1. **class** A5{
2. **void** m(){
3. System.out.println(**this**);//prints same reference ID
4. }
5. **public** **static** **void** main(String args[]){
6. A5 obj=**new** A5();
7. System.out.println(obj);//prints the reference ID
8. obj.m();
9. }
10. }

Output:

A5@22b3ea59

A5@22b3ea59

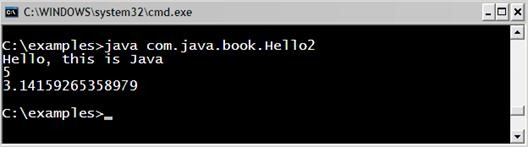
**Using print () and println ()**

Working with the appropriate methods is hassle-free because all the basic types (string, numeric and primitive types) can be printed:

Here are some examples of printing different types of data:

|  |
| --- |
| // Print String  System. *out* .println ( "Hello, this is Java" );    // Print int  System. *out* .println ( 5 );    // Print double  System. *out* .println ( 3.14159265358979 ); |

The result of executing this code looks like this:

[](https://introprogramming.info/wp-content/uploads/2011/07/clip_image015.jpg)

As we can see, using **System.out.println it** is possible to print different types, this is because **for each type there is a predefined version of the println () method in the PrintStream class** (we can see for ourselves by looking at the **PrintStream** class in the API of Java).

**The difference between print (...) and println (...)** , is that the method **print (...)** printed in the console what is submitted between brackets, but does nothing further. While the **println (…)** method is an abbreviation of "print line" which means "print line". This method does what **print (…) does** , but in addition prints a new line. In fact, the method does not print a new line, but simply puts a "command" to move the cursor to the position where the new line begins.

Here's an example that illustrates the difference between **print** and **println** :

|  |
| --- |
| System. *out* .println ( "I love" ) ;  System. *out* .print ( "this" );  System. *out* .print ( "Book!" ); |

The outcome of this example is:

|  |
| --- |
| I love  this Book! |

We note that the output of the example is printed in two lines, even though the code is three. This is because in the first line of the code we use **println () ,** this way " **I love** " is printed and then it goes back to a new line. The next two lines of code use the **print** method **,** which prints without going into a new line , leaving the words " **this** " and " **Book!** " In one line.

## How are parameters passed in Java?

### Java is Strictly Pass by Value!

Consider the following Java program that passes a **primitive type** to function.

public class Main {  
 public static void main(String[] args) {  
 int x = 5;  
 *change*(x);  
 System.*out*.println(x);  
 }  
  
 public static void change(int x) {  
 x = 10;  
 }  
}

Output:

5

We pass an int to the function “change()” and as a result the change in the value of that integer is not reflected in the main method. Like C/C++, Java creates a copy of the variable being passed in the method and then do the manipulations. Hence the change is not reflected in the main method.

**How about objects or references?**

In Java, all primitives like int, char, etc are similar to C/C++, but all non-primitives (or objects of any class) are always references. So it gets tricky when we pass object references to methods. Java creates a copy of references and pass it to method, but they still point to same memory reference. Mean if set some other object to reference passed inside method, the object from calling method as well its reference will remain unaffected.

**The changes are not reflected back if we change the object itself to refer some other location or object**  
If we assign reference to some other location, then changes are not reflected back in main().

// A Java program to show that references are also passed  
// by value.  
 class Test {  
 int x;  
  
 Test(int i) {  
 x = i;  
 }  
  
 Test() {  
 x = 0;  
 }  
 }  
  
 class Main {  
 public static void main(String[] args) {  
 // t is a reference  
 Test t = new Test(5);  
  
 // Reference is passed and a copy of reference  
 // is created in change()  
 *change*(t);  
  
 // Old value of t.x is printed  
 System.*out*.println(t.x);  
 }  
  
 public static void change(Test t) {  
 // We changed reference to refer some other location  
 // now any changes made to reference are not reflected  
 // back in main  
 t = new Test();  
  
 t.x = 10;  
 }  
 }

Output:

5

**Changes are reflected back if we do not assign reference to a new location or object:**  
If we do not change the reference to refer some other object (or memory location), we can make changes to the members and these changes are reflected back.

// A Java program to show that we can change members using using   
// reference if we do not change the reference itself.   
 class Test {  
 int x;  
  
 Test(int i) {  
 x = i;  
 }  
  
 Test() {  
 x = 0;  
 }  
 }  
  
 class Main {  
 public static void main(String[] args) {  
 // t is a reference   
 Test t = new Test(5);  
  
 // Reference is passed and a copy of reference   
 // is created in change()   
 *change*(t);  
  
 // New value of x is printed   
 System.*out*.println(t.x);  
 }  
  
 // This change() doesn't change the reference, it only   
 // changes member of object referred by reference   
 public static void change(Test t) {  
 t.x = 10;  
 }  
 }

Output:

10

In Java, parameters are always passed by value. For example, following program prints i = 10, j = 20.

// Test.java   
class Test {  
 // swap() doesn't swap i and j   
 public static void swap(Integer i, Integer j) {  
 Integer temp = new Integer(i);  
 i = j;  
 j = temp;  
 }  
 public static void main(String[] args) {  
 Integer i = new Integer(10);  
 Integer j = new Integer(20);  
 *swap*(i, j);  
 System.*out*.println("i = " + i + ", j = " + j);  
 }  
}

### Naming a variable - rules

When we want the compiler to allocate an area in memory for some information used in our program, we need to specify a name that serves as an identifier and allows us to refer to the area of ​​memory we need.

The name can be any one of our choice, but it must follow certain rules:

-     The names of the variables are formed by the letters **a** - **z** , **A** - **Z** , the numbers **0** - **9** , as well as the characters **$** and **\_** . It is generally acceptable to use Cyrillic letters, but this should be avoided.

-     Variable names cannot begin with a number.

-     Variable names cannot match a Java language keyword :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **abstract** | **default** | **if** | **private** | **this** |
| **boolean** | **to** | **implements** | **protected** | **throw** |
| **break** | **double** | **import** | **public** | **throws** |
| **byte** | **else** | **instanceof** | **return** | **transient** |
| **case** | **enum** | **int** | **short** | **try** |
| **catch** | **extends** | **interface** | **static** | **void** |
| **char** | **final** | **long** | **strictfp** | **validate** |
| **class** | **finally** | **native** | **great** | **while** |
| **const** | **float** | **new** | **switch** |  |
| **continue** | **for** | **package** | **synchronized** |  |

#### Naming Variables - Recommendations

We will give some naming recommendations, since not all names allowed by the compiler are suitable for our variables.

-     Names should be descriptive - explain what the variable is for. For example, for a person's name the appropriate name is **personName** and the inappropriate name is **a37** .

-     Only Latin letters should be used .

-     In Java, it is accepted that the variables always begin with a lowercase letter and each subsequent word begins with a capital letter. **Ex** : **firstName** , not **firstname** or **first\_name** .

“     The name must be neither too long nor too short .

-     Be careful about uppercase and lowercase letters, as Java makes a difference between them.

Here are some examples of well-named variables:

-      **firstN ame**

-      **age**

-      **startIndex**

-      **lastNegativeNumberIndex**

Here are some examples of badly named variables (though the names are correct from a Java compiler's point of view):

-      **\_firstN ame**

-      **last\_name**

-      **AGE**

-      **Start\_Index**

-      **lastNegativeNumber\_Index**

Variables must have a name that explains briefly what they are for. When a variable is named by an inappropriate name, it makes it very difficult to read the program and then change it (after a time when we forgot how it works).

|  |  |
| --- | --- |
| [clip_image001 [2]](https://introprogramming.info/wp-content/uploads/2011/07/clip_image001215.gif) | **Always try to name the variables with short but clear enough names, always following the rule that the name of the variable should make it clear what it is used for.** |

# **Stack Memory and Heap Space in Java**

## ****1. Introduction****

To run an application in an optimal way, JVM divides memory into stack and heap memory. **Whenever we declare new variables and objects, call new method, declare a String or perform similar operations, JVM designates memory to these operations from either Stack Memory or Heap Space.**

In this tutorial, we'll discuss these memory models. We'll enlist some key differences between them, how they are stored in RAM, the features they offer and where to use them.

## ****2. Stack Memory in Java****

**Stack Memory in Java is used for static memory allocation and the execution of a thread.** It contains primitive values that are specific to a method and references to objects that are in a heap, referred from the method.

Access to this memory is in Last-In-First-Out (LIFO) order. Whenever a new method is called, a new block on top of the stack is created which contains values specific to that method, like primitive variables and references to objects.

When the method finishes execution, it’s corresponding stack frame is flushed, the flow goes back to the calling method and space becomes available for the next method.

### ****2.1. Key Features of Stack Memory****

Apart from what we have discussed so far, following are some other features of stack memory:

* It grows and shrinks as new methods are called and returned respectively
* Variables inside stack exist only as long as the method that created them is running
* It's automatically allocated and deallocated when method finishes execution
* If this memory is full, Java throws java.lang.StackOverFlowError
* Access to this memory is fast when compared to heap memory
* This memory is threadsafe as each thread operates in its own stack

## ****3. Heap Space in Java****

**Heap space in Java is used for dynamic memory allocation for Java objects and JRE classes at the runtime**. New objects are always created in heap space and the references to this objects are stored in stack memory.

These objects have global access and can be accessed from anywhere in the application.

This memory model is further broken into smaller parts called generations, these are:

1. **Young Generation –**this is where all new objects are allocated and aged. A minor Garbage collection occurs when this fills up
2. **Old or Tenured Generation –**this is where long surviving objects are stored. When objects are stored in the Young Generation, a threshold for the object's age is set and when that threshold is reached, the object is moved to the old generation
3. **Permanent Generation –**this consists of JVM metadata for the runtime classes and application methods

These different portions are also discussed in this article – Difference Between JVM, JRE, and JDK.

We can always manipulate the size of heap memory as per our requirement.

### ****3.1. Key Features of Java Heap Memory****

Apart from what we have discussed so far, following are some other features of heap space:

* It's accessed via complex memory management techniques that include Young Generation, Old or Tenured Generation, and Permanent Generation
* If heap space is full, Java throws java.lang.OutOfMemoryError
* Access to this memory is relatively slower than stack memory
* This memory, in contrast to stack, isn't automatically deallocated. It needs Garbage Collector to free up unused objects so as to keep the efficiency of the memory usage
* Unlike stack, a heap isn't threadsafe and needs to be guarded by properly synchronizing the code

## ****4. Example****

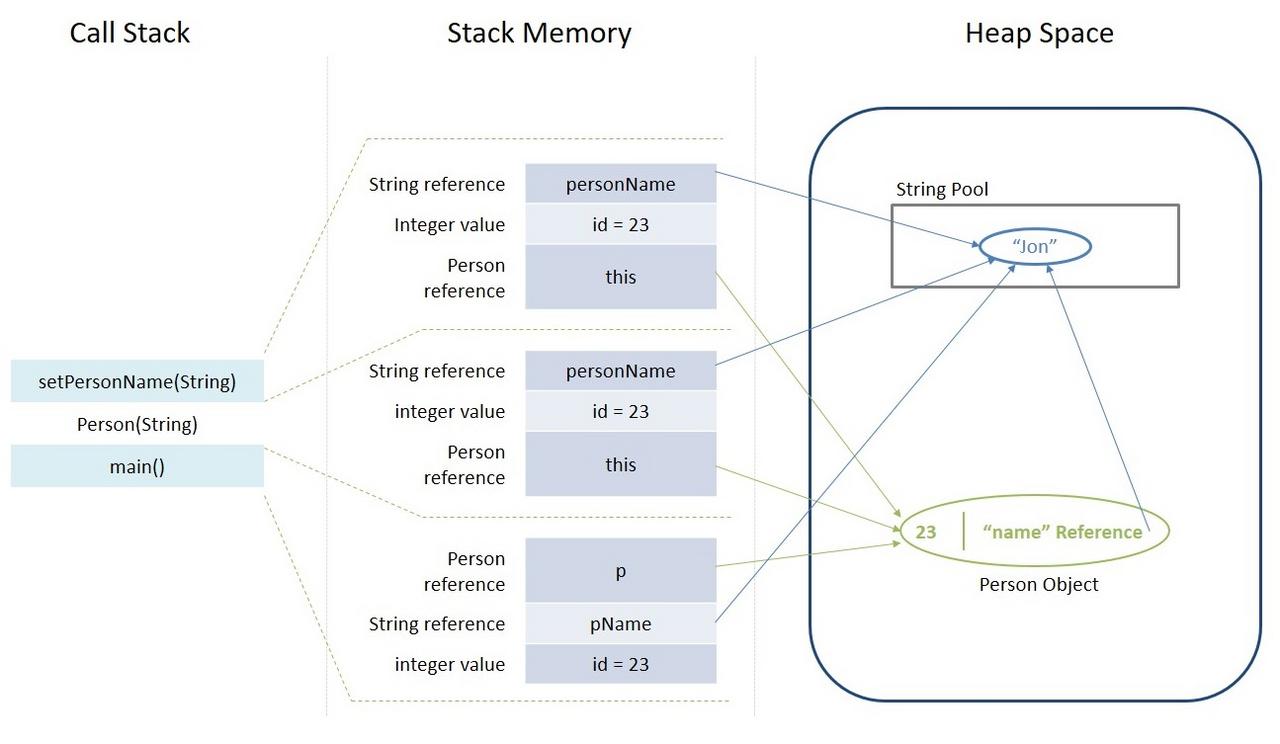
Based on what we've learned so far, let's analyze a simple Java code and let's assess how memory is managed here:

class Person {  
 int pid;  
 String name;  
  
 // constructor, setters/getters  
}  
  
public class Driver {  
 public static void main(String[] args) {  
 int id = 23;  
 String pName = "Jon";  
 Person p = null;  
 p = new Person(id, pName);  
 }  
}

Let's analyze this step by step:

1. Upon entering the main() method, a space in stack memory would be created to store primitives and references of this method
   * The primitive value of integer id will be stored directly in stack memory
   * The reference variable p of type Person will also be created in stack memory which will point to the actual object in the heap
2. The call to the parameterized constructor Person(int, String) from main() will allocate further memory on top of the previous stack. This will store:
   * The this object reference of the calling object in stack memory
   * The primitive value id in the stack memory
   * The reference variable of String argument personName which will point to the actual string from string pool in heap memory
3. This default constructor is further calling setPersonName() method, for which further allocation will take place in stack memory on top of previous one. This will again store variables in the manner described above.
4. However, for the newly created object p of type Person, all instance variables will be stored in heap memory.

This allocation is explained in this diagram:



## ****5. Summary****

Before we conclude this article, let's quickly summarize the differences between the Stack Memory and the Heap Space:

| **Parameter** | **Stack Memory** | **Heap Space** |
| --- | --- | --- |
| Application | Stack is used in parts, one at a time during execution of a thread | The entire application uses Heap space during runtime |
| Size | Stack has size limits depending upon OS and is usually smaller then Heap | There is no size limit on Heap |
| Storage | Stores only primitive variables and references to objects that are created in Heap Space | All the newly created objects are stored here |
| Order | It is accessed using Last-in First-out (LIFO) memory allocation system | This memory is accessed via complex memory management techniques that include Young Generation, Old or Tenured Generation, and Permanent Generation. |
| Life | Stack memory only exists as long as the current method is running | Heap space exists as long as the application runs |
| Efficiency | Comparatively much faster to allocate when compared to heap | Slower to allocate when compared to stack |
| Allocation/Deallocation | This Memory is automatically allocated and deallocated when a method is called and returned respectively | Heap space is allocated when new objects are created and deallocated by Garbage Collector when they are no longer referenced |

## ****6. Conclusion****

Stack and heap are two ways in which Java allocates memory. In this article, we understood how they work and when to use them for developing better Java programs.

## 

## Type conversion

Operators work on the same type of data. However, there is a wide variety of data types in Java that we can choose the most appropriate for the specific purpose. To operate on variables of two different types of data, we need to convert the two types to the same.

All Java language expressions are of type. This type can be derived from the structure of the expression and the types, variables, and literals used in the expression. It is possible to write an expression that is of an inappropriate type for the specific context. In some cases, this will lead to an error in the compilation of the program, but in other contexts it may accept a type that is similar or related to the type of expression. In this case, the program performs a hidden type conversion.

Specific transformation from type **S** to type **T** allows the expression of type **S** to be treated as an expression of type **T** during program execution. In some cases, this will require verification of the validity of the conversion. Here's a few examples:

- An         **Object** to **String** conversion will require a runtime check to confirm that the value is indeed a **String** instance or some of the **String** successor classes .

-         **String** to **Object** conversion does not require verification. **String** is a successor to **Object** and can be converted to its base class without the risk of error or data loss.

-         **Int** to **long** conversion can be performed without runtime checks because there is no risk of data loss.

-        Conversion from **double** to **long** requires conversion from 64-bit floating value to 64-bit integer. Depending on the value, data loss may occur, so an explicit type conversion is required.

In Java, not all types can be converted to all other types, but only to certain types. For convenience, we will group one of the possible Java conversions by their type into two categories:

-        Hidden transformation;

-        Explicit conversion.

### Implicit type conversion

Implicit (hidden) conversion of types is only possible when there is no possibility of data loss during conversion, that is, when we convert from a type with a smaller range to a type with a larger one (for example, from int to **long** ). In order to make an implicit conversion, we do not need to use any operator, so it is called hidden. The conversion is automatically done by the compiler when we assign a value from a smaller range to a larger range variable, or when there are different range types in the expression. Then the conversion goes to the larger scope type.

#### Implicit type conversion - an example

Here's an example of implicit type conversion:

|  |
| --- |
| **int** myInt = 5;  System. *out* .println (myInt); // 5    **long** myLong = myInt;  System. *out* .println (myLong); // 5    System. *out* .println (myLong + myInt); // 10 |

In the example, we create an **int** variable of **myInt** and assign a value of **5** . Below, we create a **myLong** variable of type **long** and set the value contained in **myInt** . The value stored in **myLong** is automatically converted from **int** to **long** . Finally, in the example, we derive the result of the sum of the two variables. Because the variables are of a different type, they are automatically converted to a longer range type, that is, **long,** and the returned result, which is printed on the console, is again **long** . Actually the method parameter passed**println ()** is**long** , but inside the method it will be converted again, this time to a  **String** teamso that it can be printed on the console.

#### Implicit conversions possible

These are the possible implicit transformations of primitive types in Java:

-           **byte** to**short, int, long, float,** or**double**

-           **short** to**int, long, float,** or**double**

-           **char** to**int, long, float,** or**double**

-           **int** to**long, float,** or**double**

-           **long** to**float** or**double**

-           **float** to**double**

There is no loss of data when converting types from smaller ranges to larger ones. The numerical value remains the same after the conversion. As with any rule, there are few exceptions here. When converting an **int** type to a **float** type (32-bit values), the difference is that **int** uses all its bits to represent an integer, while **float** uses some of its bits to represent a floating point. It follows that there may be a loss of precision due to rounding when converting from **int** to **float** . The same applies when converting 64-bit **long** to **double** .

### Explicit type conversion

The explicit conversion of types is necessary when there is a likelihood of data loss. When converting a floating point type to an integer type, there is always a data loss coming from the floating point and the use of an explicit **double** to **long** conversion is mandatory . To make such a conversion, we need to explicitly use the cast operator: ( **type** ). There may also be data loss when converting from a larger range type to a smaller range type ( **double** to **float** or **long** to **int** ).

#### Explicit type conversion - example

The following example illustrates the use of explicit type conversion and data loss:

|  |
| --- |
| **double** myDouble = 5.1d;  System. *out* .println (myDouble); // 5.1    **long** myLong = (**long** ) myDouble;  System. *out* .println (myLong); // 5    myDouble = 5e9d; // 5 \* 10 ^ 9  System. *out* .println (myDouble); // 5.0E9    **int** myInt = (**int** ) myDouble;  System. *out* .println (myInt); // 2147483647  System. *out* .println (Integer. *MAX\_VALUE* ); // 2147483647 |

In the first line of the example, we assign the value 5.1 to the **myDouble** variable . After converting it (explicitly), using the operator **(long)** to type **long** and **passing** the variable **myLong** to the console , we see that the variable has lost value after the floating point (because **long** is an integer type). Then, on the seventh line, we assign the variable **myDouble to** $ 5 billion. Finally, we convert **myDouble** to **int** using the **(int)** operator and print the **myInt** variable . The result is the same as when we print **Integer. MAX\_VALUE,**this is because **myDouble** contains more value than the **int** scope .

|  |  |
| --- | --- |
| [clip_image001 [2]](https://introprogramming.info/wp-content/uploads/2011/07/clip_image001216.gif) | **It is not always possible to predict what the value of a variable will be after the range is full and! So use large enough types and be careful when switching to the smaller type.** |

#### Data loss when converting types

Example of information loss when converting types:

|  |
| --- |
| **long** myLong = Long. *MAX\_VALUE* ;  **int** myInt = (**int** ) myLong;    System. *out* .println (myLong); // 9223372036854775807  System. *out* .println (myInt); // -1 |

The conversion operator can also be used for implicit conversion if desired. This contributes to code readability, reduces the chance of errors, and is considered a good practice by many programmers.

Here are some more examples of types conversion:

|  |
| --- |
| **float** heightInMeters = 1.74f;     // Explicit conversion  **double** maxHeight = heightInMeters;           // Implicit  **double** minHeight = (**double** ) heightInMeters; // Explicit  **float** actualHeight = (**float** ) maxHeight;      // Explicit    **float** maxHeightFloat = maxHeight; // Compilation error! |

In the example of the last line, we have an expression that will generate a compilation error. This is because we are trying to implicitly convert from **double** to **float** , which may cause data loss. Java is a strictly typed programming language and does not allow this kind of value assignment.

#### Explicit conversions are possible

These are possible explicit (explicit) conversions and all of them have the potential for data loss, so be careful:

-        **short** to **byte** or **char**

-        **char** to **byte** or **short**

-        **int** to **byte** , **short** or **char**

-        **long** to **byte** , **short** , **char** or **int**

-        **float** to **byte** , **short** , **char** , **int** or **long**

-        **double** to **byte** , **short** , **char** , **int** , **long** or **float**

These conversions can lose both number size information and precision information.

When converting **byte** to char, we first have a hidden conversion from byte to **int** , followed by an explicit conversion from **int** to **char** .

### Convert to a character string

If necessary, we can convert to a string, any single type, including the value **null** . Character strings are automatically converted whenever we use the concatenation operator and one of the arguments is not of string type. In this case, the argument is converted to a string and the operator returns a new string representing the concatenation of the two strings.

Another way to convert different objects to a string type is by calling the variable's **toString ()** method .

#### Conversion to character string - example

Let's look at some examples of converting different types of data to a character string:

|  |
| --- |
| **int** a = 5;  **int** b = 7;  String s = "Sum =" + (a + b);  System. *out* .println (s);    String incorrect = "Sum =" + a + b;  System. *out* .println (incorrect);    System. *out* .println ("Perimeter =" + 2 \* (a + b) + ". Area =" + (a \* b) + "."); |

The result of the example implementation is as follows:

|  |
| --- |
| Sum = 12  Sum = 57  Perimeter = 24. Area = 35. |

The result shows that the attachment of a number to a character string returns as a result the character string followed by the text representation of the number. Note that the " **+** " operator for string stitching can have an unpleasant effect on summation because it has the same priority as the " **+** " operator on summation. Unless we explicitly change the priority of operations by placing parentheses, they are always executed from left to right.

