Java SE

Table of Contents

What is Java? **4**
History of Java **4**
Editions of Java **4**
Java SE (Standard Edition) 5
Java EE (Enterprise Edition) 5
Features of Java Language **6**
Java is Robust 6
Platform Independent Language 7
Compiling and running of Java Program 8
Java and Platform Independence 9
Using Java SE **9**
Directory Structure of Java SE 9
Basic Tools **10**
First Java Program – Hello.java **10**
Compiling and running a Java program 11
Standard or Primitive data types in Java **12**
Rules to create Identifier **13**
Keywords in Java **13**
Escape Sequence Characters **14**
Text Blocks **15**
Operators in Java **15**
Arithmetic Operators 15
Assignment Operator 16
Relational Operators 17
Logical Operators and Short-circuiting 17
Short-circuit Logical Operators (&& and ||) 17
Bitwise Operators 18
Control Statements **18**
if statement 18
Switch statement 19
The switch expression 20
Multiple labels 21
Using yield in switch 21
The while loop 22

The do .. while loop..... 22
The for loop..... 23
The break statement..... 23
The continue statement..... 24
Enhanced for loop..... 24
Printing value using System.out.printf()..... 25
Reading input using Scanner..... 25
Creating and using Arrays..... 26
Getting array's length using length property..... 27
Initializing an array..... 28
Variable Number of Arguments..... 28
Command-line Parameters..... 29
Program - Prime.java..... 30
Object Oriented Programming..... 31
Encapsulation..... 32
Class vs. Object..... 32
Inheritance..... 33
Polymorphism..... 34
Creating a Class..... 34
Constructor..... 37
Overloading Methods..... 38
Overloading Constructors..... 39
Object Reference..... 40
Comparing object references..... 41
Assignment between two object references..... 41
Array of objects..... 42
The this reference..... 42
Static Variables..... 43
Static Methods..... 44
Why main() is a static method?..... 44
Final Variable..... 45
Inheritance..... 47
Constructors in Inheritance..... 49
Overriding Methods..... 49
@Override Annotation..... 50
Overloading vs. Overriding..... 50
Using super keyword to access superclass version..... 51

Has A" Relationship....51
Multi-level inheritance....52
Multiple Inheritance....53
Runtime Polymorphism....53
Abstract Method....54
Final Variable, Method, Class and Parameter....55
Initialization Blocks....57
Interfaces....59
Implementing an Interface....60
Using object reference of an Interface....62
An Interface extending another....63
Default Methods in Interface....64
Static Methods in Interface....64
Functional Interface....66
Abstract Class vs. Interface....67
Null Interface....67
Private Methods in Interface....67
Packages....69
Creating a Package and placing a Class in it....69
Accessing Classes of a Package....70
Import Statement....71
CLASSPATH variable....72
JAR (Java Archive) File....72
Access Modifiers....73
Static Import....75
Enumeration....76
Abstract Class....77
Sealed Classes....77
Nested Classes....78
Static member Class....78
Non-static member class (inner class)....79
Method-local Inner Class....80
Anonymous Inner Class....81
Exception Handling....82
Handling exception using try and catch....84
Handling multiple exceptions....85
Multi-Catch....86

The finally block..... 87
Types of Exceptions – checked and unchecked..... 90
Unchecked Exceptions..... 90
Checked Exceptions..... 90
Invalid vs. Valid order of catch blocks..... 91
Valid order of catch blocks..... 92
Creating user-defined exception..... 92
Exception class..... 94
User-defined exceptions example..... 94
Assertions..... 96
Collections Framework..... 97
Collection Interface..... 99
List Interface..... 100
ArrayList Class..... 101
Set Interface..... 101
HashSet Class..... 102
SortedSet Interface..... 103
TreeSet Class..... 104
Comparator<T> Interface..... 105
Queue Interface..... 105
LinkedList Class..... 106
Map Interface..... 107
HashMap Class..... 108
SortedMap Interface..... 108
TreeMap Class..... 108
Collections Class..... 110
Immutable Collections..... 111
Generics..... 112
Generic Methods..... 113
Bounded Type Parameter..... 114
Downcasting and Upcasting..... 114
The instanceof operator..... 116
Pattern matching for instanceof..... 116

What is Java?

- A general-purpose programming language.
- Invented by **James Gosling**, Patrick Naughton, Chris Warth, Ed Frank and Mike Sheridan at Sun Microsystems.
- Java was released in **May, 1995**.
- Most widely used programming language.
- Mainly used to develop web and enterprise applications.
- Provides "write once and run anywhere" features – Platform independence.
- Influenced by **C and C++**.
- Derives its syntax from C and C++. However, it eliminated features of C++ such as multiple inheritance, operator overloading, friend function etc. that could cause ambiguity.