# **Why String is Immutable in Java?**

## ****1. Introduction****

In Java, Strings are immutable. An obvious question that is quite prevalent in interviews is “Why Strings are designed as immutable in Java?”

I would use an immutable whenever I can.

He further supports his argument stating features that immutability provides, such as caching, security, easy reuse without replication, etc.

In this tutorial, we'll further explore why the Java language designers decided to keep String immutable.

## 2. What is an Immutable Object?

An immutable object is an **object whose internal state remains constant after it has been entirely created**. This means that once the object has been assigned to a variable, we can neither update the reference nor mutate the internal state by any means.

We have a separate article that discusses immutable objects in detail.

## ****3. Why is****String****Immutable in Java?****

The key benefits of keeping this class as immutable are caching, security, synchronization, and performance.

Let's discuss how these things work.

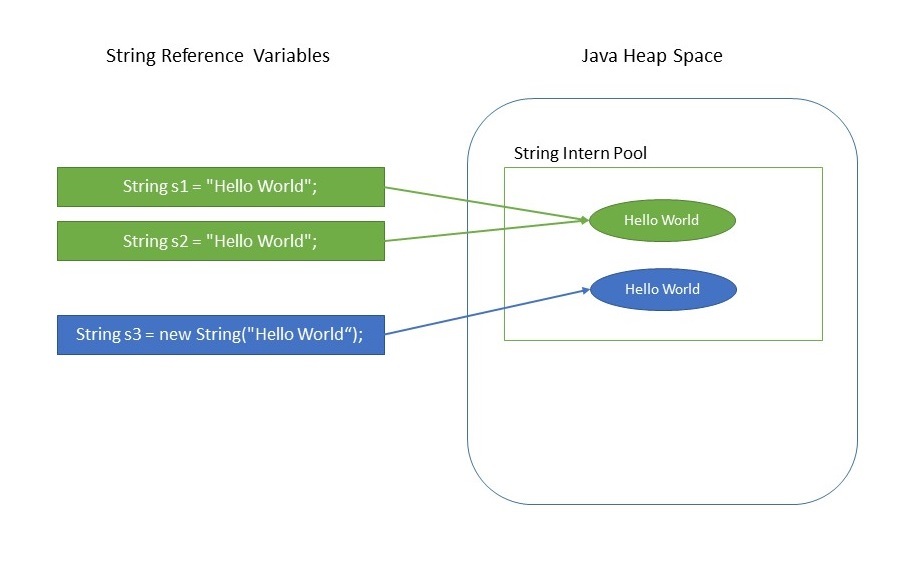
### ****3.1. Introduce to****String****Pool****

The String is the most widely used data structure. Caching the String literals and reusing them saves a lot of heap space because different String variables refer to the same object in the String pool. String intern pool serves exactly this purpose.

Java String Pool is **the special memory region where Strings are stored by the JVM**. Since Strings are immutable in Java, the JVM optimizes the amount of memory allocated for them by storing only one copy of each literal String in the pool. This process is called interning:

String s1 = "Hello World";  
String s2 = "Hello World";  
  
assertThat(s1 ==s2).  
  
isTrue();

Because of the presence of the String pool in the preceding example, two different variables are pointing to same String object from the pool, thus saving crucial memory resource.

[](https://www.baeldung.com/wp-content/uploads/2018/08/Why_String_Is_Immutable_In_Java.jpg)

We have a separate article dedicated to Java String Pool.

### ****3.2. Security****

The String is widely used in Java applications to store sensitive pieces of information like usernames, passwords, connection URLs, network connections, etc. It's also used extensively by JVM class loaders while loading classes.

Hence securing String class is crucial regarding the security of the whole application in general. For example, consider this simple code snippet:

void criticalMethod(String userName) {  
 // perform security checks  
 if (!isAlphaNumeric(userName)) {  
 throw new SecurityException();  
 }  
  
 // do some secondary tasks  
 initializeDatabase();  
  
 // critical task  
 connection.executeUpdate("UPDATE Customers SET Status = 'Active' " +  
 " WHERE UserName = '" + userName + "'");  
}

In the above code snippet, let's say that we received a String object from an untrustworthy source. We're doing all necessary security checks initially to check if the String is only alphanumeric, followed by some more operations.

Remember that our unreliable source caller method still has reference to this userName object.

**If Strings were mutable, then by the time we execute the update, we can't be sure that the String we received, even after performing security checks, would be safe.** The untrustworthy caller method still has the reference and can change the String between integrity checks. Thus making our query prone to SQL injections in this case. So mutable Strings could lead to degradation of security over time.

It could also happen that the String userName is visible to another thread, which could then change its value after the integrity check.

In general, immutability comes to our rescue in this case because it's easier to operate with sensitive code when values don't change because there are fewer interleavings of operations that might affect the result.

### ****3.3. Synchronization****

Being immutable automatically makes the String thread safe since they won't be changed when accessed from multiple threads.

Hence **immutable objects, in general, can be shared across multiple threads running simultaneously. They're also thread-safe** because if a thread changes the value, then instead of modifying the same, a new String would be created in the String pool. Hence, Strings are safe for multi-threading.

### ****3.4. Hashcode Caching****

Since String objects are abundantly used as a data structure, they are also widely used in hash implementations like HashMap, HashTable, HashSet, etc. When operating upon these hash implementations, hashCode() method is called quite frequently for bucketing.

The immutability guarantees Strings that their value won’t change. So **the hashCode() method is overridden in String class to facilitate caching, such that the hash is calculated and cached during the first hashCode() call and the same value is returned ever since.**

**This, in turn, improves the performance of collections that uses hash implementations when operated with String objects.**

On the other hand, mutable Strings would produce two different hashcodes at the time of insertion and retrieval if contents of String was modified after the operation, potentially losing the value object in the Map.

### ****3.5. Performance****

As we saw previously, String pool exists because Strings are immutable. In turn, it enhances the performance by saving heap memory and faster access of hash implementations when operated with Strings.

Since String is the most widely used data structure, improving the performance of String have a considerable effect on improving the performance of the whole application in general.

## ****4. Conclusion****

Through this article, we can conclude that **Strings are immutable precisely so that their references can be treated as a normal variable and one can pass them around, between methods and across threads, without worrying about whether the actual String object it's pointing to will change.**

We also learned as what might be the other reasons that prompted the Java language designers to make this class as immutable.

# Chapter 5.

## Class

## Difference between static and non-static variables in Java

There are three types of variables in Java:

* Local Variables
* Instance Variables
* Static Variables

The Local variables and Instance variables are together called Non-Static variables. Hence it can also be said that the Java variables can be divided into 2 categories:

* **Static Variables:** When a variable is declared as static, then a single copy of the variable is created and shared among all objects at a class level. Static variables are, essentially, global variables. All instances of the class share the same static variable.

**Important points for static variables :-**

* + We can create static variables at class-level only.
  + static block and static variables are executed in order they are present in a program.

Below is the Java program to demonstrate that static block and static variables are executed in order they are present in a program.

// Java program to demonstrate execution   
// of static blocks and variables   
 class Test {  
 // static variable   
 static int *a* = *m1*();  
  
 // static block   
 static {  
 System.*out*.println("Inside static block");  
 }  
  
 // static method   
 static int m1() {  
 System.*out*.println("from m1");  
 return 20;  
 }  
  
 // static method(main !!)   
 public static void main(String[] args) {  
 System.*out*.println("Value of a : " + *a*);  
 System.*out*.println("from main");  
 }  
 }

**Output:**

from m1

Inside static block

Value of a : 20

from main

* **Non-Static Variable**
  + **Local Variables**: A variable defined within a block or method or constructor is called local variable.
    - These variable are created when the block in entered or the function is called and destroyed after exiting from the block or when the call returns from the function.
    - The scope of these variables exists only within the block in which the variable is declared. i.e. we can access these variable only within that block.
    - Initialisation of Local Variable is Mandatory.
  + **Instance Variables:** Instance variables are non-static variables and are declared in a class outside any method, constructor or block.
    - As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
    - Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier then the default access specifier will be used.
    - Initialisation of Instance Variable is not Mandatory. Its default value is 0
    - Instance Variable can be accessed only by creating objects.

**Example :**

// Java program to demonstrate   
// non-static variables   
  
 class GfG {  
  
 // non-static variable   
 int rk = 10;  
  
 public static void main(String[] args) {  
 // Instance created inorder to access   
 // a non static variable.   
 Gfg f = new Gfg();  
  
 System.*out*.println("Non static variable"  
 + " accessed using instance"  
 + " of a class");  
 System.*out*.println("Non Static variable "  
 + f.rk);  
 }  
 }

**Output:**

Non static variable accessed using instance of a class.

Non Static variable 10

**The main differences between static and non static variables are:**

|  |  |
| --- | --- |
| **STATIC VARIABLE** | **NON STATIC VARIABLE** |
| Static variables can be accessed using class name | Non static variables can be accessed using instance of a class |
| Static variables can be accessed by static and non static methods | Non static variables cannot be accessed inside a static method. |
| Static variables reduce the amount of memory used by a program. | Non static variables do not reduce the amount of memory used by a program |
| Static variables are shared among all instances of a class. | Non static variables are specific to that instance of a class. |
| Static variable is like a global variable and is available to all methods. | Non static variable is like a local variable and they can be accessed through only instance of a class. |

## Object Oriented Programming (OOP)

Object - oriented programming is the successor to procedural (structural) programming. Procedural programming generally describes programs through a group of reusable pieces of code (procedures) that define input and output parameters. Procedural programs are a collection of procedures that call one another.

The problem with procedural programming is that code reuse is difficult to achieve and limited - only procedures can be reused, and they can hardly be made generic and flexible. There is no easy way to implement abstract data structures that have different implementations.

The object-oriented approach relies on the paradigm that each program works with data describing real-life entities (objects and phenomena). For example, an accounting program works with invoices, goods, warehouses, stocks, sales, etc.

This is how objects appear - they describe the characteristics (properties) and behavior (methods) of these real-life entities.

**Main advantages and objectives of OOP** - to enable faster development of complex software and easier maintenance. CMO allows for easy re-use of the code by relying on simple and generally accepted rules (principles). Let's look at them.

## Basic principles of OOP

In order to be an object-oriented programming language, it must not only enable the use of classes and objects, but must also enable the implementation and use of CMO principles and concepts: inheritance, abstraction, encapsulation and polymorphism. We will now look at each of these basic CMO principles in detail.

-      **Inheritance**

-      **Abstraction**

-      **Encapsulation**

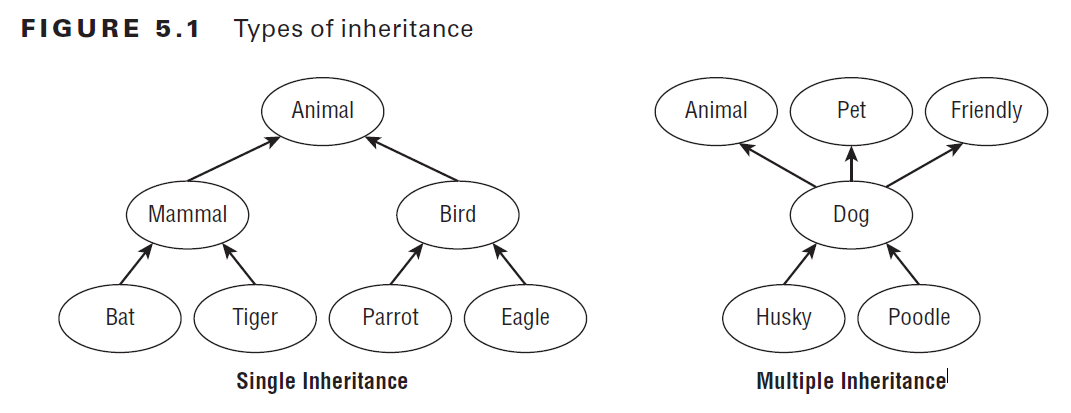
-      **Polymorphism**

# **Inheritance**

**Inheritance** is a basic principle of object-oriented programming. It allows one class to "inherit" (behavior and characteristics) from another, more general class. For example, the lion is from a cat family. All cats have four paws, they are predators, they chase their victims. This functionality can be written once in a Cat class and all predators can use it - tiger, cougar, lynx, etc.

## How to define inheritance in Java?

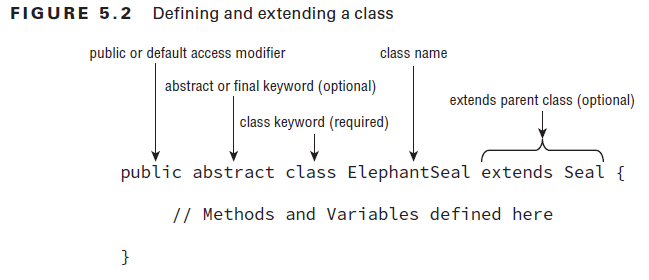
Java inheritance is done with the keyword **extends** . In Java and other advanced programming languages one class can inherit only one other class **(single inheritance)** , unlike C ++, where support multiple inheritance **(multiple inheritance)** . The limitation is that when inheriting two classes with the same method, it is difficult to decide which one to use (in C ++ this problem is very complicated). Many interfaces can be inherited in Java , which we will talk about later.



The class we inherit is called the **parent class** or another **base class (base class, super class)** .

## Class inheritance - an example

Let's look at an example of class inheritance in Java. Here's what the base (parent) class looks like :



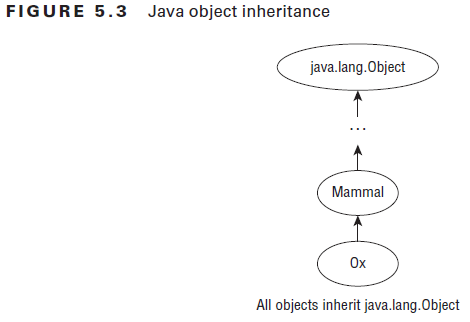
**Applying Class Access Modifiers**

You can apply access modifiers **(public, private, protected, default)** to both class methods and variables. It probably comes as no surprise that you can also apply access modifiers to class definitions, since we have been adding the public access modifier to nearly every class up to now.

Creating Java Objects

public class Zoo {  
}

public class Zoo extends java.lang.Object {  
}



|  |
| --- |
| **Felidae.java** |
| **package** introjavabook;    **public class** Felidae { // Latin word for " cat "  **private boolean** male ;  **public** Felidae () {  **this** (**true** );      }  **public** Felidae (**boolean** male) {  **this**.small = small;      }  **public boolean** isMale () {  **return** male ;      }  **public void** setMale (**boolean** male) {  **this**.small = small;      }  } |

Here's how it looks and class - successor **Lion** :

|  |
| --- |
| **Lion.java** |
| **package** introjavabook;    **public class** Lion**extends** Felidae {  **private int** weight ;  **public** Lion (**boolean** male,**int** weight) {  **super**(small); // Shall be explained in the next paragraph  **this**.weight = weight;      }  **public int** getWeight () {  **return** weight ;      }  **public void** setWeight (**int** weight) {  **this**.weight = weight;      }  } |

## Using final with Inheritance in Java

final is a keyword in java used for restricting some functionalities. We can declare variables, methods and classes with final keyword.

**Using final with inheritance**

During inheritance, we must declare methods with final keyword for which we required to follow the same implementation throughout all the derived classes. Note that it is not necessary to declare final methods in the initial stage of inheritance(base class always). We can declare final method in any subclass for which we want that if any other class extends this subclass, then it must follow same implementation of the method as in the that subclass.

public class FinalKeyword {  
 // Java program to illustrate  
// use of final with inheritance  
  
 // base class  
 abstract class Shape {  
 private double width;  
  
 private double height;  
  
 // Shape class parameterized constructor  
 public Shape(double width, double height) {  
 this.width = width;  
 this.height = height;  
 }  
  
 // getWidth method is declared as final  
 // so any class extending  
 // Shape cann't override it  
 public final double getWidth() {  
 return width;  
 }  
  
 // getHeight method is declared as final  
 // so any class extending Shape  
 // can not override it  
 public final double getHeight() {  
 return height;  
 }  
  
  
 // method getArea() declared abstract because  
 // it upon its subclasses to provide  
 // complete implementation  
 abstract double getArea();  
 }  
  
 // derived class one  
 class Rectangle extends Shape {  
 // Rectangle class parameterized constructor  
 public Rectangle(double width, double height) {  
 // calling Shape class constructor  
 super(width, height);  
 }  
  
 // getArea method is overridden and declared  
 // as final so any class extending  
 // Rectangle cann't override it  
 @Override  
 final double getArea() {  
 return this.getHeight() \* this.getWidth();  
 }  
  
 }  
  
 //derived class two  
 class Square extends Shape {  
 // Rectangle class parameterized constructor  
 public Square(double side) {  
 // calling Shape class constructor  
 super(side, side);  
 }  
  
 // getArea method is overridden and declared as  
 // final so any class extending  
 // Square cann't override it  
 @Override  
 final double getArea() {  
 return this.getHeight() \* this.getWidth();  
 }  
  
 }  
  
 // Driver class  
 public class Test {  
 public void main(String[] args) {  
 // creating Rectangle object  
 Shape s1 = new Rectangle(10, 20);  
  
 // creating Square object  
 Shape s2 = new Square(10);  
  
 // getting width and height of s1  
 System.*out*.println("width of s1 : " + s1.getWidth());  
 System.*out*.println("height of s1 : " + s1.getHeight());  
  
 // getting width and height of s2  
 System.*out*.println("width of s2 : " + s2.getWidth());  
 System.*out*.println("height of s2 : " + s2.getHeight());  
  
 //getting area of s1  
 System.*out*.println("area of s1 : " + s1.getArea());  
  
 //getting area of s2  
 System.*out*.println("area of s2 : " + s2.getArea());  
  
 }  
 }  
}

Output:

width of s1 : 10.0

height of s1 : 20.0

width of s2 : 10.0

height of s2 : 10.0

area of s1 : 200.0

area of s2 : 100.0

**Using final to Prevent Inheritance**

When a class is declared as final then it cannot be subclassed i.e. no any other class can extend it. This is particularly useful, for example, when [creating an immutable class](https://www.geeksforgeeks.org/create-immutable-class-java/) like the predefined [String](http://quiz.geeksforgeeks.org/string-class-in-java/)class. The following fragment illustrates **final** keyword with a class:

final class A

{

// methods and fields

}

// The following class is illegal.

class B extends A

{

// ERROR! Can't subclass A

}

**Note :**

* Declaring a class as final implicitly declares all of its methods as final, too.
* It is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations. For more on abstract classes, refer [abstract classes in java](https://www.geeksforgeeks.org/abstract-classes-in-java/)

**Using final to Prevent Overriding**

When a method is declared as final then it cannot be overridden by subclasses.The [Object](https://www.geeksforgeeks.org/object-class-in-java/) class does this—a number of its methods are final. The following fragment illustrates **final** keyword with a method:

class A

{

final void m1()

{

System.out.println("This is a final method.");

}

}

class B extends A

{

void m1()

{

// ERROR! Can't override.

System.out.println("Illegal!");

}

}

Normally, Java resolves calls to methods dynamically, at run time. This is called [**late or dynamic binding**](https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/). However, since final methods cannot be overridden, a call to one can be resolved at compile time. This is called [**early or static binding**](https://www.geeksforgeeks.org/static-vs-dynamic-binding-in-java/).

## The keyword super

In the example above, in the **Lion** constructor, we use the keyword **super** . It specifies that the base class be used and allows access to its methods, constructors, and member variables. With **super ()** we can call a base class constructor. With **super.method ()** we can call a base class method, pass parameters to it, and use the result of it. With **super.field** we can take the value of a member variable of the base class or assign it a different value.

In Java, inherited from the base class methods can be **rewritten** ( **override** ). This means replacing their implementation by ignoring the original source code from the base class and replacing it with another code. We will explain more about rewriting methods in the [Virtual Methods](https://introprogramming.info/intro-java-book/read-online/glava20-principi-na-obektno-orientiranoto-programirane/#_%D0%92%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D0%BD%D0%B8_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B8) section.

We can call a non-written method from base class and without **super** . The use of a keyword is only necessary if we have a rewritten method or variable of the same name in the inherited class.

|  |  |
| --- | --- |
| [clip_image001](https://introprogramming.info/wp-content/uploads/2011/07/clip_image00168.gif) | **super can be used explicitly for clarity.**super**.method ()  calls a method that is necessarily in the base class. Such code is easier to read because we know where to look for the method in question.**  **Note that this is not the case. this can mean both a method of a particular class and a method of any base class.** |

You can look at the example in the section [access levels in succession](https://introprogramming.info/intro-java-book/read-online/glava20-principi-na-obektno-orientiranoto-programirane/#_%D0%9D%D0%B8%D0%B2%D0%B0_%D0%BD%D0%B0_%D0%B4%D0%BE%D1%81%D1%82%D1%8A%D0%BF) . It clearly shows which members (methods, constructors, and member variables) of the base class we have access to.

## Super Keyword in Java

The**super** keyword in java is a reference variable that is used to refer parent class objects.  The keyword “super” came into the picture with the concept of Inheritance. It is majorly used in the following contexts:

**1. Use of super with variables:**This scenario occurs when a derived class and base class has same data members. In that case there is a possibility of ambiguity for the JVM. We can understand it more clearly using this code snippet:

/\* Base class vehicle \*/  
class Vehicle  
{  
 int maxSpeed = 120;  
}  
  
/\* sub class Car extending vehicle \*/  
class Car extends Vehicle  
{  
 int maxSpeed = 180;  
  
 void display()  
 {  
 /\* print maxSpeed of base class (vehicle) \*/  
 System.*out*.println("Maximum Speed: " + super.maxSpeed);  
 }  
}  
  
/\* Driver program to test \*/  
class Test  
{  
 public static void main(String[] args)  
 {  
 Car small = new Car();  
 small.display();  
 }  
}

Output:

Maximum Speed: 120

In the above example, both base class and subclass have a member maxSpeed. We could access maxSpeed of base class in sublcass using super keyword.

**2. Use of super with methods:**This is used when we want to call parent class method. So whenever a parent and child class have same named methods then to resolve ambiguity we use super keyword. This code snippet helps to understand the said usage of super keyword.

/\* Base class Person \*/  
class Person  
{  
 void message()  
 {  
 System.*out*.println("This is person class");  
 }  
}  
  
/\* Subclass Student \*/  
class Student extends Person  
{  
 void message()  
 {  
 System.*out*.println("This is student class");  
 }  
  
 // Note that display() is only in Student class   
 void display()  
 {  
 // will invoke or call current class message() method   
 message();  
  
 // will invoke or call parent class message() method   
 super.message();  
 }  
}  
  
/\* Driver program to test \*/  
class Test  
{  
 public static void main(String args[])  
 {  
 Student s = new Student();  
  
 // calling display() of Student   
 s.display();  
 }  
}

Output:

This is student class

This is person class

In the above example, we have seen that if we only call method message() then, the current class message() is invoked but with the use of super keyword, message() of superclass could also be invoked.