History of Java

The following are some important versions of Java, when they were released and what features were added.

- Java 1.0 was released in May, 1995. Java was initially called "oak" and then renamed to Java. JDK (Java Development Kit) 1.0 was released on January 23, 1996.
- Java 5.0 was released in 2004 adding annotations and generics.
- Java 8.0 was released in 2014 adding lambda expressions and streams.
- Java 9.0 was released in 2017 adding JShell and JPMS.
- Starting from Java 10, a new version was released every 6 months.
- The most recent release was Java 22 in Mar, 2024

Editions of Java

Java is provided as three different editions, where each edition is meant for different types of applications.

- Java Standard Edition – Java SE
- Java Enterprise Edition – Java EE

Java SE (Standard Edition)

- This is used to develop Console applications, Applets and Frame-based applications (desktop applications).
- This edition provides features of language, core API (Application Program Interface) related to IO Stream, Networking, Data structures using Collections Framework etc.
- This edition comes with Java compiler, JVM and other tools to develop applets and desktop applications.

Java EE (Enterprise Edition)

- This is used to develop web applications, web services, and enterprise applications. It uses Java Language and core API.
- It is a set of specifications to be implemented by products. BEA's WebLogic server, IBM's WebSphere, Open-source product JBoss, Glassfish etc. implement Java EE specifications.
- Major topics of Java EE are Servlets, JSPs, Enterprise Java Beans (EJB), JavaServer Faces (JSF)

Features of Java Language

The following are the major features of the Java language.

Java is

Robust	Makes programs robust (free from potential errors). Java detects all potential problems at the time of compiling the program. It also checks for array boundary violation etc. at run time.
Object Oriented	Supports encapsulation, inheritance and polymorphism. Java allows the entire code to be placed only in classes. It doesn't allow any code to be placed outside the class including the main() method.
Dynamic	Allocates memory for arrays and objects at run time and handles allocation and deallocation of memory on its own. Garbage collector is responsible for releasing unused memory blocks.
Multithreaded	Supports creating and managing multiple threads.
Distributed	Supports programs to run on different systems and yet communicate with each other. Provides a mechanism to transmit data from one application to another.
Platform Independent	Allows a compiled program (byte code) to run on any other platform.

Robust

- Java, being a robust language, checks for potential errors both at the time of compilation and at runtime.
- Java is a strongly typed language, where data type of a value cannot be changed to another, and is case-sensitive.
- The following snippets show operations that are allowed in **C** but not in **Java.**

```
01: int a[10];
02: int i, j;         // local variables, not initialized
03: a[10] = 25;      //accessing element outside the bounds
04: i = 10.50;      // assigning float to int
05: j++;           // using without initializing it
```

- Java also checks whether a function that is supposed to return a value is really returning a value.

```
01: /* This function does not compile as it does not
02:  return value when condition is false. */
03: int fun(int v) {
04:     if (v > 10)
05:         return 1;
06:   }
```

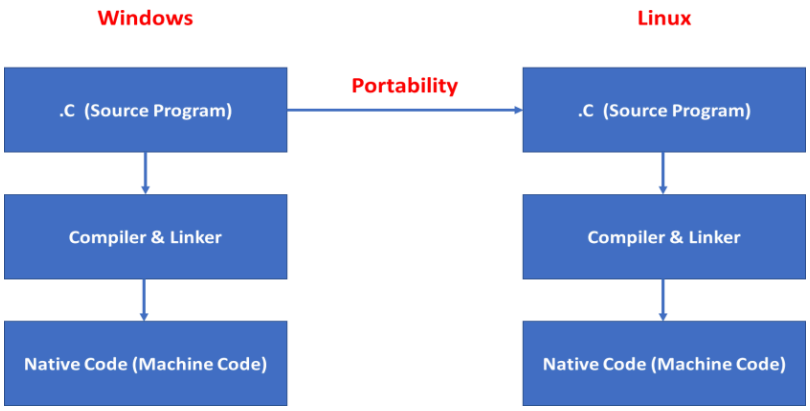
- Java checks whether a piece of code is reachable during the execution of a program. If code is not reachable in all paths of execution, then it complains about it.

```
01: /* Unreachable code example */
02: int fun(int v) {
03:    if (v > 10)
04:        return 1;
05:    else
06:       return 0;
07:
08:    V ++; // cannot be reached.
09: }
```

Platform Independent Language

The main feature of the Java language is its **platform independence**. This allows a Java program that is compiled on one platform to run on any other platform.