**3**. **Use of super with constructors:**super keyword can also be used to access the parent class constructor. One more important thing is that, ‘’super’ can call both parametric as well as non parametric constructors depending upon the situation. Following is the code snippet to explain the above concept:

/\* superclass Person \*/  
class Person {  
 Person() {  
 System.*out*.println("Person class Constructor");  
 }  
}  
  
/\* subclass Student extending the Person class \*/  
class Student extends Person {  
 Student() {  
 // invoke or call parent class constructor   
 super();  
  
 System.*out*.println("Student class Constructor");  
 }  
}  
  
/\* Driver program to test\*/  
class Test {  
 public void main(String[] args) {  
 Student s = new Student();  
 }  
}

Output:

Person class Constructor

Student class Constructor

In the above example we have called the superclass constructor using keyword ‘super’ via subclass constructor.

**Other Important points:**

1. Call to super() must be first statement in Derived(Student) Class constructor.
2. If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error. Object does have such a constructor, so if Object is the only superclass, there is no problem.
3. If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that a whole chain of constructors called, all the way back to the constructor of Object. This, in fact, is the case. It is called constructor chaining..

### Inheritance questions

class q1 {  
 class Base {  
 public void show() {  
 System.*out*.println("Base::show() called");  
 }  
 }  
  
 class Derived extends Base {  
 public void show() {  
 System.*out*.println("Derived::show() called");  
 }  
 }  
  
 public class Main {  
 public void main(String[] args) {  
 Base b = new Derived();;  
 b.show();  
 }  
 }  
 //(A) Derived::show() called  
 //(B) Base::show() called  
 //  
 //  
 // Answer: (A)  
 //  
 // Explanation: In the above program, b is a reference of Base type and refers to an abject of Derived class.  
 //  
 // In Java, functions are virtual by default. So the run time polymorphism happens and derived fun() is called.  
  
}

public class q2 {  
 class Base {  
 final public void show() {  
 System.*out*.println("Base::show() called");  
 }  
 }  
  
 class Derived extends Base {  
 public void show() {  
 System.*out*.println("Derived::show() called");  
 }  
 }  
  
 class Main {  
 public void main(String[] args) {  
 Base b = new Derived();;  
 b.show();  
 }  
 }  
//(A) Base::show() called  
//(B) Derived::show() called  
//(C) Compiler Error  
//(D) Runtime Error  
//  
//  
// Answer: (C)  
//  
// Explanation: Final methods cannot be overridden. See the compiler error here.  
  
}

class Base {  
 public static void show() {  
 System.*out*.println("Base::show() called");  
 }  
}  
  
class Derived extends Base {  
 public static void show() {  
 System.*out*.println("Derived::show() called");  
 }  
}  
  
class Main {  
 public static void main(String[] args) {  
 Base b = new Derived();;  
 b.*show*();  
 }  
}   
//(A) Base::show() called  
// (B) Derived::show() called  
// (C) Compiler Error  
//  
//  
// Answer: (A)  
//  
// Explanation: Like C++, when a function is static, runtime polymorphism doesn’t happen.

# **The “final” Keyword in Java**

## ****1. Overview****

While inheritance enables us to reuse existing code, sometimes we do need to **set limitations on extensibility** for various reasons; the final keyword allows us to do exactly that.

In this tutorial, we'll take a look at what the final keyword means for classes, methods, and variables.

## ****2.****Final****Classes****

**Classes marked as final can’t be extended.** If we look at the code of Java core libraries, we’ll find many final classes there. One example is the String class.

Consider the situation if we can extend the String class, override any of its methods, and substitute all the String instances with the instances of our specific String subclass.

The result of the operations over String objects will then become unpredictable. And given that the String class is used everywhere, it’s unacceptable. That’s why the String class is marked as final.

Any attempt to inherit from a final class will cause a compiler error. To demonstrate this, let’s create the final class Cat:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public final class Cat {        private int weight;        // standard getter and setter  } |

And let’s try to extend it:

|  |  |
| --- | --- |
| 1  2 | public class BlackCat extends Cat {  } |

We’ll see the compiler error:

|  |  |
| --- | --- |
| 1 | The type BlackCat cannot subclass the final class Cat |

Note that **the final keyword in a class declaration doesn’t mean that the objects of this class are immutable**. We can change the fields of Cat object freely:

|  |  |
| --- | --- |
| 1  2  3  4 | Cat cat = new Cat();  cat.setWeight(1);    assertEquals(1, cat.getWeight()); |

We just can’t extend it.

If we follow the rules of good design strictly, we should create and document a class carefully or declare it final for safety reasons. However, we should use caution when creating final classes.

Notice that making a class final means that no other programmer can improve it. Imagine that we're using a class and don’t have the source code for it, and there's a problem with one method.

If the class is final, we can’t extend it to override the method and fix the problem. In other words, we lose extensibility, one of the benefits of object-oriented programming.

## ****3.****Final****Methods****

**Methods marked as final cannot be overridden.** When we design a class and feel that a method shouldn’t be overridden, we can make this method final. We can also find many final methods in Java core libraries.

Sometimes we don’t need to prohibit a class extension entirely, but only prevent overriding of some methods. A good example of this is the Thread class. It’s legal to extend it and thus create a custom thread class. But its isAlive() methods is final.

This method checks if a thread is alive. It’s impossible to override the isAlive() method correctly for many reasons. One of them is that this method is native. Native code is implemented in another programming language and is often specific to the operating system and hardware it's running on.

Let’s create a Dog class and make its sound() method final:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class Dog {      public final void sound() {          // ...      }  } |

Now let’s extend the Dog class and try to override its sound() method:

|  |  |
| --- | --- |
| 1  2  3  4 | public class BlackDog extends Dog {      public void sound() {      }  } |

We’ll see the compiler error:

|  |  |
| --- | --- |
| 1  2  3  4 | - overrides  com.baeldung.finalkeyword.Dog.sound  - Cannot override the final method from Dog  sound() method is final and can’t be overridden |

If some methods of our class are called by other methods, we should consider making the called methods final. Otherwise, overriding them can affect the work of callers and cause surprising results.

If our constructor calls other methods, we should generally declare these methods final for the above reason.

What’s the difference between making all methods of the class final and marking the class itself final? In the first case, we can extend the class and add new methods to it.

In the second case, we can’t do this.

## ****4.****Final****Variables****

**Variables marked as final can't be reassigned.** Once a final variable is initialized, it can’t be altered.

### ****4.1.****Final****Primitive Variables****

Let’s declare a primitive final variable i, then assign 1 to it.

And let’s try to assign a value of 2 to it:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public void whenFinalVariableAssign\_thenOnlyOnce() {      final int i = 1;      //...      i=2;  } |

The compiler says:

|  |  |
| --- | --- |
| 1 | The final local variable i may already have been assigned |

### ****4.2.****Final****Reference Variables****

If we have a final reference variable, we can’t reassign it either. But **this doesn’t mean that the object it refers to is immutable**. We can change the properties of this object freely.

To demonstrate this, let’s declare the final reference variable cat and initialize it:

|  |  |
| --- | --- |
| 1 | final Cat cat = new Cat(); |

If we try to reassign it we’ll see a compiler error:

|  |  |
| --- | --- |
| 1 | The final local variable cat cannot be assigned. It must be blank and not using a compound assignment |

But we can change the properties of Cat instance:

|  |  |
| --- | --- |
| 1  2  3 | cat.setWeight(5);    assertEquals(5, cat.getWeight()); |

### ****4.3.****Final****Fields****

**Final fields can be either constants or write-once fields.** To distinguish them, we should ask a question — would we include this field if we were to serialize the object? If no, then it’s not part of the object, but a constant.

Note that according to naming conventions, class constants should be uppercase, with components separated by underscore (“\_”) characters:

|  |  |
| --- | --- |
| 1 | static final int MAX\_WIDTH = 999; |

Note that **any final field must be initialized before the constructor completes**.

For static final fields, this means that we can initialize them:

* upon declaration as shown in the above example
* in the static initializer block

For instance final fields, this means that we can initialize them:

* upon declaration
* in the instance initializer block
* in the constructor

Otherwise, the compiler will give us an error.

### ****4.4.****Final****Arguments****

The final keyword is also legal to put before method arguments. **A final argument can’t be changed inside a method**:

|  |  |
| --- | --- |
| 1  2  3 | public void methodWithFinalArguments(final int x) {      x=1;  } |

The above assignment causes the compiler error:

|  |  |
| --- | --- |
| 1 | The final local variable x cannot be assigned. It must be blank and not using a compound assignment |

## ****5. Conclusion****

In this article, we learned what the final keyword means for classes, methods, and variables. Although we may not use the final keyword often in our internal code, it may be a good design solution.

As always, the complete code for this article can be found in the [GitHub project](https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-lang-oop).

# **Abstraction**

Data Abstraction is the property by virtue of which only the essential details are displayed to the user.The trivial or the non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.

Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details.The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

In java, abstraction is achieved by [interfaces](https://www.geeksforgeeks.org/interfaces-in-java/) and [abstract classes](https://www.geeksforgeeks.org/abstract-classes-in-java/). We can achieve 100% abstraction using interfaces.

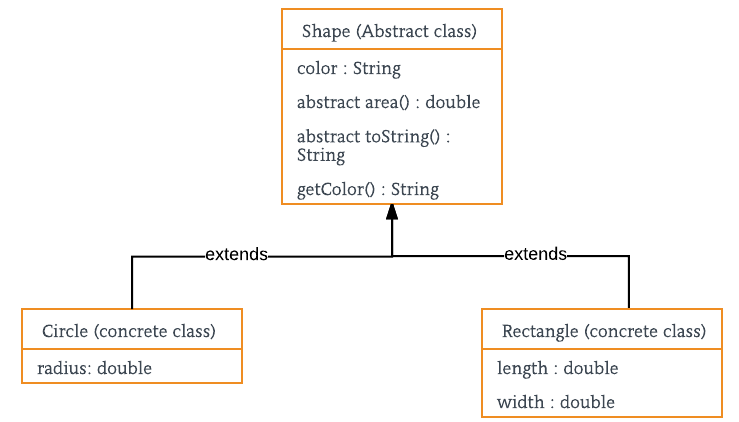
**Abstract classes and Abstract methods :**

1. An abstract class is a class that is declared with [abstract keyword.](https://www.geeksforgeeks.org/abstract-keyword-in-java/)
2. An abstract method is a method that is declared without an implementation.
3. An abstract class may or may not have all abstract methods. Some of them can be concrete methods
4. A method defined abstract must always be redefined in the subclass, thus making [overriding](http://contribute.geeksforgeeks.org/overriding-in-java/) compulsory OR either make subclass itself abstract.
5. Any class that contains one or more abstract methods must also be declared with abstract keyword.
6. There can be no object of an abstract class. That is, an abstract class cannot be directly instantiated with the [*new operator*](https://www.geeksforgeeks.org/new-operator-java/).
7. An abstract class can have parametrized constructors and default constructor is always present in an abstract class.

**When to use abstract classes and abstract methods with an example**

There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes we will want to create a superclass that only defines a generalization form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

Consider a classic “shape” example, perhaps used in a computer-aided design system or game simulation. The base type is “shape” and each shape has a color, size and so on. From this, specific types of shapes are derived(inherited)-circle, square, triangle and so on – each of which may have additional characteristics and behaviors. For example, certain shapes can be flipped. Some behaviors may be different, such as when you want to calculate the area of a shape. The type hierarchy embodies both the similarities and differences between the shapes.



|  |
| --- |
| package techno.study.ch5.abstraction;  // Java program to illustrate the concept of Abstraction abstract class Shape {  String color;   // these are abstract methods  abstract double area();  public abstract String toString();   // abstract class can have constructor  public Shape(String color) {  System.*out*.println("Shape constructor called");  this.color = color;  }   // this is a concrete method  public String getColor() {  return color;  } } class Circle extends Shape {  double radius;   public Circle(String color,double radius) {   // calling Shape constructor  super(color);  System.*out*.println("Circle constructor called");  this.radius = radius;  }   @Override  double area() {  return Math.*PI* \* Math.*pow*(radius, 2);  }   @Override  public String toString() {  return "Circle color is " + super.color +  "and area is : " + area();  }  } class Rectangle extends Shape{   double length;  double width;   public Rectangle(String color,double length,double width) {  // calling Shape constructor  super(color);  System.*out*.println("Rectangle constructor called");  this.length = length;  this.width = width;  }   @Override  double area() {  return length\*width;  }   @Override  public String toString() {  return "Rectangle color is " + super.color +  "and area is : " + area();  }  } class Test {  public static void main(String[] args)  {  Shape s1 = new Circle("Red", 2.2);  Shape s2 = new Rectangle("Yellow", 2, 4);   System.*out*.println(s1.toString());  System.*out*.println(s2.toString());  } } |

Output:

Shape constructor called

Circle constructor called

Shape constructor called

Rectangle constructor called

Circle color is Redand area is : 15.205308443374602

Rectangle color is Yellowand area is : 8.0

**Encapsulation vs Data Abstraction**

1. [Encapsulation](http://contribute.geeksforgeeks.org/encapsulation-in-java/) is data hiding(information hiding) while Abstraction is detail hiding(implementation hiding).
2. While encapsulation groups together data and methods that act upon the data, data abstraction deals with exposing the interface to the user and hiding the details of implementation.

**Advantages of Abstraction**

1. It reduces the complexity of viewing the things.
2. Avoids code duplication and increases reusability.
3. Helps to increase security of an application or program as only important details are provided to the user.

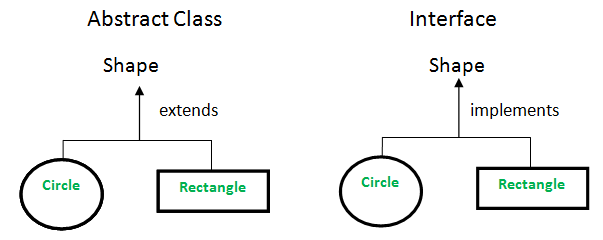
## Difference between Abstract Class and Interface in Java

Prerequisite – [Interface](http://quiz.geeksforgeeks.org/interfaces-in-java/), [Abstract Class](https://www.geeksforgeeks.org/abstract-classes-in-java/)

**Abstraction:**Hiding the internal implementation of the feature and only showing the functionality to the users. i.e. what it works (showing), how it works (hiding). Both [abstract class](https://www.geeksforgeeks.org/abstract-classes-in-java/) and [interface](http://quiz.geeksforgeeks.org/interfaces-in-java/) are used for abstraction.

**Abstract class vs Interface**

1. **Type of methods:** Interface can have only abstract methods. Abstract class can have abstract and non-abstract methods. From Java 8, it can have default and static methods also.
2. **Final Variables:** Variables declared in a Java interface are by default final. An abstract class may contain non-final variables.
3. **Type of variables:**Abstract class can have final, non-final, static and non-static variables. Interface has only static and final variables.
4. **Implementation:** Abstract class can provide the implementation of interface. Interface can’t provide the implementation of abstract class.
5. **Inheritance vs Abstraction:** A Java interface can be implemented using keyword “implements” and abstract class can be extended using keyword “extends”.
6. **Multiple implementation:** An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
7. **Accessibility of Data Members:** Members of a Java interface are public by default. A Java abstract class can have class members like private, protected, etc.



// Java program to illustrate the   
// concept of abstract class   
  
import java.io.\*;  
  
// abstract class   
abstract class Shape {  
 // declare fields   
 String objectName = " ";  
  
 Shape(String name) {  
 this.objectName = name;  
 }  
  
 // declare non-abstract methods   
 // it has default implementation   
 public void moveTo(int x, int y) {  
 System.*out*.println(this.objectName + " " + "has been moved to" + " x = " + x + " and y = " + y);  
 }  
  
 // abstract methods which will be   
 // implemented by its subclass(es)   
 abstract public double area();  
  
 abstract public void draw();  
}  
  
class Rectangle extends Shape {  
  
 int length, width;  
  
 // constructor   
 Rectangle(int length, int width, String name) {  
  
 super(name);  
 this.length = length;  
 this.width = width;  
 }  
  
 @Override  
 public void draw() {  
 System.*out*.println("Rectangle has been drawn ");  
 }  
  
 @Override  
 public double area() {  
 return (double) (length \* width);  
 }  
}  
  
class Circle extends Shape {  
  
 double pi = 3.14;  
 int radius;  
  
 //constructor   
 Circle(int radius, String name) {  
  
 super(name);  
 this.radius = radius;  
 }  
  
 @Override  
 public void draw() {  
  
 System.*out*.println("Circle has been drawn ");  
 }  
  
 @Override  
 public double area() {  
 return (double) ((pi \* radius \* radius) / 2);  
 }  
}  
  
class GFG {  
 public static void main(String[] args) {  
  
 // creating the Object of Rectangle class   
 // and using shape class reference.   
 Shape rect = new Rectangle(2, 3, "Rectangle");  
 System.*out*.println("Area of rectangle: " + rect.area());  
 rect.moveTo(1, 2);  
  
 System.*out*.println(" ");  
  
 // creating the Objects of circle class   
 Shape circle = new Circle(2, "Cicle");  
 System.*out*.println("Area of circle: " + circle.area());  
 circle.moveTo(2, 4);  
  
 }  
}

Output:

Area of rectangle: 6.0

Rectangle has been moved to x = 1 and y = 2

Area of circle: 6.28

Cicle has been moved to x = 2 and y = 4

In you don’t have any common code between rectangle and circle then go with interface.  
See this…..

|  |
| --- |
| // Java program to illustrate the  // concept of interface   import java.io.\*;  interface Shape {  // abstract method   void draw();   double area(); }  class Rectangle implements Shape {  int length, width;   // constructor   Rectangle(int length, int width) {  this.length = length;  this.width = width;  }   @Override  public void draw() {  System.*out*.println("Rectangle has been drawn ");  }   @Override  public double area() {  return (double) (length \* width);  } }  class Circle implements Shape {   double pi = 3.14;  int radius;   //constructor   Circle(int radius) {   this.radius = radius;  }   @Override  public void draw() {  System.*out*.println("Circle has been drawn ");  }   @Override  public double area() {  return (double) ((pi \* radius \* radius) / 2);  } }  class GFG {  public static void main(String[] args) {   // creating the Object of Rectangle class   // and using shape interface reference.   Shape rect = new Rectangle(2, 3);  System.*out*.println("Area of rectangle: " + rect.area());   // creating the Objects of circle class   Shape circle = new Circle(2);  System.*out*.println("Area of circle: " + circle.area());  } } |

output

Area of rectangle: 6.0

Area of circle: 6.28

**When to use what?**

Consider using abstract classes if any of these statements apply to your situation:

* In java application, there are some related classes that need to share some lines of code then you can put these lines of code within abstract class and this abstract class should be extended by all these related classes.
* You can define non-static or non-final field(s) in abstract class, so that via a method you can access and modify the state of Object to which they belong.
* You can expect that the classes that extend an abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).

Consider using interfaces if any of these statements apply to your situation:

* It is total abstraction, All methods declared within an interface must be implemented by the class(es) that implements this interface.
* A class can implement more than one interface. It is called multiple inheritance.
* You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.

## Difference between Abstract Class and Concrete Class in Java

[**Abstract Class**](https://www.geeksforgeeks.org/pure-virtual-functions-and-abstract-classes/)**:** An abstract class is a type of class in Java that is declared by the abstract keyword. An abstract class cannot be instantiated directly, i.e. object of such class cannot be created directly using new keyword. An abstract class can be instantiated either by concrete subclass, or by defining all the abstract method along with the new statement. It may or may not contain abstract method. An abstract method is declared by abstract keyword, such methods cannot have a body. If a class contains abstract method, then it also needs to be abstract.

**Concrete Class:** A concrete class in Java is a type of subclass, which implements all the abstract method of its super abstract class which it extends to. It also has implementations of all methods of interfaces it implements.

**Abstract Class vs Concrete Class**

1. **Modifier:** An abstract class is declared using abstract modifier. Concrete class should not be declared using abstract keyword, on doing so, it will also become abstract class.
2. **Instantiation:** An abstract class cannot be instantiated directly, i.e. object of such class cannot be created directly using new keyword. An abstract class can be instantiated either by concrete subclass, or by defining all the abstract method along with the new statement. A concrete class can be instantiated directly, using a new keyword.

**Example:** Invalid direct instantiation of an abstract class.

abstract class DemoAbstractClass {  
 abstract void display();  
}  
  
public class JavaApplication {  
 public static void main(String[] args) {  
 DemoAbstractClass AC = new DemoAbstractClass();  
 System.*out*.println("Hello");  
 }  
}

**Compile Error:**

prog.java:9: error: DemoAbstractClass is abstract; cannot be instantiated

DemoAbstractClass AC = new DemoAbstractClass();

^

**Example:** Valid instantiation by defining all abstract method of an abstract class.

abstract class DemoAbstractClass {  
 abstract void display();  
}  
  
public class JavaApplication {  
 public static void main(String[] args) {  
 DemoAbstractClass AC = new DemoAbstractClass() {  
 void display() {  
 System.*out*.println("Hi.");  
 }  
 };  
 AC.display();  
 System.*out*.println("How are you?");  
 }  
}

**Output:**

Hi.

How are you?

**Example:** Direct instantiation of concrete using new keyword.

abstract class DemoAbstractClass {  
 abstract void display();  
}  
  
class ConcreteClass extends DemoAbstractClass {  
 void display() {  
 System.*out*.println("Hi.");  
 }  
}  
  
public class JavaApplication {  
 public static void main(String[] args) {  
 ConcreteClass C = new ConcreteClass();  
 C.display();  
 System.*out*.println("How are you?");  
 }  
}

**Output:**

Hi.

How are you?

1. **Abstract methods:** An abstract class may or may not, have an abstract method. A concrete class cannot have an abstract method, because class containing an abstract method must also be abstract.
2. **Final:** An abstract class cannot be **final**, because all its abstract methods must defined in the subclass. A concrete class can be declared as **final**.
3. **Interface:** Interface implementation is not possible with abstract class, however, it is possible with concrete class.

| **ABSTRACT CLASS** | **CONCRETE CLASS** |
| --- | --- |
| An abstract class is declared using abstract modifier. | A concrete class is note declared using abstract modifier. |
| An abstract class cannot be directly instantiated using the new keyword. | A concrete class can be directly instantiated using the new keyword. |
| An abstract class may or may not contain abstract methods. | A concrete class cannot contain an abstract method. |
| An abstract class cannot be declared as final. | A concrete class can be declared as final. |
| Interface implementation is not possible | Interface implementation is possible. |

**Some important points:**

* A concrete class is a subclass of an abstract class, which implements all its abstract method.
* Abstract methods cannot have body.
* Abstract class can have static fields and static method, like other classes.
* An abstract class cannot be declared as final.
* Only abstract class can have abstract methods.
* A private, final, static method cannot be abstract, as it cannot be overridden in a subclass.
* Abstract class cannot have abstract constructors.
* Abstract class cannot have abstract static methods.
* If a class extends an abstract class, then it should define all the abstract methods (override) of the base abstract class. If not, the subclass(the class extending abstract class) must also be defined as abstract class.