This is called WORA (Write Once, Run Anywhere). Platform independence is different from **Portability**.
Portability allows source code to be ported to different platforms. Though the source program is ported we need to compile the source program on the target platform before running it.



All high-level languages have portability.

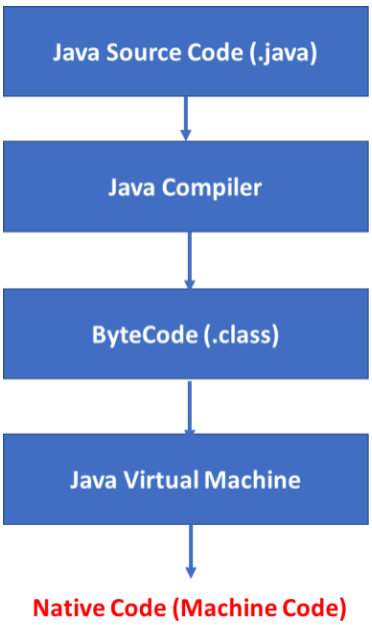
Native code is the code understood by the Microprocessor and operating system of the computer.

NOTE: Platform is the environment in which application programs run. It is typically the combination of operating system and hardware. Platform is generally synonymous with operating systems.

Compiling and running of Java Program

The following figure shows the steps related to compiling and running a Java program. Java programs are NOT compiled to native code and instead they are compiled to **Bytecode**, which is close to native code but not native code of any platform.

Once a Java program is compiled to bytecode then it can be run on any platform where JVM (Java Virtual Machine) is available.



NOTE: Java compiler and JVM are specific to each platform. Only bytecode is platform independent.

Java and Platform Independence

The following picture shows how a Java program that is compiled on a Windows system can run on a Linux system.

Once a Java program is compiled to Bytecode (.class file), it can be executed on any machine where JVM is available.

Most of the modern operating systems provide JVM along with operating systems allowing byte code to run straight away.

Using Java SE

- Download Java SE (a.k.a. JDK – Java Development Kit) from https://www.oracle.com/in/java/technologies/javase-downloads.html
- Installing Java SE is as simple as double clicking on downloaded file (**.exe** file) and following the steps.
- It is better if you install Java SE into a directory like **c:\jdk**, instead of the default directory (which is **program files** directory).

Directory Structure of Java SE

After you install Java SE into a directory (say c:\jdk), the structure of the directory will look like below:

```
c:\jdk
    bin
      javac.exe, java.exe
    include
    lib
       *.jar
    conf
       *.properties
```

Directory	Description
bin	Contains executable files related to JDK. It contains JAVAC.EXE and JAVA.EXE.
include	Includes files for Java Native Interface.
lib	Contains Java libraries, which are in the form of .jar (Java archive) files.
conf	Contains configuration files.

Basic Tools

These tools are the foundation of the JDK. They are the tools you use to create and build applications.

Tool Name	Brief Description
Javac	The compiler for the Java programming language.
Java	The launcher for Java applications (JVM).
Javadoc	API documentation generator.
Jar	Create and manage Java Archive (JAR) files.
Jdb	The Java Debugger.

First Java Program – Hello.java

The following is the smallest possible Java program.

```
01: // Program to print Hello World
02: public class Hello {
03:  public static void main(String args[]) {
04:        System.out.println("Hello World!");
05:  }
06: }
```

The above Java program prints the message *Hello World!,* when you compile and run it. The following are some of the important components of this program.

- Every function in Java must be inside a class, so we have to create a class **Hello** to put the main**()** function in it.
- Function **main**() must be declared as **static**. It must return nothing (void) and it takes an array of type **String** as argument. All these are mandatory.
- **System**.**out**.**println**() is used to print the given message.

Compiling and running a Java program

To compile a java program using JDK, follow the steps given below:

1. Go to command prompt using programs □ accessories □ command prompt.
2. Set the PATH command of Windows to the bin directory of **jdk** as follows at the command line.

```
path c:\jdk\bin
```

PATH is required as *javac.exe* and *java.exe* are in the bin directory.

3. Go to the directory where **Hello**.**java** is saved (c:\jdk\programs).
4. Compile Java program using JAVAC.EXE as follows:

```
javac Hello.java
```

5. Run java program as follows from the command line:

```
java Hello
```

NOTE: PATH command is used to inform Operating System (Windows) where it should search in the file system for executable files. By default, OS searches for executable files only in the current directory. To see the current PATH setting, just type PATH at the command prompt.

Standard or Primitive data types in Java

The table below shows the list of data types available in Java along with the number of bytes occupied by the data type and range of values that can be stored in the data type.