# **Polymorphism**

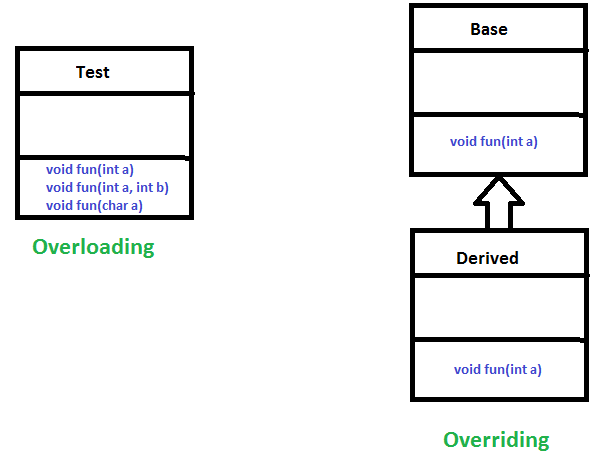
The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

**Real life example of polymorphism:** A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behaviour in different situations. This is called polymorphism.

Polymorphism is considered as one of the important features of Object Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms.

**In Java polymorphism is mainly divided into two types:**

* Compile time Polymorphism
* Runtime Polymorphism

1. **Compile time polymorphism**: It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.  
   [](https://www.geeksforgeeks.org/overloading-in-java/)
   * **Method Overloading**: When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.
     1. Example: By using different types of arguments

|  |
| --- |
| // Java program for Method overloading  class MultiplyFun {   // Method with 2 parameter  static int Multiply(int a, int b) {  return a \* b;  }   // Method with the same name but 3 parameter  static int Multiply(int a, int b, int c) {  return a \* b \* c;  } }  class Main {  public static void main(String[] args) {  System.*out*.println(MultiplyFun.*Multiply*(2, 4));   System.*out*.println(MultiplyFun.*Multiply*(2, 7, 3));  } } |

**Output:**

8

42

* + 1. **Example:** By using different numbers of arguments

// Java program for Method overloading   
  
class MultiplyFun {  
  
 // Method with 2 parameter   
 static int Multiply(int a, int b) {  
 return a \* b;  
 }  
  
 // Method with the same name but 2 double parameter   
 static double Multiply(double a, double b) {  
 return a \* b;  
 }  
}  
  
class Main {  
 public static void main(String[] args) {  
  
 System.*out*.println(MultiplyFun.*Multiply*(2, 4));  
  
 System.*out*.println(MultiplyFun.*Multiply*(5.5, 6.3));  
 }  
}

**Output:**

8

34.65

* + **Operator Overloading**: Java also provide option to overload operators. For example, we can make the operator (‘+’) for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator ‘+’ when placed between integer operands, adds them and when placed between string operands, concatenates them.

In java, Only “+” operator can be overloaded:

* + 1. To add integers
    2. To concatenate strings

**Example**:

// Java program for Operator overloading  
  
class OperatorOVERDDN {  
  
 void operator(String str1, String str2) {  
 String s = str1 + str2;  
 System.*out*.println("Concatinated String - "  
 + s);  
 }  
  
 void operator(int a, int b) {  
 int c = a + b;  
 System.*out*.println("Sum = " + c);  
 }  
}  
  
class Main {  
 public static void main(String[] args) {  
 OperatorOVERDDN obj = new OperatorOVERDDN();  
 obj.operator(2, 3);  
 obj.operator("joe", "now");  
 }  
}

**Output:**

Sum = 5

Concatinated String - joenow

1. [**Runtime polymorphism**](https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/): It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.
   * [**Method overriding**](https://www.geeksforgeeks.org/overriding-in-java/), on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

**Example:**

**Output:**

subclass1

subclass2

### Can we Overload or Override static methods in java ?

Let us first define Overloading and Overriding.

[**Overriding**](https://www.geeksforgeeks.org/overriding-in-java/): Overriding is a feature of OOP languages like Java that is related to run-time polymorphism. A subclass (or derived class) provides a specific implementation of a method in superclass (or base class).  
The implementation to be executed is decided at run-time and decision is made according to the object used for call. Note that signatures of both methods must be same. Refer [Overriding in Java](https://www.geeksforgeeks.org/overriding-in-java/) for details.

[**Overloading**](https://www.geeksforgeeks.org/overloading-in-java/): Overloading is also a feature of OOP languages like Java that is related to compile time (or static) polymorphism. This feature allows different methods to have same name, but different signatures, especially number of input parameters and type of input paramaters. Note that in both C++ and Java, [methods cannot be overloaded according to return type.](https://www.geeksforgeeks.org/g-fact-75/)

**Can we overload static methods?**  
The answer is ‘Yes’. We can have two or more static methods with same name, but differences in input parameters. For example, consider the following Java program.

class OverridingVsOverloading {  
 // filename Test.java   
 public class Test {  
 public static void foo() {  
 System.*out*.println("Test.foo() called ");  
 }  
  
 public static void foo(int a) {  
 System.*out*.println("Test.foo(int) called ");  
 }  
  
 public static void main(String args[]) {  
 Test.*foo*();  
 Test.*foo*(10);  
 }  
 }  
}

Output:

Test.foo() called

Test.foo(int) called

**Can we overload methods that differ only by static keyword?**  
We cannot overload two methods in Java if they differ only by static keyword (number of parameters and types of parameters is same). See following Java program for example. This behaviour is same in C++ (See point 2 of [this](https://www.geeksforgeeks.org/function-overloading-in-c/)).

// filename Test.java   
public class Test {  
 public static void foo() {  
 System.*out*.println("Test.foo() called ");  
 }  
  
 public void foo() { // Compiler Error: cannot redefine foo()   
 System.*out*.println("Test.foo(int) called ");  
 }  
  
 public static void main(String args[]) {  
 Test.foo();  
 }  
}

Output: Compiler Error, cannot redefine foo()

**Can we Override static methods in java?**  
We can declare static methods with same signature in subclass, but it is not considered overriding as there won’t be any run-time polymorphism. Hence the answer is ‘No’.  
If a derived class defines a static method with same signature as a static method in base class, the method in the derived class hides the method in the base class.

/\* Java program to show that if static method is redefined by   
 a derived class, then it is not overriding. \*/  
  
 // Superclass   
 class Base {  
  
 // Static method in base class which will be hidden in subclass   
 public static void display() {  
 System.*out*.println("Static or class method from Base");  
 }  
  
 // Non-static method which will be overridden in derived class   
 public void print() {  
 System.*out*.println("Non-static or Instance method from Base");  
 }  
 }  
  
 // Subclass   
 class Derived extends Base {  
  
 // This method hides display() in Base   
 public static void display() {  
 System.*out*.println("Static or class method from Derived");  
 }  
  
 // This method overrides print() in Base   
 public void print() {  
 System.*out*.println("Non-static or Instance method from Derived");  
 }  
 }  
  
 // Driver class   
 public class Test {  
 public static void main(String args[]) {  
 Base obj1 = new Derived();  
  
 // As per overriding rules this should call to class Derive's static   
 // overridden method. Since static method can not be overridden, it   
 // calls Base's display()   
 obj1.*display*();  
  
 // Here overriding works and Derive's print() is called   
 obj1.print();  
 }  
 }

Output:

Static or class method from Base

Non-static or Instance method from Derived

Following are some important points for method overriding and static methods in Java.  
**1)** For class (or static) methods, the method according to the type of reference is called, not according to the object being referred, which means method call is decided at compile time.

**2)** For instance (or non-static) methods, the method is called according to the type of object being referred, not according to the type of reference, which means method calls is decided at run time.

**3)** An instance method cannot override a static method, and a static method cannot hide an instance method. For example, the following program has two compiler errors.

/\* Java program to show that if static methods are redefined by   
 a derived class, then it is not overriding but hidding. \*/  
  
 // Superclass   
 class Base {  
  
 // Static method in base class which will be hidden in subclass   
 public static void display() {  
 System.*out*.println("Static or class method from Base");  
 }  
  
 // Non-static method which will be overridden in derived class   
 public void print() {  
 System.*out*.println("Non-static or Instance method from Base");  
 }  
 }  
  
 // Subclass   
 class Derived extends Base {  
  
 // Static is removed here (Causes Compiler Error)   
 public void display() {  
 System.*out*.println("Non-static method from Derived");  
 }  
  
 // Static is added here (Causes Compiler Error)   
 public static void print() {  
 System.*out*.println("Static method from Derived");  
 }  
 }

**4)** In a subclass (or Derived Class), we can overload the methods inherited from the superclass. Such overloaded methods neither hide nor override the superclass methods — they are new methods, unique to the subclass.

**References:**  
<http://docs.oracle.com/javase/tutorial/java/IandI/override.html>

### Can we override private methods in Java?

Let us first consider the following Java program as a simple example of Overriding or Runtime Polymorphism.

class Base {  
 public void fun() {  
 System.*out*.println("Base fun");  
 }  
}  
  
class Derived extends Base {  
 public void fun() { // overrides the Base's fun()   
 System.*out*.println("Derived fun");  
 }  
 public static void main(String[] args) {  
 Base obj = new Derived();  
 obj.fun();  
 }  
}

The program prints “Derived fun”.  
The Base class reference ‘obj’ refers to a derived class object (see expression “Base obj = new Derived()”). When fun() is called on obj, the call is made according to the type of referred object, not according to the reference.

**Is Overiding possible with private methods?**  
Predict the output of following program.

|  |
| --- |
| class Base {  private void fun() {  System.*out*.println("Base fun");  } }  class Derived extends Base {  private void fun() {  System.*out*.println("Derived fun");  }  public static void main(String[] args) {  Base obj = new Derived();  obj.fun();  } } |

We get compiler error “fun() has private access in Base” (See [this](http://ideone.com/arKk3c)). So the compiler tries to call base class function, not derived class, means fun() is not overridden.

**An inner class can access private members of its outer class. What if we extend an inner class and create fun() in the inner class?**  
An Inner classes can access private members of its outer class, for example in the following program, fun() of Inner accesses private data member msg which is fine by the compiler.

/\* Java program to demonstrate whether we can override private method   
 of outer class inside its inner class \*/  
class Outer {  
 private String msg = "GeeksforGeeks";  
 private void fun() {  
 System.*out*.println("Outer fun()");  
 }  
  
 class Inner extends Outer {  
 private void fun() {  
 System.*out*.println("Accessing Private Member of Outer: " + msg);  
 }  
 }  
  
 public static void main(String args[]) {  
  
 // In order to create instance of Inner class, we need an Outer   
 // class instance. So, first create Outer class instance and then   
 // inner class instance.   
 Outer o = new Outer();  
 Inner i = o.new Inner();  
  
 // This will call Inner's fun, the purpose of this call is to   
 // show that private members of Outer can be accessed in Inner.   
 i.fun();  
  
 // o.fun() calls Outer's fun (No run-time polymorphism).   
 o = i;  
 o.fun();  
 }  
}

Output:

Accessing Private Member of Outer: GeeksforGeeks

Outer fun()

In the above program, we created an outer class and an inner class. We extended Inner from Outer and created a method fun() in both Outer and Inner. If we observe our output, then it is clear that the method fun() has not been overriden. It is so because ***private methods are bonded during compile time and it is the type of the reference variable – not the type of object that it refers to – that determines what method to be called.***. As a side note, private methods may be performance-wise better (compared to non-private and non-final methods) due to static binding.

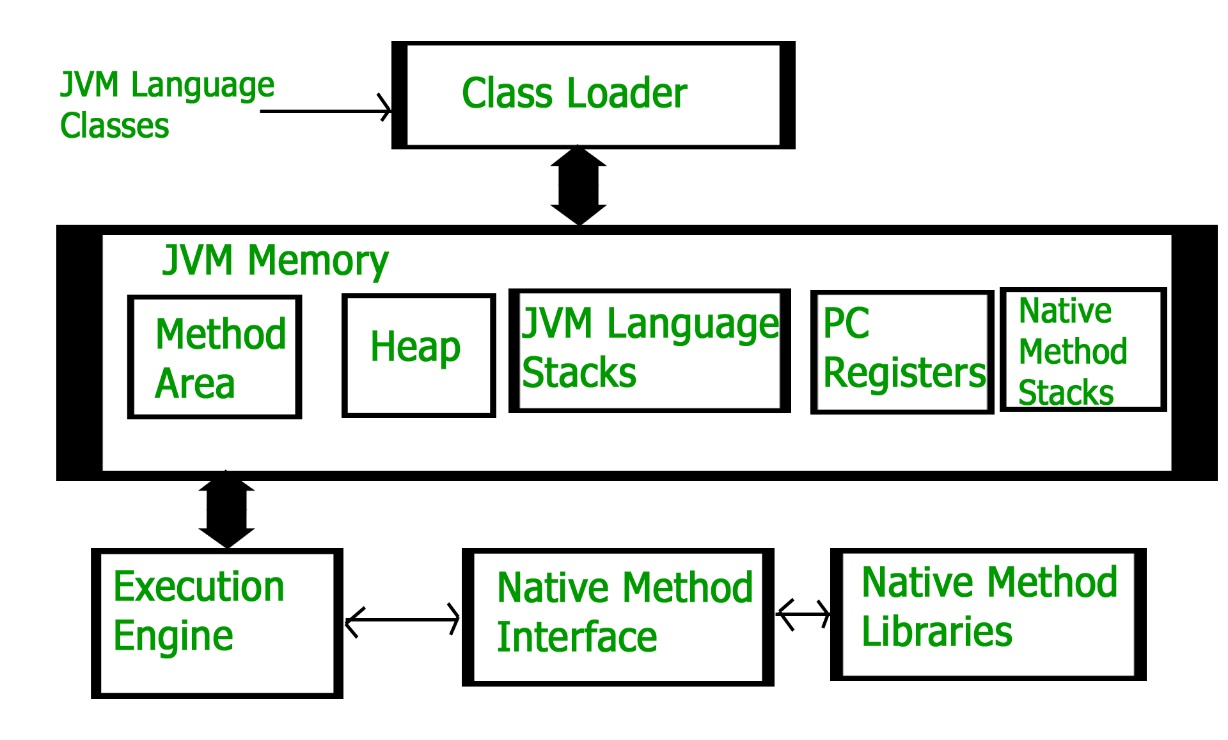
# **Chapter8**

## How JVM Works – JVM Architecture?

JVM(Java Virtual Machine) acts as a run-time engine to run Java applications. JVM is the one that actually calls the **main** method present in a java code. JVM is a part of JRE(Java Runtime Environment).

Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java enabled system without any adjustment. This is all possible because of JVM.

When we compile a .java file, .class files(contains byte-code) with the same class names present in .java file are generated by the Java compiler. This .class file goes into various steps when we run it. These steps together describe the whole JVM.

[](https://media.geeksforgeeks.org/wp-content/uploads/jvm-3.jpg)

**Class Loader Subsystem**  
It is mainly responsible for three activities.

* Loading
* Linking
* Initialization

**Loading :** The Class loader reads the .class file, generate the corresponding binary data and save it in method area. For each .class file, JVM stores following information in method area.

* Fully qualified name of the loaded class and its immediate parent class.
* Whether .class file is related to Class or Interface or Enum
* Modifier, Variables and Method information etc.

After loading .class file, JVM creates an object of type Class to represent this file in the heap memory. Please note that this object is of type Class predefined in java.lang package. This Class object can be used by the programmer for getting class level information like name of class, parent name, methods and variable information etc. To get this object reference we can use getClass() method of [Object](https://www.geeksforgeeks.org/object-class-in-java/) class.

// A Java program to demonstrate working of a Class type   
// object created by JVM to represent .class file in   
// memory.   
import java.lang.reflect.Field;  
 import java.lang.reflect.Method;  
  
// Java code to demonstrate use of Class object   
// created by JVM   
public class Test  
{  
 public static void main(String[] args)  
 {  
 Student s1 = new Student();  
  
 // Getting hold of Class object created   
 // by JVM.   
 Class c1 = s1.getClass();  
  
 // Printing type of object using c1.   
 System.*out*.println(c1.getName());  
  
 // getting all methods in an array   
 Method m[] = c1.getDeclaredMethods();  
 for (Method method : m)  
 System.*out*.println(method.getName());  
  
 // getting all fields in an array   
 Field f[] = c1.getDeclaredFields();  
 for (Field field : f)  
 System.*out*.println(field.getName());  
 }  
}  
  
// A sample class whose information is fetched above using   
// its Class object.   
class Student  
{  
 private String name;  
 private int roll\_No;  
  
 public String getName() { return name; }  
 public void setName(String name) { this.name = name; }  
 public int getRoll\_no() { return roll\_No; }  
 public void setRoll\_no(int roll\_no) {  
 this.roll\_No = roll\_no;  
 }  
}

Output:

Student

getName

setName

getRoll\_no

setRoll\_no

name

roll\_No

**Note :** For every loaded .class file, only **one** object of Class is created.

Student s2 = new Student();

// c2 will point to same object where

// c1 is pointing

Class c2 = s2.getClass();

System.out.println(c1==c2); // true

**Linking :** Performs verification, preparation, and (optionally) resolution.

* Verification : It ensures the correctness of .class file i.e. it check whether this file is properly formatted and generated by valid compiler or not. If verification fails, we get run-time exception java.lang.VerifyError.
* Preparation : JVM allocates memory for class variables and initializing the memory to default values.
* Resolution : It is the process of replacing symbolic references from the type with direct references. It is done by searching into method area to locate the referenced entity.

**Initialization :** In this phase, all static variables are assigned with their values defined in the code and static block(if any). This is executed from top to bottom in a class and from parent to child in class hierarchy.  
In general, there are three class loaders :

* Bootstrap class loader : Every JVM implementation must have a bootstrap class loader, capable of loading trusted classes. It loads core java API classes present in JAVA\_HOME/jre/lib directory. This path is popularly known as bootstrap path. It is implemented in native languages like C, C++.
* Extension class loader : It is child of bootstrap class loader. It loads the classes present in the extensions directories JAVA\_HOME/jre/lib/ext(Extension path) or any other directory specified by the java.ext.dirs system property. It is implemented in java by the sun.misc.Launcher$ExtClassLoader class.
* System/Application class loader : It is child of extension class loader. It is responsible to load classes from application class path. It internally uses Environment Variable which mapped to java.class.path. It is also implemented in Java by the sun.misc.Launcher$AppClassLoader class.

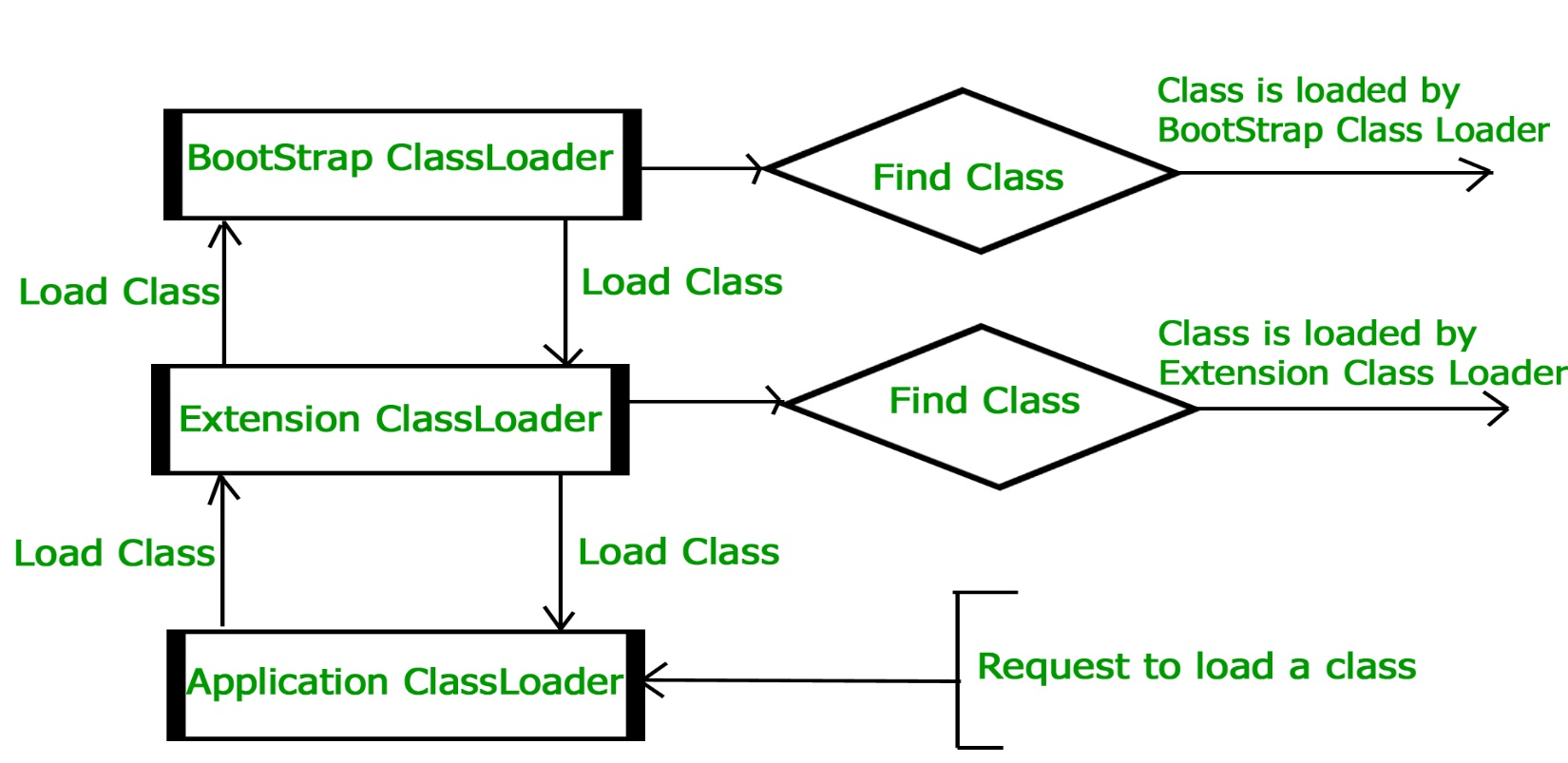
// Java code to demonstrate Class Loader subsystem   
public class Test  
{  
 public static void main(String[] args)  
 {  
 // String class is loaded by bootstrap loader, and   
 // bootstrap loader is not Java object, hence null   
 System.*out*.println(String.class.getClassLoader());  
  
 // Test class is loaded by Application loader   
 System.*out*.println(Test.class.getClassLoader());  
 }  
}

Output:

null

[sun.misc.Launcher$AppClassLoader@73d16e93](mailto:sun.misc.Launcher$AppClassLoader@73d16e93)

**Note :**JVM follow Delegation-Hierarchy principle to load classes. System class loader delegate load request to extension class loader and extension class loader delegate request to boot-strap class loader. If class found in boot-strap path, class is loaded otherwise request again transfers to extension class loader and then to system class loader. At last if system class loader fails to load class, then we get run-time exception *java.lang.ClassNotFoundException*.

[](https://media.geeksforgeeks.org/wp-content/uploads/jvmclassloader.jpg)

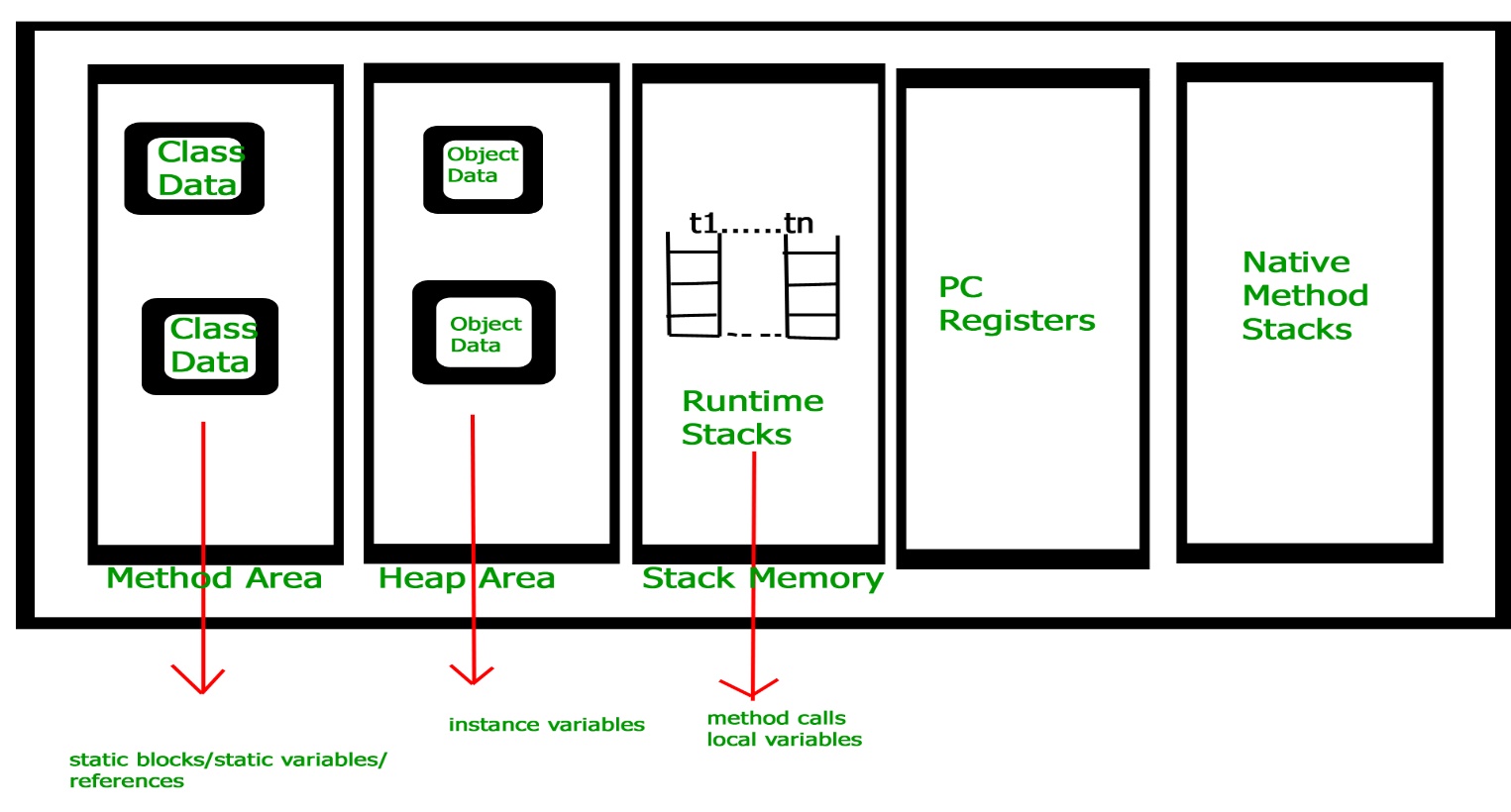
**JVM Memory**  
**Method area :**In method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.

**Heap area :**Information of all objects is stored in heap area. There is also one Heap Area per JVM. It is also a shared resource.

**Stack area :**For every thread, JVM create one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which store methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminate, it’s run-time stack will be destroyed by JVM. It is not a shared resource.

**PC Registers :**Store address of current execution instruction of a thread. Obviously each thread has separate PC Registers.

**Native method stacks :**For every thread, separate native stack is created. It stores native method information.

[](https://media.geeksforgeeks.org/wp-content/uploads/jvm-memory-2.jpg)

**Execution Engine**  
Execution engine execute the *.class* (bytecode). It reads the byte-code line by line, use data and information present in various memory area and execute instructions. It can be classified in three parts :-

* *Interpreter* : It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.
* *Just-In-Time Compiler(JIT)* : It is used to increase efficiency of interpreter.It compiles the entire bytecode and changes it to native code so whenever interpreter see repeated method calls,JIT provide direct native code for that part so re-interpretation is not required,thus efficiency is improved.
* *Garbage Collector* : It destroy un-referenced objects.For more on Garbage Collector,refer [Garbage Collector](https://www.geeksforgeeks.org/garbage-collection-java/).

**Java Native Interface (JNI) :**  
It is an interface which interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

**Native Method Libraries :**  
It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine.

**Class Loaders in Java**

## 1. Introduction to Class Loaders

Class loaders are responsible for**loading Java classes during runtime dynamically to the JVM** (Java Virtual Machine). Also, they are part of the JRE (Java Runtime Environment). Hence, the JVM doesn't need to know about the underlying files or file systems in order to run Java programs thanks to class loaders.

Also, these Java classes aren't loaded into memory all at once, but when required by an application. This is where class loaders come into the picture. They are responsible for loading classes into memory.

In this tutorial, we're going to talk about different types of built-in class loaders, how they work and an introduction to our own custom implementation.

## 2. Types of Built-in Class Loaders

Let's start by learning how different classes are loaded using various class loaders using a simple example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public void printClassLoaders() throws ClassNotFoundException {        System.out.println("Classloader of this class:"          + PrintClassLoader.class.getClassLoader());        System.out.println("Classloader of Logging:"          + Logging.class.getClassLoader());        System.out.println("Classloader of ArrayList:"          + ArrayList.class.getClassLoader());  } |

When executed the above method prints:

|  |  |
| --- | --- |
| 1  2  3 | Class loader of this class:sun.misc.Launcher$AppClassLoader@18b4aac2  Class loader of Logging:sun.misc.Launcher$ExtClassLoader@3caeaf62  Class loader of ArrayList:null |

As we can see, there are three different class loaders here; application, extension, and bootstrap (displayed as null).

The application class loader loads the class where the example method is contained. **An application or system class loader loads our own files in the classpath.**

Next, the extension one loads the Logging class.**Extension class loaders load classes that are an extension of the standard core Java classes.**

Finally, the bootstrap one loads the ArrayList class. **A bootstrap or primordial class loader is the parent of all the others.**

However, we can see that the last out, for the ArrayList it displays null in the output. **This is because the bootstrap class loader is written in native code, not Java – so it doesn't show up as a Java class.**Due to this reason, the behavior of the bootstrap class loader will differ across JVMs.

Let's now discuss more in detail about each of these class loaders.

### 2.1. Bootstrap Class Loader

Java classes are loaded by an instance of java.lang.ClassLoader. However, class loaders are classes themselves. Hence, the question is, who loads the java.lang.ClassLoader itself?

This is where the bootstrap or primordial class loader comes into the picture.

It's mainly responsible for loading JDK internal classes, typically rt.jar and other core libraries located in $JAVA\_HOME/jre/lib directory. Additionally, **Bootstrap class loader serves as a parent of all the other ClassLoader instances**.

**This bootstrap class loader is part of the core JVM and is written in native code** as pointed out in the above example. Different platforms might have different implementations of this particular class loader.

### 2.2. Extension Class Loader

The **extension class loader is a child of the bootstrap class loader and takes care of loading the extensions of the standard core Java classes** so that it's available to all applications running on the platform.

Extension class loader loads from the JDK extensions directory, usually $JAVA\_HOME/lib/ext directory or any other directory mentioned in the java.ext.dirs system property.

### 2.3. System Class Loader

The system or application class loader, on the other hand, takes care of loading all the application level classes into the JVM. **It loads files found in the classpath environment variable, -classpath or -cp command line option**. Also, it's a child of Extensions classloader.

## 3. How do Class Loaders Work?

Class loaders are part of the Java Runtime Environment. When the JVM requests a class, the class loader tries to locate the class and load the class definition into the runtime using the fully qualified class name.

The **java.lang.ClassLoader.loadClass() method is responsible for loading the class definition into runtime**. It tries to load the class based on a fully qualified name.

If the class isn't already loaded, it delegates the request to the parent class loader. This process happens recursively.

Eventually, if the parent class loader doesn’t find the class, then the child class will call java.net.URLClassLoader.findClass() method to look for classes in the file system itself.

If the last child class loader isn't able to load the class either, it throws [java.lang.NoClassDefFoundError or java.lang.ClassNotFoundException.](https://www.baeldung.com/java-classnotfoundexception-and-noclassdeffounderror)

Let's look at an example of output when ClassNotFoundException is thrown.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | java.lang.ClassNotFoundException: com.baeldung.classloader.SampleClassLoader      at java.net.URLClassLoader.findClass(URLClassLoader.java:381)      at java.lang.ClassLoader.loadClass(ClassLoader.java:424)      at java.lang.ClassLoader.loadClass(ClassLoader.java:357)      at java.lang.Class.forName0(Native Method)      at java.lang.Class.forName(Class.java:348) |

If we go through the sequence of events right from calling java.lang.Class.forName(), we can understand that it first tries to load the class through parent class loader and then java.net.URLClassLoader.findClass() to look for the class itself.

When it still doesn't find the class, it throws a ClassNotFoundException.

There are three important features of class loaders.

### 3.1. Delegation Model

Class loaders follow the delegation model where **on request to find a class or resource, a ClassLoader instance will delegate the search of the class or resource to the parent class loader**.

Let's say we have a request to load an application class into the JVM. The system class loader first delegates the loading of that class to its parent extension class loader which in turn delegates it to the bootstrap class loader.

Only if the bootstrap and then the extension class loader is unsuccessful in loading the class, the system class loader tries to load the class itself.

### 3.2. Unique Classes

As a consequence of the delegation model, it's easy to ensure **unique classes as we always try to delegate upwards**.

If the parent class loader isn't able to find the class, only then the current instance would attempt to do so itself.

### 3.3. Visibility

In addition, **children class loaders are visible to classes loaded by its parent class loaders**.

For instance, classes loaded by the system class loader have visibility into classes loaded by the extension and Bootstrap class loaders but not vice-versa.

To illustrate this, if Class A is loaded by an application class loader and class B is loaded by the extensions class loader, then both A and B classes are visible as far as other classes loaded by Application class loader are concerned.

Class B, nonetheless, is the only class visible as far as other classes loaded by the extension class loader are concerned.

## 4. Custom ClassLoader

The built-in class loader would suffice in most of the cases where the files are already in the file system.

However, in scenarios where we need to load classes out of the local hard drive or a network, we may need to make use of custom class loaders.

In this section, we'll cover some other uses cases for custom class loaders and we'll demonstrate how to create one.

### 4.1. Custom Class Loaders Use-Cases

Custom class loaders are helpful for more than just loading the class during runtime, a few use cases might include:

1. Helping in modifying the existing bytecode, e.g. weaving agents
2. Creating classes dynamically suited to the user's needs. e.g in JDBC, switching between different driver implementations is done through dynamic class loading.
3. Implementing a class versioning mechanism while loading different bytecodes for classes with same names and packages. This can be done either through URL class loader (load jars via URLs) or custom class loaders.

There are more concrete examples where custom class loaders might come in handy.

**Browsers, for instance, use a custom class loader to load executable content from a website.** A browser can load applets from different web pages using separate class loaders. The applet viewer which is used to run applets contains a ClassLoader that accesses a website on a remote server instead of looking in the local file system.

And then loads the raw bytecode files via HTTP, and turns them into classes inside the JVM. Even if these**applets have the same name, they are considered as different components if loaded by different class loaders**.

Now that we understand why custom class loaders are relevant, let's implement a subclass of ClassLoader to extend and summarise the functionality of how the JVM loads classes.

### 4.2. Creating our Custom Class Loader

For illustration purposes, let's say we need to load classes from a file using a custom class loader.

**We need to extend the ClassLoader class and override the findClass() method:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | public class CustomClassLoader extends ClassLoader {        @Override      public Class findClass(String name) throws ClassNotFoundException {          byte[] b = loadClassFromFile(name);          return defineClass(name, b, 0, b.length);      }        private byte[] loadClassFromFile(String fileName)  {          InputStream inputStream = getClass().getClassLoader().getResourceAsStream(                  fileName.replace('.', File.separatorChar) + ".class");          byte[] buffer;          ByteArrayOutputStream byteStream = new ByteArrayOutputStream();          int nextValue = 0;          try {              while ( (nextValue = inputStream.read()) != -1 ) {                  byteStream.write(nextValue);              }          } catch (IOException e) {              e.printStackTrace();          }          buffer = byteStream.toByteArray();          return buffer;      }  } |

In the above example, we defined a custom class loader that extends the default class loader and loads a byte array from the specified file.

### 5. Understanding java.lang.ClassLoader

Let's discuss a few essential methods from the java.lang.ClassLoader class to get a clearer picture of how it works.

### 5.1. The loadClass() Method

|  |  |
| --- | --- |
| 1 | public Class<?> loadClass(String name, boolean resolve) throws ClassNotFoundException { |

This method is responsible for loading the class given a name parameter. The name parameter refers to the fully qualified class name.

The Java Virtual Machine invokes loadClass() method to resolve class references setting resolve to true. However, it isn't always necessary to resolve a class. **If we only need to determine if the class exists or not, then resolve parameter is set to false.**

This method serves as an entry point for the class loader.

We can try to understand the internal working of the loadClass() method from the source code of java.lang.ClassLoader:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | protected Class<?> loadClass(String name, boolean resolve)    throws ClassNotFoundException {        synchronized (getClassLoadingLock(name)) {          // First, check if the class has already been loaded          Class<?> c = findLoadedClass(name);          if (c == null) {              long t0 = System.nanoTime();                  try {                      if (parent != null) {                          c = parent.loadClass(name, false);                      } else {                          c = findBootstrapClassOrNull(name);                      }                  } catch (ClassNotFoundException e) {                      // ClassNotFoundException thrown if class not found                      // from the non-null parent class loader                  }                    if (c == null) {                      // If still not found, then invoke findClass in order                      // to find the class.                      c = findClass(name);                  }              }              if (resolve) {                  resolveClass(c);              }              return c;          }      } |

The default implementation of the method searches for classes in the following order:

1. Invokes the findLoadedClass(String) method to see if the class is already loaded.
2. Invokes the loadClass(String) method on the parent class loader.
3. Invoke the findClass(String) method to find the class.

### 5.2. The defineClass() Method

|  |  |
| --- | --- |
| 1  2 | protected final Class<?> defineClass(    String name, byte[] b, int off, int len) throws ClassFormatError |

This method is responsible for the conversion of an array of bytes into an instance of a class. And before we use the class, we need to resolve it.

In case data didn't contain a valid class, it throws a ClassFormatError.

Also, we can't override this method since it's marked as final.

### 5.3. The findClass() Method

|  |  |
| --- | --- |
| 1  2 | protected Class<?> findClass(    String name) throws ClassNotFoundException |

This method finds the class with the fully qualified name as a parameter. We need to override this method in custom class loader implementations that follow the delegation model for loading classes.

Also, loadClass() invokes this method if the parent class loader couldn't find the requested class.

The default implementation throws a ClassNotFoundException if no parent of the class loader finds the class.

### 5.4. The getParent() Method

|  |  |
| --- | --- |
| 1 | public final ClassLoader getParent() |

This method returns the parent class loader for delegation.

Some implementations like the one seen before in Section 2. use null to represent the bootstrap class loader.

### 5.5. The getResource() Method

|  |  |
| --- | --- |
| 1 | public URL getResource(String name) |

This method tries to find a resource with the given name.

It will first delegate to the parent class loader for the resource. **If the parent is null, the path of the class loader built into the virtual machine is searched.**

If that fails, then the method will invoke findResource(String) to find the resource. The resource name specified as an input can be relative or absolute to the classpath.

It returns an URL object for reading the resource, or null if the resource could not be found or if the invoker doesn't have adequate privileges to return the resource.

It's important to note that Java loads resources from the classpath.

Finally, **resource loading in Java is considered location-independent** as it doesn't matter where the code is running as long as the environment is set to find the resources.

## 6. Context Classloaders

In general, context class loaders provide an alternative method to the class-loading delegation scheme introduced in J2SE.

Like we've learned before, **classloaders in a JVM follow a hierarchical model such that every class loader has a single parent with the exception of the bootstrap class loader.**

However, sometimes when JVM core classes need to dynamically load classes or resources provided by application developers, we might encounter a problem.

For example, in JNDI the core functionality is implemented by bootstrap classes in rt.jar. But these JNDI classes may load JNDI providers implemented by independent vendors (deployed in the application classpath). This scenario calls for the bootstrap class loader (parent class loader) to load a class visible to application loader (child class loader).

**J2SE delegation doesn't work here and to get around this problem, we need to find alternative ways of class loading. And it can be achieved using thread context loaders.**

The java.lang.Thread class has a method **getContextClassLoader() that returns the ContextClassLoader for the particular thread**. The ContextClassLoader is provided by the creator of the thread when loading resources and classes.

If the value isn't set, then it defaults to the class loader context of the parent thread.

# **Understanding Memory Leaks in Java**

## ****1. Introduction****

One of the core benefits of Java is the automated memory management with the help of the built-in Garbage Collector (or **GC** for short). The GC implicitly takes care of allocating and freeing up memory and thus is capable of handling the majority of the memory leak issues.

While the GC effectively handles a good portion of memory, it doesn't guarantee a foolproof solution to memory leaking. The GC is pretty smart, but not flawless. Memory leaks can still sneak up even in applications of a conscientious developer.

There still might be situations where the application generates a substantial number of superfluous objects, thus depleting crucial memory resources, sometimes resulting in the whole application's failure.

Memory leaks are a genuine problem in Java. In this tutorial, we'll see **what the potential causes of memory leaks are, how to recognize them at runtime, and how to deal with them in our application**.

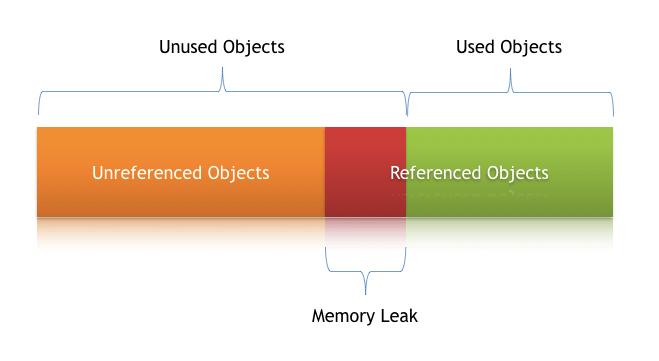
## ****2. What Is a Memory Leak****

A Memory Leak is a situation **when there are objects present in the heap that are no longer used, but the garbage collector is unable to remove them from memory** and, thus they are unnecessarily maintained.

A memory leak is bad because it **blocks memory resources and degrades system performance over time**. And if not dealt with, the application will eventually exhaust its resources, finally terminating with a fatal java.lang.OutOfMemoryError.

There are two different types of objects that reside in Heap memory — referenced and unreferenced. Referenced objects are those who have still active references within the application whereas unreferenced objects don't have any active references.

**The garbage collector removes unreferenced objects periodically, but it never collects the objects that are still being referenced.** This is where memory leaks can occur:

[](https://www.baeldung.com/wp-content/uploads/2018/11/Memory-_Leak-_In-_Java.png)

**Symptoms of a Memory Leak**

* Severe performance degradation when the application is continuously running for a long time
* OutOfMemoryError heap error in the application
* Spontaneous and strange application crashes
* The application is occasionally running out of connection objects

Let's have a closer look at some of these scenarios and how to deal with them.

## ****3. Types of Memory Leaks in Java****

In any application, memory leaks can occur for numerous reasons. In this section, we'll discuss the most common ones.

### ****3.1. Memory Leak Through****static****Fields****

The first scenario that can cause a potential memory leak is heavy use of static variables.

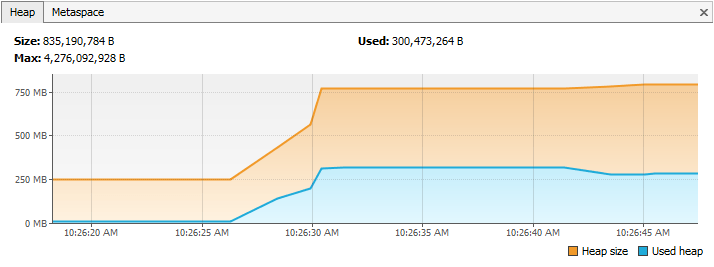
In Java, **static fields have a life that usually matches the entire lifetime of the running application** (unless ClassLoader becomes eligible for garbage collection).

Let's create a simple Java program that populates a static List:

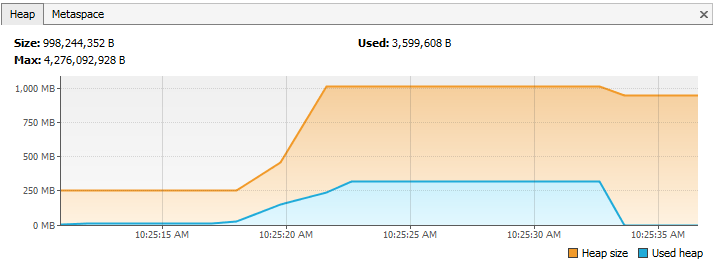
public class StaticTest {  
 public static List<Double> *list* = new ArrayList<>();  
  
 public void populateList() {  
 for (int i = 0; i < 10000000; i++) {  
 *list*.add(Math.*random*());  
 }  
 Log.info("Debug Point 2");  
 }  
  
 public static void main(String[] args) {  
 Log.info("Debug Point 1");  
 new StaticTest().populateList();  
 Log.info("Debug Point 3");  
 }  
}

Now if we analyze the Heap memory during this program execution, then we'll see that between debug points 1 and 2, as expected, the heap memory increased.

But when we leave the populateList() method at the debug point 3, **the heap memory isn't yet garbage collected** as we can see in this VisualVM response:

[](https://www.baeldung.com/wp-content/uploads/2018/11/memory-with-static.png)

However, in the above program, in line number 2, if we just drop the keyword static, then it will bring a drastic change to the memory usage, this Visual VM response shows:

[](https://www.baeldung.com/wp-content/uploads/2018/11/memory-without-static.png)

The first part until the debug point is almost the same as what we obtained in the case of static. But this time after we leave the populateList() method, **all the memory of the list is garbage collected because we don't have any reference to it**.

Hence we need to pay very close attention to our usage of static variables. If collections or large objects are declared as static, then they remain in the memory throughout the lifetime of the application, thus blocking the vital memory that could otherwise be used elsewhere.

**How to Prevent It?**

* Minimize the use of static variables
* When using singletons, rely upon an implementation that lazily loads the object instead of eagerly loading

### ****3.2. Through Unclosed Resources****

Whenever we make a new connection or open a stream, the JVM allocates memory for these resources. A few examples include database connections, input streams, and session objects.

Forgetting to close these resources can block the memory, thus keeping them out of the reach of GC. This can even happen in case of an exception that prevents the program execution from reaching the statement that's handling the code to close these resources.

In either case, **the open connection left from resources consumes memory**, and if we don't deal with them, they can deteriorate performance and may even result in OutOfMemoryError.

**How to Prevent It?**

* Always use finally block to close resources
* The code (even in the finally block) that closes the resources should not itself have any exceptions
* When using Java 7+, we can make use of try-with-resources block

### ****3.3. Improper****equals()****and**** hashCode()****Implementations****

When defining new classes, a very common oversight is not writing proper overridden methods for equals() and hashCode() methods.

HashSet and HashMap use these methods in many operations, and if they're not overridden correctly, then they can become a source for potential memory leak problems.

Let's take an example of a trivial Person class and use it as a key in a HashMap:

public class Person {  
 public String name;  
  
 public Person(String name) {  
 this.name = name;  
 }  
}

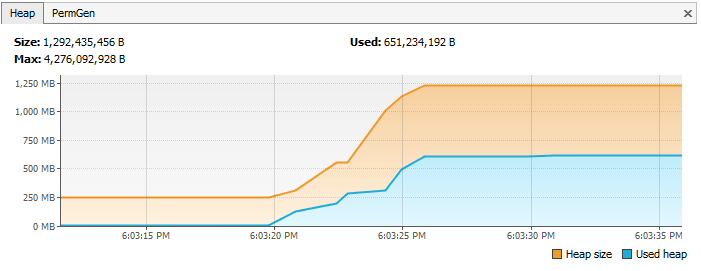
Now we'll insert duplicate Person objects into a Map that uses this key.

Remember that a Map cannot contain duplicate keys:

@Test  
public void givenMap\_whenEqualsAndHashCodeNotOverridden\_thenMemoryLeak() {  
 Map<Person, Integer> map = new HashMap<>();  
 for (int i = 0; i < 100; i++) {  
 map.put(new Person("jon"), 1);  
 }  
 Assert.assertFalse(map.size() == 1);  
}

Here we're using Person as a key. Since Map doesn't allow duplicate keys, the numerous duplicate Person objects that we've inserted as a key shouldn't increase the memory.

But **since we haven't defined proper equals() method, the duplicate objects pile up and increase the memory**, that's why we see more than one object in the memory. The Heap Memory in VisualVM for this looks like:

[](https://www.baeldung.com/wp-content/uploads/2018/11/Before_implementing_equals_and_hashcode.png)

However, **if we had overridden the equals() and hashCode() methods properly, then there would only exist one Person object in this *Map***.

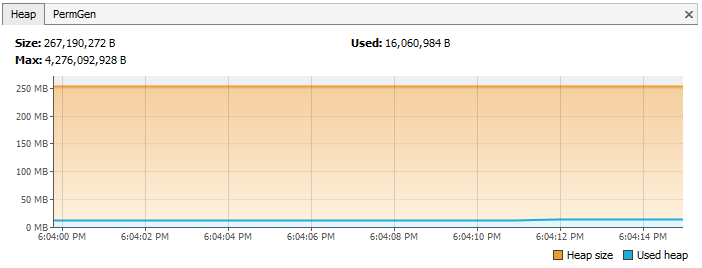
Let's take a look at proper implementations of equals() and hashCode() for our Person class:

public class Person {  
 public String name;  
  
 public Person(String name) {  
 this.name = name;  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (o == this) return true;  
 if (!(o instanceof Person)) {  
 return false;  
 }  
 Person person = (Person) o;  
 return person.name.equals(name);  
 }  
  
 @Override  
 public int hashCode() {  
 int result = 17;  
 result = 31 \* result + name.hashCode();  
 return result;  
 }  
}

And in this case, the following assertions would be true:

@Test  
public void givenMap\_whenEqualsAndHashCodeNotOverridden\_thenMemoryLeak() {  
 Map<Person, Integer> map = new HashMap<>();  
 for (int i = 0; i < 2; i++) {  
 map.put(new Person("jon"), 1);  
 }  
 Assert.assertTrue(map.size() == 1);  
}

After properly overriding equals() and hashCode(), the Heap Memory for the same program looks like:

[](https://www.baeldung.com/wp-content/uploads/2018/11/Afterimplementing_equals_and_hashcode.png)

Another example is of using an ORM tool like Hibernate, which uses equals() and hashCode() methods to analyze the objects and saves them in the cache.

**The chances of memory leak are quite high if these methods are not overridden** because Hibernate then wouldn't be able to compare objects and would fill its cache with duplicate objects.

**How to Prevent It?**

* As a rule of thumb, when defining new entities, always override equals() and hashCode() methods
* It's not just enough to override, but these methods must be overridden in an optimal way as well

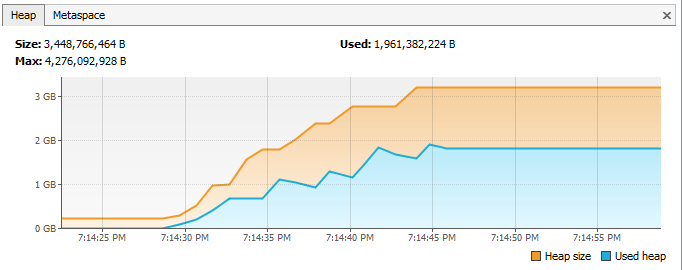
For more information, visit our tutorials [Generate equals() and hashCode() with Eclipse](https://www.baeldung.com/java-eclipse-equals-and-hashcode) and [Guide to hashCode() in Java](https://www.baeldung.com/java-hashcode).

### ****3.4. Inner Classes That Reference Outer Classes****

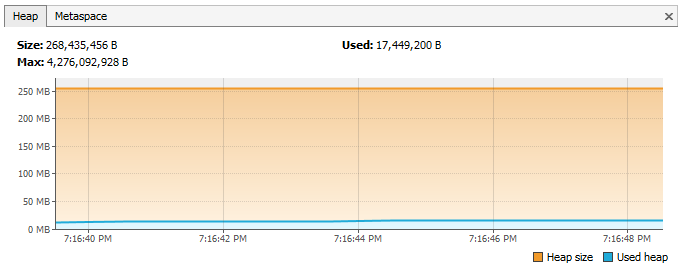
This happens in the case of non-static inner classes (anonymous classes). For initialization, these inner classes always require an instance of the enclosing class.

Every non-static Inner Class has, by default, an implicit reference to its containing class. If we use this inner class' object in our application, then **even after our containing class' object goes out of scope, it will not be garbage collected**.

Consider a class that holds the reference to lots of bulky objects and has a non-static inner class. Now when we create an object of just the inner class, the memory model looks like:

[](https://www.baeldung.com/wp-content/uploads/2018/11/Inner_Classes_That_Reference_Outer_Classes.png)

However, if we just declare the inner class as static, then the same memory model looks like this:

[](https://www.baeldung.com/wp-content/uploads/2018/11/Static_Classes_That_Reference_Outer_Classes.png)

This happens because the inner class object implicitly holds a reference to the outer class object, thereby making it an invalid candidate for garbage collection. The same happens in the case of anonymous classes.

**How to Prevent It?**

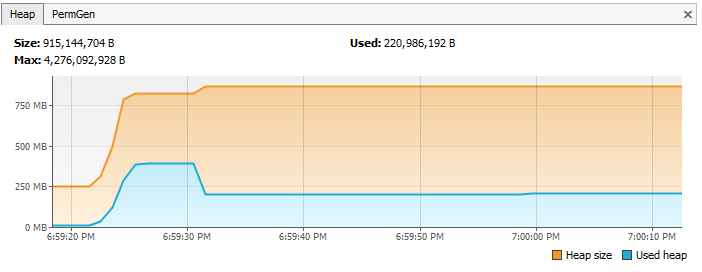
* If the inner class doesn't need access to the containing class members, consider turning it into a static class

### ****3.5. Through****finalize()****Methods****

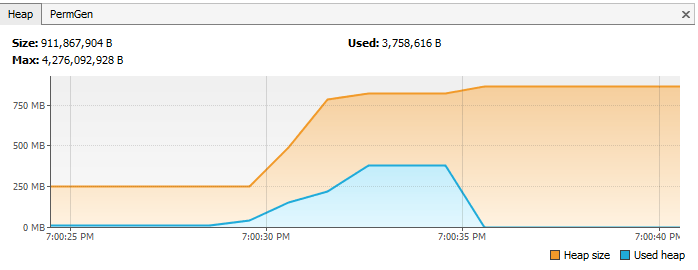
Use of finalizers is yet another source of potential memory leak issues. Whenever a class' finalize() method is overridden, then **objects of that class aren't instantly garbage collected.** Instead, the GC queues them for finalization, which occurs at a later point in time.

Additionally, if the code written in finalize() method is not optimal and if the finalizer queue cannot keep up with the Java garbage collector, then sooner or later, our application is destined to meet an OutOfMemoryError.

To demonstrate this, let's consider that we have a class for which we have overridden the finalize() method and that the method takes a little bit of time to execute. When a large number of objects of this class gets garbage collected, then in VisualVM, it looks like:

[](https://www.baeldung.com/wp-content/uploads/2018/11/Finalize_method_overridden.png)

However, if we just remove the overridden finalize() method, then the same program gives the following response:

[](https://www.baeldung.com/wp-content/uploads/2018/11/Finalize_method_not_overridden.png)

**How to Prevent It?**

* We should always avoid finalizers

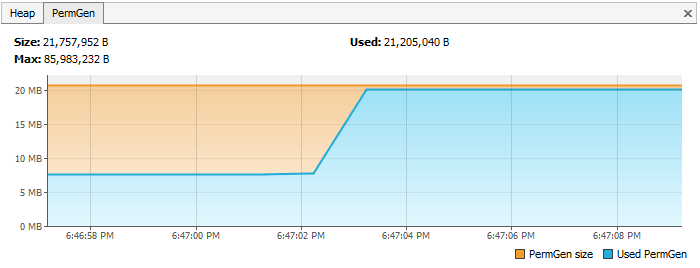
For more detail about finalize(), read section 3 (Avoiding Finalizers) in our [Guide to the finalize Method in Java](https://www.baeldung.com/java-finalize).

### ****3.6. Interned****Strings

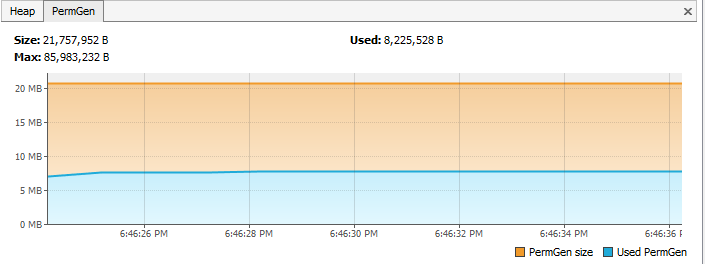
The Java String pool had gone through a major change in Java 7 when it was transferred from PermGen to HeapSpace. But for applications operating on version 6 and below, we should be more attentive when working with large Strings.

**If we read a huge massive String object, and call intern() on that object, then it goes to the string pool, which is located in PermGen (permanent memory) and will stay there as long as our application runs.** This blocks the memory and creates a major memory leak in our application.

The PermGen for this case in JVM 1.6 looks like this in VisualVM:

[](https://www.baeldung.com/wp-content/uploads/2018/11/Interned_Strings.png)

In contrast to this, in a method, if we just read a string from a file and do not intern it, then the PermGen looks like:

[](https://www.baeldung.com/wp-content/uploads/2018/11/Normal_Strings.png)

**How to Prevent It?**

* The simplest way to resolve this issue is by upgrading to latest Java version as String pool is moved to HeapSpace from Java version 7 onwards
* If working on large Strings, increase the size of the PermGen space to avoid any potential OutOfMemoryErrors:

|  |  |
| --- | --- |
| 1 | -XX:MaxPermSize=512m |

### ****3.7. Using****ThreadLocal****s****

[ThreadLocal](https://docs.oracle.com/javase/7/docs/api/java/lang/ThreadLocal.html) (discussed in detail in [Introduction to ThreadLocal in Java](https://www.baeldung.com/java-threadlocal) tutorial) is a construct that gives us the ability to isolate state to a particular thread and thus allows us to achieve thread safety.

When using this construct, **each thread will hold an implicit reference to its copy of a ThreadLocal variable and will maintain its own copy, instead of sharing the resource across multiple threads, as long as the thread is alive.**

Despite its advantages, the use of ThreadLocal variables is controversial, as they are infamous for introducing memory leaks if not used properly. Joshua Bloch [once commented on thread local usage](http://jsr166-concurrency.10961.n7.nabble.com/Threadlocals-and-memory-leaks-in-J2EE-td3960.html#a3984):

“Sloppy use of thread pools in combination with sloppy use of thread locals can cause unintended object retention, as has been noted in many places. But placing the blame on thread locals is unwarranted.”

**Memory leaks with ThreadLocals**

ThreadLocals are supposed to be garbage collected once the holding thread is no longer alive. But the problem arises when ThreadLocals are used along with modern application servers.

Modern application servers use a pool of threads to process requests instead of creating new ones (for example [the Executor](https://tomcat.apache.org/tomcat-7.0-doc/config/executor.html) in case of Apache Tomcat). Moreover, they also use a separate classloader.

Since [Thread Pools](https://www.baeldung.com/thread-pool-java-and-guava) in application servers work on the concept of thread reuse, they are never garbage collected — instead, they're reused to serve another request.

Now, if any class creates a ThreadLocal variable but doesn't explicitly remove it, then a copy of that object will remain with the worker Thread even after the web application is stopped, thus preventing the object from being garbage collected.

**How to Prevent It?**

* It's a good practice to clean-up ThreadLocals when they're no longer used — ThreadLocals provide the [**remove()**](https://docs.oracle.com/javase/7/docs/api/java/lang/ThreadLocal.html#remove()) method, which removes the current thread's value for this variable
* **Do not use ThreadLocal.set(null) to clear the value** — it doesn't actually clear the value but will instead look up the Map associated with the current thread and set the key-value pair as the current thread and null respectively
* It's even better to consider ThreadLocal as a resource that needs to be closed in a finally block just to make sure that it is always closed, even in the case of an exception:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | try {      threadLocal.set(System.nanoTime());      //... further processing  }  finally {      threadLocal.remove();  } |

## ****4. Other Strategies for Dealing with Memory Leaks****

Although there is no one-size-fits-all solution when dealing with memory leaks, there are some ways by which we can minimize these leaks.

### ****4.1. Enable Profiling****

Java profilers are tools that monitor and diagnose the memory leaks through the application. They analyze what's going on internally in our application — for example, how memory is allocated.

**Using profilers, we can compare different approaches and find areas where we can optimally use our resources.**

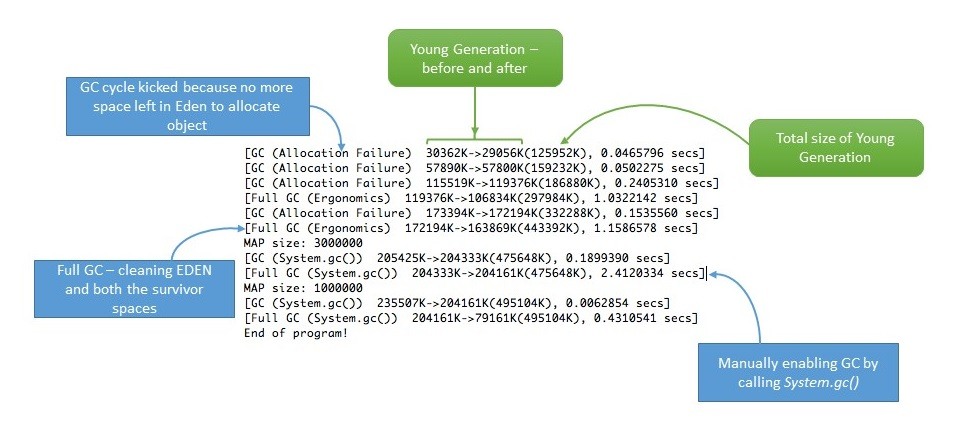
We have used [Java VisualVM](https://visualvm.github.io/) throughout section 3 of this tutorial. Please check out our [Guide to Java Profilers](https://www.baeldung.com/java-profilers) to learn about different types of profilers, like Mission Control, JProfiler, YourKit, Java VisualVM, and the Netbeans Profiler.

### ****4.2. Verbose Garbage Collection****

By enabling verbose garbage collection, we're tracking detailed trace of the GC. To enable this, we need to add the following to our JVM configuration:

|  |
| --- |
| -verbose:gc |

By adding this parameter, we can see the details of what's happening inside GC:

[](https://www.baeldung.com/wp-content/uploads/2018/11/verbose-garbage-collection.jpg)

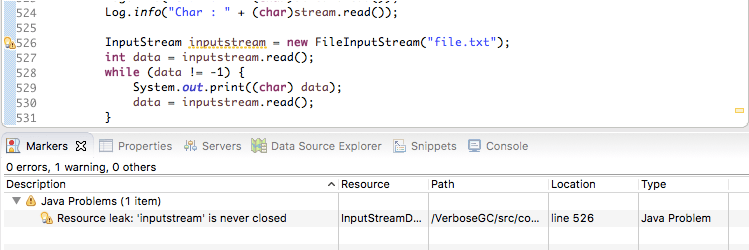
### ****4.3. Use Reference Objects to Avoid Memory Leaks****

We can also resort to reference objects in Java that comes in-built with java.lang.ref package to deal with memory leaks. Using java.lang.ref package, instead of directly referencing objects, we use special references to objects that allow them to be easily garbage collected.

Reference queues are designed for making us aware of actions performed by the Garbage Collector. For more information, read [Soft References in Java](https://www.baeldung.com/java-soft-references) Baeldung tutorial, specifically section 4.

### ****4.4. Eclipse Memory Leak Warnings****

For projects on JDK 1.5 and above, Eclipse shows warnings and errors whenever it encounters obvious cases of memory leaks. So when developing in Eclipse, we can regularly visit the “Problems” tab and be more vigilant about memory leak warnings (if any):

[](https://www.baeldung.com/wp-content/uploads/2018/11/Eclipse-_Memor-_Leak-_Warnings.png)

### ****4.5. Benchmarking****

We can measure and analyze the Java code's performance by executing benchmarks. This way, we can compare the performance of alternative approaches to do the same task. This can help us choose a better approach and may help us to conserve memory.

For more information about benchmarking, please head over to our [Microbenchmarking with Java](https://www.baeldung.com/java-microbenchmark-harness) tutorial.

### ****4.6. Code Reviews****

Finally, we always have the classic, old-school way of doing a simple code walk-through.

In some cases, even this trivial looking method can help in eliminating some common memory leak problems.

## ****5. Conclusion****

In layman's terms, we can think of memory leak as a disease that degrades our application's performance by blocking vital memory resources. And like all other diseases, if not cured, it can result in fatal application crashes over time.

Memory leaks are tricky to solve and finding them requires intricate mastery and command over the Java language. **While dealing with memory leaks, there is no one-size-fits-all solution, as leaks can occur through a wide range of diverse events.**

However, if we resort to best practices and regularly perform rigorous code walk-throughs and profiling, then we can minimize the risk of memory leaks in our application.

# **Java equals() and hashCode() Contracts**

## ****1. Overview****

In this tutorial, we'll introduce two methods that closely belong together: equals() and hashCode(). We'll focus on their relationship with each other, how to correctly override them, and why we should override both or neither.

## ****2.****equals()

The Object class defines both the equals() and hashCode() methods – which means that these two methods are implicitly defined in every Java class, including the ones we create:

class Money {  
 int amount;  
 String currencyCode;  
}  
  
Money income = new Money(55, "USD");  
Money expenses = new Money(55, "USD");  
boolean balanced = income.equals(expenses)

We would expect income.equals(expenses) to return true. But with the Money class in its current form, it won't.

**The default implementation of equals() in the class Object says that equality is the same as object identity. And income and expenses are two distinct instances.**

### ****2.1. Overriding****equals()

Let's override the equals() method so that it doesn't consider only object identity, but rather also the value of the two relevant properties:

@Override  
public boolean equals(Object o) {  
 if (o == this)  
 return true;  
 if (!(o instanceof Money))  
 return false;  
 Money other = (Money)o;  
 boolean currencyCodeEquals = (this.currencyCode == null && other.currencyCode == null)  
 || (this.currencyCode != null && this.currencyCode.equals(other.currencyCode));  
 return this.amount == other.amount && currencyCodeEquals;  
}

### ****2.2.****equals()****Contract****

Java SE defines a contract that our implementation of the equals() method must fulfill. **Most of the criteria are common sense.** The equals() method must be:

* reflexive: an object must equal itself
* symmetric: **x.equals(y) must return the same result as y.equals(x)**
* transitive: if x.equals(y) and y.equals(z) then also x.equals(z)
* consistent: the value of equals() should change only if a property that is contained in equals() changes (no randomness allowed)

We can look up the exact criteria in the [Java SE Docs for the Object class](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html).

### ****2.3. Violating****equals()****Symmetry with Inheritance****

If the criteria for equals() is such common sense, how can we violate it at all? Well, **violations happen most often, if we extend a class that has overridden equals().** Let's consider a Voucher class that extends our Money class:

class WrongVoucher extends Money {  
  
 private String store;  
  
 @Override  
 public boolean equals(Object o) {  
 if (o == this)  
 return true;  
 if (!(o instanceof WrongVoucher))  
 return false;  
 WrongVoucher other = (WrongVoucher)o;  
 boolean currencyCodeEquals = (this.currencyCode == null && other.currencyCode == null)  
 || (this.currencyCode != null && this.currencyCode.equals(other.currencyCode));  
 boolean storeEquals = (this.store == null && other.store == null)  
 || (this.store != null && this.store.equals(other.store));  
 return this.amount == other.amount && currencyCodeEquals && storeEquals;  
 }  
  
 // other methods  
}

At first glance, the Voucher class and its override for equals() seem to be correct. And both equals() methods behave correctly as long as we compare Money to Money or Voucher to Voucher. **But what happens, if we compare these two objects?**

Money cash = new Money(42, "USD");  
 WrongVoucher voucher = new WrongVoucher(42, "USD", "Amazon");  
   
 voucher.equals(cash) => false // As expected.  
 cash.equals(voucher) => true // That's wrong.

**That violates the symmetry criteria of the equals() contract.**

### ****2.4. Fixing****equals()****Symmetry with Composition****

To avoid this pitfall, we should**favor composition over inheritance.**

Instead of subclassing Money, let's create a Voucher class with a Money property:

class Voucher {  
  
 private Money value;  
 private String store;  
  
 Voucher(int amount, String currencyCode, String store) {  
 this.value = new Money(amount, currencyCode);  
 this.store = store;  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (o == this)  
 return true;  
 if (!(o instanceof Voucher))  
 return false;  
 Voucher other = (Voucher) o;  
 boolean valueEquals = (this.value == null && other.value == null)  
 || (this.value != null && this.value.equals(other.value));  
 boolean storeEquals = (this.store == null && other.store == null)  
 || (this.store != null && this.store.equals(other.store));  
 return valueEquals && storeEquals;  
 }  
  
 // other methods  
}  
class Voucher {  
  
 private Money value;  
 private String store;  
  
 Voucher(int amount, String currencyCode, String store) {  
 this.value = new Money(amount, currencyCode);  
 this.store = store;  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (o == this)  
 return true;  
 if (!(o instanceof Voucher))  
 return false;  
 Voucher other = (Voucher) o;  
 boolean valueEquals = (this.value == null && other.value == null)  
 || (this.value != null && this.value.equals(other.value));  
 boolean storeEquals = (this.store == null && other.store == null)  
 || (this.store != null && this.store.equals(other.store));  
 return valueEquals && storeEquals;  
 }  
  
 // other methods  
}

And now, equals will work symmetrically as the contract requires.

## ****3.****hashCode()

hashCode() returns an integer representing the current instance of the class. We should calculate this value consistent with the definition of equality for the class. Thus **if we override the equals()** method, we also have to override hashCode().

For some more details, check out our [guide to hashCode()](https://www.baeldung.com/java-hashcode).

### ****3.1.****hashCode()****Contract****

Java SE also defines a contract for the hashCode() method. A thorough look at it shows how closely related hashCode() and equals() are.

**All three criteria in the contract of hashCode() mention in some ways the equals() method:**

* internal consistency: the value of hashCode() may only change if a property that is in equals() changes
* equals consistency:**objects that are equal to each other must return the same hashCode**
* collisions:**unequal objects may have the same hashCode**

### ****3.2. Violating the Consistency of****hashCode()****and****equals()

The 2nd criteria of the hashCode methods contract has an important consequence: **If we override equals(), we must also override hashCode().** And this is by far the most widespread violation regarding the contracts of the equals() and hashCode() methods.

Let's see such an example:

class Team {  
  
 String city;  
 String department;  
  
 @Override  
 public final boolean equals(Object o) {  
 // implementation  
 }  
}

The Team class overrides only equals(), but it still implicitly uses the default implementation of hashCode() as defined in the Object class. And this returns a different hashCode() for every instance of the class. **This violates the second rule.**

Now if we create two Team objects, both with city “New York” and department “marketing”, they will be equal,**but they will return different hashCodes.**

### ****3.3.****HashMap****Key with an Inconsistent****hashCode()

But why is the contract violation in our Team class a problem? Well, the trouble starts when some hash-based collections are involved. Let's try to use our Team class as a key of a HashMap:

Map<Team, String> leaders = new HashMap<>();  
leaders.put(new Team("New York","development"), "Anne");  
leaders.put(new Team("Boston","development"), "Brian");  
leaders.put(new Team("Boston","marketing"), "Charlie");  
Team myTeam = new Team("New York", "development");  
String myTeamLeader = leaders.get(myTeam);

We would expect myTeamLeader to return “Anne”. **But with the current code, it doesn't.**

If we want to use instances of the Team class as HashMap keys, we have to override the hashCode() method so that it adheres to the contract: **Equal objects return the same hashCode.**

Let's see an example implementation:

@Override  
public final int hashCode() {  
 int result = 17;  
 if (city != null) {  
 result = 31 \* result + city.hashCode();  
 }  
 if (department != null) {  
 result = 31 \* result + department.hashCode();  
 }  
 return result;  
}

After this change, leaders.get(myTeam) returns “Anne” as expected.

## ****4. When Do We Override****equals()****and****hashCode()****?****

**Generally, we want to override either both of them or neither of them.** We've just seen in Section 3 the undesired consequences if we ignore this rule.

Domain-Driven Design can help us decide circumstances when we should leave them be. For entity classes – for objects having an intrinsic identity – the default implementation often makes sense.

However, **for value objects, we usually prefer equality based on their properties**. Thus want to override equals() and hashCode(). Remember our Money class from Section 2: 55 USD equals 55 USD – even if they're two separate instances.

## ****5. Implementation Helpers****

We typically don't write the implementation of these methods by hand. As can be seen, there are quite a few pitfalls.

One common way is to [let our IDE](https://www.baeldung.com/java-eclipse-equals-and-hashcode) generate the equals() and hashCode() methods.

[Apache Commons Lang](https://www.baeldung.com/java-commons-lang-3) and [Google Guava](https://www.baeldung.com/whats-new-in-guava-19) have helper classes in order to simplify writing both methods.

[Project Lombok](https://www.baeldung.com/intro-to-project-lombok) also provides an @EqualsAndHashCode annotation. **Note again how equals() and hashCode() “go together” and even have a common annotation.**

## ****6. Verifying the Contracts****

If we want to check whether our implementations adhere to the Java SE contracts and also to some best practices, **we can use the EqualsVerifier library.**

Let's add the [EqualsVerifier](https://mvnrepository.com/artifact/nl.jqno.equalsverifier/equalsverifier) Maven test dependency:

<dependency>  
 <groupId>nl.jqno.equalsverifier</groupId>  
 <artifactId>equalsverifier</artifactId>  
 <version>3.0.3</version>  
 <scope>test</scope>  
</dependency>

Let's verify that our Team class follows the equals() and hashCode() contracts:

@Test  
public void equalsHashCodeContracts() {  
 EqualsVerifier.forClass(Team.class).verify();  
}

It's worth noting that EqualsVerifier tests both the equals() and hashCode() methods.

**EqualsVerifier is much stricter than the Java SE contract.** For example, it makes sure that our methods can't throw a NullPointerException. Also, it enforces that both methods, or the class itself, is final.

It's important to realize that **the default configuration of EqualsVerifier allows only immutable fields**. This is a stricter check than what the Java SE contract allows. This adheres to a recommendation of Domain-Driven Design to make value objects immutable.

If we find some of the built-in constraints unnecessary, we can add a suppress(Warning.SPECIFIC\_WARNING) to our EqualsVerifier call.

## ****7. Conclusion****

In this article, we've discussed the equals() and hashCode() contracts. We should remember to:

* Always override hashCode() if we override equals()
* Override equals() and hashCode() for value objects
* Be aware of the traps of extending classes that have overridden equals() and hashCode()
* Consider using an IDE or a third-party library for generating the equals() and hashCode() methods
* Consider using EqualsVerifier to test our implementation

Finally, all code examples can be found [over on GitHub](https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-lang-oop-2).

# **Exception Handling in Java**

## ****1. Overview****

In this tutorial, we'll go through the basics of exception handling in Java as well as some of its gotchas.

## ****2. First Principles****

### ****2.1. What is it?****

To better understand exceptions and exception handling, let's make a real-life comparison.

Imagine that we order a product online, but while en-route, there's a failure in delivery. A good company can handle this problem and gracefully re-route our package so that it still arrives on time.

Likewise, in Java, the code can experience errors while executing our instructions. Good exception handling can handle errors and gracefully re-route the program to give the user still a positive experience.

### ****2.2. Why use it?****

We usually write code in an idealized environment: the filesystem always contains our files, the network is healthy, and the JVM always has enough memory. Sometimes we call this the “happy path”.

In production, though, filesystems can corrupt, networks break down, and JVMs run out of memory. The wellbeing of our code depends on how it deals with “unhappy paths”.

We must handle these conditions because they affect the flow of the application negatively and form exceptions:

public static List<Player> getPlayers() throws IOException {  
 Path path = Paths.get("players.dat");  
 List<String> players = Files.readAllLines(path);  
  
 return players.stream()  
 .map(Player::new)  
 .collect(Collectors.toList());  
}

This code chooses not to handle the IOException, passing it up the call stack instead. In an idealized environment, the code works fine.

But what might happen in production if players.dat is missing?

Exception in thread "main" java.nio.file.NoSuchFileException: players.dat <-- players.dat file doesn't exist  
at sun.nio.fs.WindowsException.translateToIOException(Unknown Source)  
at sun.nio.fs.WindowsException.rethrowAsIOException(Unknown Source)  
// ... more stack trace  
at java.nio.file.Files.readAllLines(Unknown Source)  
at java.nio.file.Files.readAllLines(Unknown Source)  
at Exceptions.getPlayers(Exceptions.java:12) <-- Exception arises in getPlayers() method, on line 12  
at Exceptions.main(Exceptions.java:19) <-- getPlayers() is called by main(), on line 19

**Without handling this exception, an otherwise healthy program may stop running altogether!** We need to make sure that our code has a plan for when things go wrong.

Also note one more benefit here to exceptions, and that is the stack trace itself. Because of this stack trace, we can often pinpoint offending code without needing to attach a debugger.

## ****3. Exception Hierarchy****

Ultimately, exceptions are just Java objects with all of them extending from Throwable:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | ---> Throwable <---                |    (checked)     |                |                  |                |                  |        ---> Exception           Error        |    (checked)        (unchecked)        |  RuntimeException    (unchecked) |

There are three main categories of exceptional conditions:

* Checked exceptions
* Unchecked exceptions / Runtime exceptions
* Errors

**Runtime and unchecked exceptions refer to the same thing. We can often use them interchangeably.**

### 3.1. ****Checked Exceptions****

Checked exceptions are exceptions that the Java compiler requires us to handle. We have to either declaratively throw the exception up the call stack, or we have to handle it ourselves. More on both of these in a moment.

[Oracle's documentation](https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html) tells us to use checked exceptions when we can reasonably expect the caller of our method to be able to recover.

A couple of examples of checked exceptions are IOException and ServletException.

### ****3.2. Unchecked Exceptions****

Unchecked exceptions are exceptions that the Java compiler does not require us to handle.

Simply put, if we create an exception that extends RuntimeException, it will be unchecked; otherwise, it will be checked.

And while this sounds convenient, [Oracle's documentation](https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html) tells us that there are good reasons for both concepts, like differentiating between a situational error (checked) and a usage error (unchecked).

Some examples of unchecked exceptions are NullPointerException, IllegalArgumentException, and SecurityException.

### ****3.3. Errors****

Errors represent serious and usually irrecoverable conditions like a library incompatibility, infinite recursion, or memory leaks.

And even though they don't extend RuntimeException, they are also unchecked.

In most cases, it'd be weird for us to handle, instantiate or extend Errors. Usually, we want these to propagate all the way up.

A couple of examples of errors are a StackOverflowError and OutOfMemoryError.

## ****4. Handling Exceptions****

In the Java API, there are plenty of places where things can go wrong, and some of these places are marked with exceptions, either in the signature or the Javadoc:

*/\*\*  
 \** ***@exception*** *FileNotFoundException ...  
 \*/*public Scanner(String fileName) throws FileNotFoundException {  
 // ...  
}

As stated a little bit earlier, when we call these “risky” methods, we must handle the checked exceptions, and we may handle the unchecked ones. Java gives us several ways to do this:

### ****4.1.****throws

The simplest way to “handle” an exception is to rethrow it:

public int getPlayerScore(String playerFile)  
 throws FileNotFoundException {  
  
 Scanner contents = new Scanner(new File(playerFile));  
 return Integer.*parseInt*(contents.nextLine());  
}

Because FileNotFoundException is a checked exception, this is the simplest way to satisfy the compiler, but **it does mean that anyone that calls our method now needs to handle it too!**

parseInt can throw a NumberFormatException, but because it is unchecked, we aren't required to handle it.

### ****4.2.****try****–****catch

If we want to try and handle the exception ourselves, we can use a try-catch block. We can handle it by rethrowing our exception:

public int getPlayerScore(String playerFile) {  
 try {  
 Scanner contents = new Scanner(new File(playerFile));  
 return Integer.*parseInt*(contents.nextLine());  
 } catch (FileNotFoundException noFile) {  
 throw new IllegalArgumentException("File not found");  
 }  
}

Or by performing recovery steps:

public int getPlayerScore(String playerFile) {  
 try {  
 Scanner contents = new Scanner(new File(playerFile));  
 return Integer.*parseInt*(contents.nextLine());  
 } catch ( FileNotFoundException noFile ) {  
 logger.warn("File not found, resetting score.");  
 return 0;  
 }  
}

### ****4.3.****finally

Now, there are times when we have code that needs to execute regardless of whether an exception occurs, and this is where the finally keyword comes in.

In our examples so far, there ‘s been a nasty bug lurking in the shadows, which is that Java by default won't return file handles to the operating system.

Certainly, whether we can read the file or not, we want to make sure that we do the appropriate cleanup!

Let's try this the “lazy” way first:

public int getPlayerScore(String playerFile)  
 throws FileNotFoundException {  
 Scanner contents = null;  
 try {  
 contents = new Scanner(new File(playerFile));  
 return Integer.*parseInt*(contents.nextLine());  
 } finally {  
 if (contents != null) {  
 contents.close();  
 }  
 }  
}

Here, the finally block indicates what code we want Java to run regardless of what happens with trying to read the file.

Even if a FileNotFoundException is thrown up the call stack, Java will call the contents of finally before doing that.

We can also both handle the exception and make sure that our resources get closed:

public int getPlayerScore(String playerFile) {  
 Scanner contents;  
 try {  
 contents = new Scanner(new File(playerFile));  
 return Integer.*parseInt*(contents.nextLine());  
 } catch (FileNotFoundException noFile ) {  
 logger.warn("File not found, resetting score.");  
 return 0;  
 } finally {  
 try {  
 if (contents != null) {  
 contents.close();  
 }  
 } catch (IOException io) {  
 logger.error("Couldn't close the reader!", io);  
 }  
 }  
}

**Because close is also a “risky” method, we also need to catch its exception!**

This may look pretty complicated, but we need each piece to handle each potential problem that can arise correctly.

### ****4.4.****try****-with-resources****

Fortunately, as of Java 7, we can simplify the above syntax when working with things that extend AutoCloseable:

public int getPlayerScore(String playerFile) {  
 try (Scanner contents = new Scanner(new File(playerFile))) {  
 return Integer.*parseInt*(contents.nextLine());  
 } catch (FileNotFoundException e ) {  
 logger.warn("File not found, resetting score.");  
 return 0;  
 }  
}

When we place references that are AutoClosable in the try declaration, then we don't need to close the resource ourselves.

We can still use a finally block, though, to do any other kind of cleanup we want.

Check out our article dedicated to [try-with-resources](https://www.baeldung.com/java-try-with-resources) to learn more.

### ****4.5. Multiple****catch****Blocks****

Sometimes, the code can throw more than one exception, and we can have more than one catch block handle each individually:

public int getPlayerScore(String playerFile) {  
 try (Scanner contents = new Scanner(new File(playerFile))) {  
 return Integer.*parseInt*(contents.nextLine());  
 } catch (IOException e) {  
 logger.warn("Player file wouldn't load!", e);  
 return 0;  
 } catch (NumberFormatException e) {  
 logger.warn("Player file was corrupted!", e);  
 return 0;  
 }  
}

Multiple catches give us the chance to handle each exception differently, should the need arise.

Also note here that we didn't catch FileNotFoundException, and that is because it extends IOException. Because we're catching IOException, Java will consider any of its subclasses also handled.

Let's say, though, that we need to treat FileNotFoundException differently from the more general IOException:

public int getPlayerScore(String playerFile) {  
 try (Scanner contents = new Scanner(new File(playerFile)) ) {  
 return Integer.*parseInt*(contents.nextLine());  
 } catch (FileNotFoundException e) {  
 logger.warn("Player file not found!", e);  
 return 0;  
 } catch (IOException e) {  
 logger.warn("Player file wouldn't load!", e);  
 return 0;  
 } catch (NumberFormatException e) {  
 logger.warn("Player file was corrupted!", e);  
 return 0;  
 }  
}

Java lets us handle subclass exceptions separately,**remember to place them higher in the list of catches.**

### ****4.6. Union****catch****Blocks****

When we know that the way we handle errors is going to be the same, though, Java 7 introduced the ability to catch multiple exceptions in the same block:

public int getPlayerScore(String playerFile) {  
 try (Scanner contents = new Scanner(new File(playerFile))) {  
 return Integer.*parseInt*(contents.nextLine());  
 } catch (IOException | NumberFormatException e) {  
 logger.warn("Failed to load score!", e);  
 return 0;  
 }  
}

## ****5. Throwing Exceptions****

If we don't want to handle the exception ourselves or we want to generate our exceptions for others to handle, then we need to get familiar with the throw keyword.

Let's say that we have the following checked exception we've created ourselves:

public class TimeoutException extends Exception {  
 public TimeoutException(String message) {  
 super(message);  
 }  
}

and we have a method that could potentially take a long time to complete:

public List<Player> loadAllPlayers(String playersFile) {  
 // ... potentially long operation  
}

### ****5.1. Throwing a Checked Exception****

Like returning from a method, we can throw at any point.

Of course, we should throw when we are trying to indicate that something has gone wrong:

public List<Player> loadAllPlayers(String playersFile) throws TimeoutException {  
 while ( !tooLong ) {  
 // ... potentially long operation  
 }  
 throw new TimeoutException("This operation took too long");  
}

Because TimeoutException is checked, we also must use the throws keyword in the signature so that callers of our method will know to handle it.

### ****5.2.****Throw****ing an Unchecked Exception****

If we want to do something like, say, validate input, we can use an unchecked exception instead:

public List<Player> loadAllPlayers(String playersFile) throws TimeoutException {  
 if(!isFilenameValid(playersFile)) {  
 throw new IllegalArgumentException("Filename isn't valid!");  
 }  
  
 // ...  
}

Because IllegalArgumentException is unchecked, we don't have to mark the method, though we are welcome to.

Some mark the method anyway as a form of documentation.

### ****5.3. Wrapping and Rethrowing****

We can also choose to rethrow an exception we've caught:

public List<Player> loadAllPlayers(String playersFile)  
 throws IOException {  
 try {  
 // ...  
 } catch (IOException io) {  
 throw io;  
 }  
}

Or do a wrap and rethrow:

public List<Player> loadAllPlayers(String playersFile)  
 throws PlayerLoadException {  
 try {  
 // ...  
 } catch (IOException io) {  
 throw new PlayerLoadException(io);  
 }  
}

This can be nice for consolidating many different exceptions into one.

### ****5.4. Rethrowing****Throwable****or****Exception

Now for a special case.

If the only possible exceptions that a given block of code could raise are unchecked exceptions, then we can catch and rethrow Throwable or Exception without adding them to our method signature:

public List<Player> loadAllPlayers(String playersFile) {  
 try {  
 throw new NullPointerException();  
 } catch (Throwable t) {  
 throw t;  
 }  
}

While simple, the above code can't throw a checked exception and because of that, even though we are rethrowing a checked exception, we don't have to mark the signature with a throws clause.

**This is handy with proxy classes and methods.**More about this can be found [here](http://4comprehension.com/sneakily-throwing-exceptions-in-lambda-expressions-in-java/).

### 5.5. ****Inheritance****

When we mark methods with a throws keyword, it impacts how subclasses can override our method.

In the circumstance where our method throws a checked exception:

public class Exceptions {  
 public List<Player> loadAllPlayers(String playersFile)  
 throws TimeoutException {  
 // ...  
 }  
}

A subclass can have a “less risky” signature:

public class FewerExceptions extends Exceptions {  
 @Override  
 public List<Player> loadAllPlayers(String playersFile) {  
 // overridden  
 }  
}

But not a “more riskier” signature:

public class MoreExceptions extends Exceptions {  
 @Override  
 public List<Player> loadAllPlayers(String playersFile) throws MyCheckedException {  
 // overridden  
 }  
}

This is because contracts are determined at compile time by the reference type. If I create an instance of MoreExceptions and save it to Exceptions:

|  |  |
| --- | --- |
| 1  2 | Exceptions exceptions = new MoreExceptions();  exceptions.loadAllPlayers("file"); |

Then the JVM will only tell me to catch the TimeoutException, which is wrong since I've said that MoreExceptions#loadAllPlayers throws a different exception.

**Simply put, subclasses can throw fewer checked exceptions than their superclass, but not more.**

## ****6. Anti-Patterns****

### ****6.1. Swallowing Exceptions****

Now, there’s one other way that we could have satisfied the compiler:

public int getPlayerScore(String playerFile) {  
 try {  
 // ...  
 } catch (Exception e) {} // <== catch and swallow  
 return 0;  
}

**The above is called** **swallowing an exception**. Most of the time, it would be a little mean for us to do this because it doesn't address the issue and it keeps other code from being able to address the issue, too.

There are times when there's a checked exception that we are confident will just never happen. **In those cases, we should still at least add a comment stating that we intentionally ate the exception**:

public int getPlayerScore(String playerFile) {  
 try {  
 // ...  
 } catch (IOException e) {  
 // this will never happen  
 }  
}

Another way we can “swallow” an exception is to print out the exception to the error stream simply:

public int getPlayerScore(String playerFile) {  
 try {  
 // ...  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 return 0;  
}

We've improved our situation a bit by a least writing the error out somewhere for later diagnosis.

It'd be better, though, for us to use a logger:

public int getPlayerScore(String playerFile) {  
 try {  
 // ...  
 } catch (IOException e) {  
 logger.error("Couldn't load the score", e);  
 return 0;  
 }  
}

While it's very convenient for us to handle exceptions in this way, we need to make sure that we aren't swallowing important information that callers of our code could use to remedy the problem.

Finally, we can inadvertently swallow an exception by not including it as a cause when we are throwing a new exception:

public int getPlayerScore(String playerFile) {  
 try {  
 // ...  
 } catch (IOException e) {  
 throw new PlayerScoreException();  
 }  
}

Here, we pat ourselves on the back for alerting our caller to an error, but **we fail to include the IOException as the cause.** Because of this, we've lost important information that callers or operators could use to diagnose the problem.

We'd be better off doing:

public int getPlayerScore(String playerFile) {  
 try {  
 // ...  
 } catch (IOException e) {  
 throw new PlayerScoreException(e);  
 }  
}

Notice the subtle difference of including IOException as the cause of PlayerScoreException.

### ****6.2. Using****return****in a****finally ****Block****

Another way to swallow exceptions is to return from the finally block. This is bad because, by returning abruptly, the JVM will drop the exception, even if it was thrown from by our code:

public int getPlayerScore(String playerFile) {  
 int score = 0;  
 try {  
 throw new IOException();  
 } finally {  
 return score; // <== the IOException is dropped  
 }  
}

According to the [Java Language Specification](https://docs.oracle.com/javase/specs/jls/se7/html/jls-14.html#jls-14.20.2):

If execution of the try block completes abruptly for any other reason R, then the finally block is executed, and then there is a choice.

If the finally block completes normally, then the try statement completes abruptly for reason R.

If the finally block completes abruptly for reason S, then the try statement completes abruptly for reason S (and reason R is discarded).

### ****6.3. Using****throw****in a****finally ****Block****

Similar to using return in a finally block, the exception thrown in a finally block will take precedence over the exception that arises in the catch block.

This will “erase” the original exception from the try block, and we lose all of that valuable information:

public int getPlayerScore(String playerFile) {  
 try {  
 // ...  
 } catch ( IOException io ) {  
 throw new IllegalStateException(io); // <== eaten by the finally  
 } finally {  
 throw new OtherException();  
 }  
}

### ****6.4. Using****throw****as a****goto

Some people also gave into the temptation of using throw as a goto statement:

public void doSomething() {  
 try {  
 // bunch of code  
 throw new MyException();  
 // second bunch of code  
 } catch (MyException e) {  
 // third bunch of code  
 }  
}

This is odd because the code is attempting to use exceptions for flow control as opposed to error handling.

## ****7. Common Exceptions and Errors****

Here are some common exceptions and errors that we all run into from time to time:

### ****7.1. Checked Exceptions****

* IOException – This exception is typically a way to say that something on the network, filesystem, or database failed.

### ****7.2. RuntimeExceptions****

* ArrayIndexOutOfBoundsException – this exception means that we tried to access a non-existent array index, like when trying to get index 5 from an array of length 3.
* ClassCastException – this exception means that we tried to perform an illegal cast, like trying to convert a String into a List. We can usually avoid it by performing defensive instanceof checks before casting.
* IllegalArgumentException – this exception is a generic way for us to say that one of the provided method or constructor parameters is invalid.
* IllegalStateException – This exception is a generic way for us to say that our internal state, like the state of our object, is invalid.
* NullPointerException – This exception means we tried to reference a null object. We can usually avoid it by either performing defensive null checks or by using Optional.
* NumberFormatException – This exception means that we tried to convert a String into a number, but the string contained illegal characters, like trying to convert “5f3” into a number.

### ****7.3. Errors****

* StackOverflowError – this exception means that the stack trace is too big. This can sometimes happen in massive applications; however, it usually means that we have some infinite recursion happening in our code.
* NoClassDefFoundError – this exception means that a class failed to load either due to not being on the classpath or due to failure in static initialization.
* OutOfMemoryError –  this exception means that the JVM doesn't have any more memory available to allocate for more objects. Sometimes, this is due to a memory leak.

## ****8. Conclusion****

In this article, we've gone through the basics of exception handling as well as some good and poor practice examples.

As always, all code found in this article can be found [over on GitHub](https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-lang)!

# **Practical Java Examples of the Big O Notation**

## ****1. Overview****

In this tutorial, we'll talk about what **Big O Notation means. We'll go through a few examples to investigate its effect on the running time of your code.**

## ****2. The Intuition of Big O Notation****

We often hear the **performance of an algorithm described using Big O Notation.**

The study of the performance of algorithms – or algorithmic complexity – falls into the field of [algorithm analysis](https://en.wikipedia.org/wiki/Analysis_of_algorithms). Algorithm analysis answers the question of how many resources, such as disk space or time, an algorithm consumes.

We'll be looking at time as a resource. Typically, the less time an algorithm takes to complete, the better.

## ****3. Constant Time Algorithms –****O(1)

How does this input size of an algorithm affect its running time? **Key to understanding Big O is understanding the rates at which things can grow.** The rate in question here is **time taken per input size.**

Consider this simple piece of code:

|  |  |
| --- | --- |
| 1  2 | int n = 1000;  System.out.println("Hey - your input is: " + n); |

Clearly, it doesn't matter what n is, above. This piece of code takes a constant amount of time to run. It's not dependent on the size of n.

Similarly:

|  |  |
| --- | --- |
| 1  2  3  4 | int n = 1000;  System.out.println("Hey - your input is: " + n);  System.out.println("Hmm.. I'm doing more stuff with: " + n);  System.out.println("And more: " + n); |

The above example is also constant time. Even if it takes 3 times as long to run, it doesn't depend on the size of the input, n. We denote constant time algorithms as follows: O(1). Note that O(2), O(3) or even O(1000) would mean the same thing.

We don't care about exactly how long it takes to run, only that it takes constant time.

## ****4. Logarithmic Time Algorithms –****O(log n)

Constant time algorithms are (asymptotically) the quickest. **Logarithmic time is the next quickest.** Unfortunately, they're a bit trickier to imagine.

One common example of a logarithmic time algorithm is the [binary search](https://en.wikipedia.org/wiki/Binary_search_algorithm) algorithm. To see how to implement binary search in Java, [click here.](https://www.baeldung.com/java-binary-search)

What is important here is that the **running time grows in proportion to the logarithm of the input (in this case, log to the base 2):**

|  |  |
| --- | --- |
| 1  2  3 | for (int i = 1; i < n; i = i \* 2){      System.out.println("Hey - I'm busy looking at: " + i);  } |

If n is 8, the output will be the following:

|  |  |
| --- | --- |
| 1  2  3 | Hey - I'm busy looking at: 1  Hey - I'm busy looking at: 2  Hey - I'm busy looking at: 4 |

Our simple algorithm ran log(8) = 3 times.

## ****5. Linear Time Algorithms –****O(n)

After logarithmic time algorithms, we get the next fastest class: **linear time algorithms.**

If we say something grows linearly, we mean that it grows directly proportional to the size of its inputs.

Think of a simple for loop:

|  |  |
| --- | --- |
| 1  2  3 | for (int i = 0; i < n; i++) {      System.out.println("Hey - I'm busy looking at: " + i);  } |

How many times does this for loop run? n times, of course! We don't know exactly how long it will take for this to run – and we don't worry about that.

**What we do know is that the simple algorithm presented above will grow linearly with the size of its input.**

We'd prefer a run time of 0.1n than (1000n + 1000), but both are still linear algorithms; they both grow directly in proportion to the size of their inputs.

Again, if the algorithm was changed to the following:

|  |  |
| --- | --- |
| 1  2  3  4  5 | for (int i = 0; i < n; i++) {      System.out.println("Hey - I'm busy looking at: " + i);      System.out.println("Hmm.. Let's have another look at: " + i);      System.out.println("And another: " + i);  } |

The runtime would still be linear in the size of its input, n. We denote linear algorithms as follows: O(n).

As with the constant time algorithms, we don't care about the specifics of the runtime. **O(2n+1) is the same as O(n)**, as Big O Notation concerns itself with growth for input sizes.

## ****6. N Log N Time Algorithms –****O(n log n)

**n log n is the next class of algorithms.** The running time grows in proportion to n log n of the input:

|  |  |
| --- | --- |
| 1  2  3  4  5 | for (int i = 1; i <= n; i++){      for(int j = 1; j < 8; j = j \* 2) {          System.out.println("Hey - I'm busy looking at: " + i + " and " + j);      }  } |

For example, if the n is 8, then this algorithm will run 8 \* log(8) = 8 \* 3 = 24 times. Whether we have strict inequality or not in the for loop is irrelevant for the sake of a Big O Notation.

## ****7. Polynomial Time Algorithms –****O(np)

Next up we've got polynomial time algorithms. These algorithms are even slower than n log n algorithms.

The term polynomial is a general term which contains quadratic (n2), cubic (n3), quartic (n4), etc. functions. **What's important to know is that O(n2) is faster than O(n3) which is faster than O(n4), etc.**

Let's have a look at a simple example of a quadratic time algorithm:

|  |  |
| --- | --- |
| 1  2  3  4  5 | for (int i = 1; i <= n; i++) {      for(int j = 1; j <= n; j++) {          System.out.println("Hey - I'm busy looking at: " + i + " and " + j);      }  } |

This algorithm will run 82 = 64 times. Note, if we were to nest another for loop, this would become an O(n3) algorithm.

## ****8. Exponential Time Algorithms –****O(****k****n)

Now we are getting into dangerous territory; these algorithms grow in proportion to some factor exponentiated by the input size.

For example, **O(2n) algorithms double with every additional input.** So, if n = 2, these algorithms will run four times; if n = 3, they will run eight times (kind of like the opposite of logarithmic time algorithms).

**O(3n) algorithms triple with every additional input, O(kn) algorithms will get k times bigger with every additional input.**

Let's have a look at a simple example of an O(2n) time algorithm:

|  |  |
| --- | --- |
| 1  2  3 | for (int i = 1; i <= Math.pow(2, n); i++){      System.out.println("Hey - I'm busy looking at: " + i);  } |

This algorithm will run 28 = 256 times.

## ****9. Factorial Time Algorithms –****O(n!)

In most cases, this is pretty much as bad as it'll get. This class of algorithms has a run time proportional to the [factorial](https://en.wikipedia.org/wiki/Factorial) of the input size.

A classic example of this is solving the [traveling salesman](https://en.wikipedia.org/wiki/Travelling_salesman_problem) problem using a brute-force approach to solve it.

An explanation of the solution to the traveling salesman problem is beyond the scope of this article.

Instead, let's look at a simple O(n!) algorithm, as in the previous sections:

|  |  |
| --- | --- |
| 1  2  3 | for (int i = 1; i <= factorial(n); i++){      System.out.println("Hey - I'm busy looking at: " + i);  } |

where factorial(n) simply calculates n!. If n is 8, this algorithm will run 8! = 40320 times.

## ****10. Asymptotic Functions****

**Big O is what is known as an asymptotic function.** All this means, is that it concerns itself with the performance of an algorithm at the limit – i.e. – when lots of input is thrown at it.

Big O doesn't care about how well your algorithm does with inputs of small size. It's concerned with large inputs (think sorting a list of one million numbers vs. sorting a list of 5 numbers).

Another thing to note is that **there are other asymptotic functions.** Big Θ (theta) and Big Ω (omega) also both describes algorithms at the limit (remember, the limit this just means for huge inputs).

To understand the differences between these 3 important functions, we first need to know that each of Big O, Big Θ, and Big Ω describes a set (i.e., a collection of elements).

Here, the members of our sets are algorithms themselves:

* Big O describes the set of all algorithms that run no worse than a certain speed (it's an upper bound)
* Conversely, Big Ω describes the set of all algorithms that run no better than a certain speed (it's a lower bound)
* Finally, Big Θ describes the set of all algorithms that run at a certain speed (it's like equality)

The definitions we've put above are not mathematically accurate, but they will aid our understanding.

**Usually, you'll hear things described using Big O**, but it doesn't hurt to know about Big Θ and Big Ω.

## ****11. Conclusion****

In this article, we discussed Big O notation, and how **understanding the complexity of an algorithm can affect the running time of your code.**

A great visualization of the different complexity classes [can be found here.](http://bigocheatsheet.com/)

As usual, the code snippets for this tutorial can be found [over on GitHub](https://github.com/eugenp/tutorials/tree/master/algorithms-miscellaneous-3).

# **Design Patterns**

## Java Singleton Design

In [previous](https://www.geeksforgeeks.org/singleton-design-pattern-introduction/) articles, we discussed about singleton design pattern and singleton class [implementation](https://www.geeksforgeeks.org/singleton-design-pattern/)in detail.  
In this article, we will see how we can create singleton classes. After reading this article you will be able to create your singleton class according to your use, simplicity and removed bottlenecks.  
There are many ways this can be done in Java. All these ways differs in their implementation of the pattern, but in the end, they all achieve the same end result of a single instance.

1. **Eager initialization:** This is the simplest method of creating a singleton class. In this, object of class is created when it is loaded to the memory by JVM. It is done by assigning the reference an instance directly.  
   It can be used when program will always use instance of this class, or the cost of creating the instance is not too large in terms of resources and time.

// Java code to create singleton class by   
// Eager Initialization   
public class GFG {  
 // public instance initialized when loading the class   
 private static final GFG *instance* = new GFG();  
  
 private GFG() {  
 // private constructor   
 }  
  
 public static GFG getInstance() {  
 return *instance*;  
 }  
}

**Pros:**

* 1. Very simple to implement.

**Cons:**

* 1. May lead to resource wastage. Because instance of class is created always, whether it is required or not.
  2. CPU time is also wasted in creation of instance if it is not required.
  3. Exception handling is not possible.

1. **Using static block:** This is also a sub part of Eager initialization. The only difference is object is created in a static block so that we can have access on its creation, like exception handling. In this way also, object is created at the time of class loading.  
   It can be used when there is a chance of exceptions in creating object with eager initialization.

// Java code to create singleton class   
// Using Static block   
public class GFG {  
 // public instance   
 public static GFG *instance*;  
  
 private GFG() {  
 // private constructor   
 }  
  
 {  
 // static block to initialize instance   
 *instance* = new GFG();  
 }  
}

**Pros:**

* 1. Very simple to implement.
  2. No need to implement getInstance() method. Instance can be accessed directly.
  3. Exceptions can be handled in static block.

**Cons:**

* 1. May lead to resource wastage. Because instance of class is created always, whether it is required or not.
  2. CPU time is also wasted in creation of instance if it is not required.

1. **Lazy initialization:** In this method, object is created only if it is needed. This may prevent resource wastage. An implementation of getInstance() method is required which return the instance. There is a null check that if object is not created then create, otherwise return previously created. To make sure that class cannot be instantiated in any other way, constructor is made final. As object is created with in a method, it ensures that object will not be created until and unless it is required. Instance is kept private so that no one can access it directly.  
   It can be used in a single threaded environment because multiple threads can break singleton property because they can access get instance method simultaneously and create multiple objects.

//Java Code to create singleton class   
// With Lazy initialization   
public class GFG {  
 // private instance, so that it can be   
 // accessed by only by getInstance() method   
 private static GFG *instance*;  
  
 private GFG() {  
 // private constructor   
 }  
  
 //method to return instance of class   
 public static GFG getInstance() {  
 if (*instance* == null) {  
 // if instance is null, initialize   
 *instance* = new GFG();  
 }  
 return *instance*;  
 }  
}

**Pros:**

* 1. Object is created only if it is needed. It may overcome resource overcome and wastage of CPU time.
  2. Exception handling is also possible in method.

**Cons:**

* 1. Every time a condition of null has to be checked.
  2. instance can’t be accessed directly.
  3. In multithreaded environment, it may break singleton property.

1. **Thread Safe Singleton:** A thread safe singleton in created so that singleton property is maintained even in multithreaded environment. To make a singleton class thread-safe, getInstance() method is made synchronized so that multiple threads can’t access it simultaneously.

// Java program to create Thread Safe   
// Singleton class   
public class GFG {  
 // private instance, so that it can be   
 // accessed by only by getInstance() method   
 private static GFG *instance*;  
  
 private GFG() {  
 // private constructor   
 }  
  
 //synchronized method to control simultaneous access   
 synchronized public static GFG getInstance() {  
 if (*instance* == null) {  
 // if instance is null, initialize   
 *instance* = new GFG();  
 }  
 return *instance*;  
 }  
}

**Pros:**

* 1. Lazy initialization is possible.
  2. It is also thread safe.

**Cons:**

* 1. getInstance() method is synchronized so it causes slow performance as multiple threads can’t access it simultaneously.

1. **Lazy initialization with Double check locking:** In this mechanism, we overcome the overhead problem of synchronized code. In this method, getInstance is not synchronized but the block which creates instance is synchronized so that minimum number of threads have to wait and that’s only for first time.

// Java code to explain double check locking   
public class GFG {  
 // private instance, so that it can be   
 // accessed by only by getInstance() method   
 private static GFG *instance*;  
  
 private GFG() {  
 // private constructor   
 }  
  
 public static GFG getInstance() {  
 if (*instance* == null) {  
 //synchronized block to remove overhead   
 synchronized (GFG.class) {  
 if (*instance* == null) {  
 // if instance is null, initialize   
 *instance* = new GFG();  
 }  
  
 }  
 }  
 return *instance*;  
 }  
}

**Pros:**

* 1. Lazy initialization is possible.
  2. It is also thread safe.
  3. Performance reduced because of synchronized keyword is overcome.

**Cons:**

* 1. First time, it can affect performance.

As cons. of double check locking method is bearable so it can be used for high performance multi-threaded applications.

1. **Bill Pugh Singleton Implementation:** Prior to Java5, memory model had a lot of issues and above methods caused failure in certain scenarios in multithreaded environment. So, Bill Pugh suggested a concept of inner static classes to use for singleton.

// Java code for Bill Pugh Singleton Implementaion   
public class GFG {  
  
 private GFG() {  
 // private constructor   
 }  
  
 // Inner class to provide instance of class   
 private static class BillPughSingleton {  
 private static final GFG *INSTANCE* = new GFG();  
 }  
  
 public static GFG getInstance() {  
 return BillPughSingleton.*INSTANCE*;  
 }  
}

When the singleton class is loaded, inner class is not loaded and hence doesn’t create object when loading the class. Inner class is created only when getInstance() method is called. So it may seem like eager initialization but it is lazy initialization.  
This is the most widely used approach as it doesn’t use synchronization.

**When to use What**

1. Eager initialization is easy to implement but it may cause resource and CPU time wastage. Use it only if cost of initializing a class is less in terms of resources or your program will always need the instance of class.
2. By using Static block in Eager initialization we can provide exception handling and also can control over instance.
3. Using synchronized we can create singleton class in multi-threading environment also but it can cause slow performance, so we can use Double check locking mechanism.
4. Bill Pugh implementation is most widely used approach for singleton classes. Most developers prefer it because of its simplicity and advantages.

# **How to Replace Many if Statements in Java**

## ****1. Overview****

Decision constructs are a vital part of any programming language. But we land up in coding a huge number of nested if statements which make our code more complex and difficult to maintain.

In this tutorial, we'll walk through the **various ways of replacing nested if statements**.

Let's explore different options how we can simplify the code.

## ****2. Case Study****

Often we encounter a business logic which involves a lot of conditions, and each of them needs different processing. For the sake of a demo, let's take the example of a Calculator class. We will have a method which takes two numbers and an operator as input and returns the result based on the operation:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public int calculate(int a, int b, String operator) {      int result = Integer.MIN\_VALUE;        if ("add".equals(operator)) {          result = a + b;      } else if ("multiply".equals(operator)) {          result = a \* b;      } else if ("divide".equals(operator)) {          result = a / b;      } else if ("subtract".equals(operator)) {          result = a - b;      }      return result;  } |

We can also implement this using switch statements:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public int calculateUsingSwitch(int a, int b, String operator) {      switch (operator) {      case "add":          result = a + b;          break;      // other cases      }      return result;  } |

In typical development, **the if statements may grow much bigger and more complex in nature**. Also, **the switch statements do not fit well when there are complex conditions**.

Another side effect of having nested decision constructs is they become unmanageable. For example, if we need to add a new operator, we have to add a new if statement and implement the operation.

## ****3. Refactoring****

Let's explore the alternate options to replace the complex if statements above into much simpler and manageable code.

### ****3.1. Factory Class****

Many times we encounter decision constructs which end up doing the similar operation in each branch. This provides an opportunity to **extract a factory method which returns an object of a given type and performs the operation based on the concrete object behavior**.

For our example, let's define an Operation interface which has a single apply method:

|  |  |
| --- | --- |
| 1  2  3 | public interface Operation {      int apply(int a, int b);  } |

The method takes two number as input and returns the result. Let's define a class for performing additions:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class Addition implements Operation {      @Override      public int apply(int a, int b) {          return a + b;      }  } |

We'll now implement a factory class which returns instances of Operation based on the given operator:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class OperatorFactory {      static Map<String, Operation> operationMap = new HashMap<>();      static {          operationMap.put("add", new Addition());          operationMap.put("divide", new Division());          // more operators      }        public static Optional<Operation> getOperation(String operator) {          return Optional.ofNullable(operationMap.get(operator));      }  } |

Now, in the Calculator class, we can query the factory to get the relevant operation and apply on the source numbers:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public int calculateUsingFactory(int a, int b, String operator) {      Operation targetOperation = OperatorFactory        .getOperation(operator)        .orElseThrow(() -> new IllegalArgumentException("Invalid Operator"));      return targetOperation.apply(a, b);  } |

In this example, we have seen how the responsibility is delegated to loosely coupled objects served by a factory class. But there could be chances where the nested if statements are simply shifted to the factory class which defeats our purpose.

Alternatively, **we can maintain a repository of objects in a Map which could be queried for a quick lookup**. As we have seen OperatorFactory#operationMap serves our purpose. We can also initialize Map at runtime and configure them for lookup.

### ****3.2. Use of Enums****

In addition to the use of Map, **we can also use Enum to label particular business logic**. After that, we can use them either in the nested if statements or switch case statements. Alternatively, we can also use them as a factory of objects and strategize them to perform the related business logic.

That would reduce the number of nested if statements as well and delegate the responsibility to individual Enum values.

Let's see how we can achieve it. At first, we need to define our Enum:

|  |  |
| --- | --- |
| 1  2  3 | public enum Operator {      ADD, MULTIPLY, SUBTRACT, DIVIDE  } |

As we can observe, the values are the labels of the different operators which will be used further for calculation. We always have an option to use the values as different conditions in nested if statements or switch cases, but let's design an alternate way of delegating the logic to the Enum itself.

We'll define methods for each of the Enum values and do the calculation. For instance:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | ADD {      @Override      public int apply(int a, int b) {          return a + b;      }  },  // other operators    public abstract int apply(int a, int b); |

And then in the Calculator class, we can define a method to perform the operation:

|  |  |
| --- | --- |
| 1  2  3 | public int calculate(int a, int b, Operator operator) {      return operator.apply(a, b);  } |

Now, we can invoke the method by **converting the String value to the Operator by using the Operator#valueOf() method**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Test  public void whenCalculateUsingEnumOperator\_thenReturnCorrectResult() {      Calculator calculator = new Calculator();      int result = calculator.calculate(3, 4, Operator.valueOf("ADD"));      assertEquals(7, result);  } |

### ****3.3. Command Pattern****

In the previous discussion, we have seen the use of factory class to return the instance of the correct business object for the given operator. Later, the business object is used to perform the calculation in the Calculator.

**We can also design a Calculator#calculate method to accept a command which can be executed on the inputs**. This will be another way of replacing nested if statements.

We'll first define our Command interface:

|  |  |
| --- | --- |
| 1  2  3 | public interface Command {      Integer execute();  } |

Next, let's implement an AddCommand:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class AddCommand implements Command {      // Instance variables        public AddCommand(int a, int b) {          this.a = a;          this.b = b;      }        @Override      public Integer execute() {          return a + b;      }  } |

Finally, let's introduce a new method in the Calculator which accepts and execute the Command:

|  |  |
| --- | --- |
| 1  2  3 | public int calculate(Command command) {      return command.execute();  } |

Next, we can invoke the calculation by instantiating an AddCommand and send it to the Calculator#calculate method:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Test  public void whenCalculateUsingCommand\_thenReturnCorrectResult() {      Calculator calculator = new Calculator();      int result = calculator.calculate(new AddCommand(3, 7));      assertEquals(10, result);  } |

### ****3.4. Rule Engine****

When we end up writing a large number of nested if statements, each of the conditions depicts a business rule which has to be evaluated for the correct logic to be processed. A rule engine takes such complexity out of the main code. **A RuleEngine evaluates the Rules and returns the result based on the input.**

Let's walk through an example by designing a simple RuleEngine which processes an Expression through a set of Rules and returns the result from the selected Rule. First, we'll define a Rule interface:

|  |  |
| --- | --- |
| 1  2  3  4 | public interface Rule {      boolean evaluate(Expression expression);      Result getResult();  } |

Second, let's implement a RuleEngine:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class RuleEngine {      private static List<Rule> rules = new ArrayList<>();        static {          rules.add(new AddRule());      }        public Result process(Expression expression) {          Rule rule = rules            .stream()            .filter(r -> r.evaluate(expression))            .findFirst()            .orElseThrow(() -> new IllegalArgumentException("Expression does not matches any Rule"));          return rule.getResult();      }  } |

The RuleEngine accepts an Expression object and returns the Result. Now, let's design the Expression class as a group of two Integer objects with the Operator which will be applied:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class Expression {      private Integer x;      private Integer y;      private Operator operator;  } |

And finally let's define a custom AddRule class which evaluates only when the ADD Operation is specified:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class AddRule implements Rule {      @Override      public boolean evaluate(Expression expression) {          boolean evalResult = false;          if (expression.getOperator() == Operator.ADD) {              this.result = expression.getX() + expression.getY();              evalResult = true;          }          return evalResult;      }  } |

We'll now invoke the RuleEngine with an Expression:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Test  public void whenNumbersGivenToRuleEngine\_thenReturnCorrectResult() {      Expression expression = new Expression(5, 5, Operator.ADD);      RuleEngine engine = new RuleEngine();      Result result = engine.process(expression);        assertNotNull(result);      assertEquals(10, result.getValue());  } |

## ****4. Conclusion****

In this tutorial, we explored a number of different options to simplify complex code. We also learned how to replace nested if statements by the use of effective design patterns.

As always, we can find the complete source code over the [GitHub repository](https://github.com/eugenp/tutorials/tree/master/patterns/design-patterns-creational).