Data Type	Size (Bytes)	Range
byte	1	-128 to 127
short	2	-32768 to 32767
int	4	-2147483648 to 2147483647
long	8	-9223372036854775808 to 9223372036854775807
float	4	3.4e-038 to 3.4e+038
double	8	1.7e-308 to 1.7e+308
boolean		true or false
char	2	Supports Unicode characters

- All numeric types in Java are signed.
- The size of **boolean** is JVM dependent.
- Data type **char** occupies two bytes because it supports Unicode, which needs 16 bits to store characters.
- Unicode supports characters of all languages and some languages of Japanese and Chinese have more than 256 characters.
- The size of data types does NOT change from platform to platform.

NOTE: Java doesn't allow an un-initialized variable to be used. By default, local variables are not initialized by Java.

Rules to create Identifier

A name in the program is called an identifier. The following rules are to be followed while creating identifiers.

- Can contain letters, digits, _ (underscore) and $.
- Must NOT begin with a number. Can start with a currency symbol ($) or _ (underscore).
- Keywords of Java cannot be used.

Valid identifiers	first_number, name$, _first, $amount
Invalid identifiers	2num, total-amount

Keywords in Java

The following is the list of Java keywords. All these keywords are in lowercase.

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

- true and false are boolean literals
- null is null literal
- var, yield, and record are restricted identifiers

Literals

Literal denotes a constant value, which remains unchanged throughout the program.

- Java has three reserved literals – **null**, **true**, **false**.
- Integer literals are treated as integers unless they are suffixed with L (letter L). Example: 100L
- Java 7.0 onwards we can use _ (underscore) as a separator between digits.
- Integer literals can be specified in octal (0123) or hexadecimal (oxff). Octal numbers are prefixed with **0** and hexadecimal numbers are prefixed with **0x**.
- Floating point literals are by default taken as **double**. Use suffix **f** to denote float literal. Example: 10.50f**.**
- Starting from Java 7.0, we can have binary literals prefixed with **0b**.
- Escape sequence as well as Unicode values can appear in string literals. For example, *"How are you ...\t Imaginnovate\u000d"* represents a string that contains a horizontal tab (\t) and carriage return (\u000d in hexadecimal is equal to decimal 13).

Integer	200 -7 1_23_456 0b10001111
Floating point	2.4 -2.5 .5 0.55
Character	'a' '9' ':'
Boolean	true false
String	"one" "123" "work hard"

Escape Sequence Characters

The following are the escape sequence letters supported by Java.

Character	Meaning
\ddd	Three octal digits
\uxxxx	Four hexadecimal digits
\’	Single quote
\”	Double quote
\\	Single backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace

Text Blocks

- A text block is enclosed in """ (three double-quotes) and """ and can contain multiple lines and quotes.
- Special escape sequence **\s** inserts space and **** suppresses new lines.

```
 String text = """
First Line
\s\sSecond Line \
is here!
\s\sThird Line
""";

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

```
First Line
  Second Line is here!
  Third Line
```

Operators in Java

An operator performs a specific operation on the given value(s) called operands. Operators are divided into different categories based on the operation performed by the operator.

Arithmetic Operators

The following table shows arithmetic operators available in Java. They are shown according to the precedence of the operators.

Operator	Meaning
++	Increment
--	Decrement
*	Multiplication
/	Division
%	Modulus
+	Addition
-	Subtraction

Associativity specifies whether operators are grouped from left to right or right to left. Unary ++ and -– operators are having ***right to left*** *associativity* and other operators are having ***left to right*** *associativity*.

Conversion and Type Casting

- Java allows assignment between variables of compatible types.
- Java promotes all integer types such as **byte** and **short** to **int**.
- When we are converting a large data type to a short data type, explicit **type casting** is required.

```
// will FAIL as 10.50 is taken as double
 float f = 10.50;

// Use f as suffix to indicate float
 float f = 10.50f;

// converts double to float and then assigns
 float f= (float) 10.50;
```

Assignment Operator

Assignment operator (=) is used to assign a value to a variable. The value may be either a literal or an expression.

The variable and result of the expression must be of compatible data types. Java also supports multiple assignments, where you can assign a single value to multiple variables in a single assignment.

```
a = 0;
a = b = c = 0;
```

Relational Operators

- All relational operators are binary operators.
- The result of the evaluation is always a boolean.
- Relational operators have ***left to right associativity***.

Operator	Meaning
==	Equal To
>	Greater than
<	Less than
<=	Less than or equal to
>=	Greater than or equal to
!=	Not Equal To

Logical Operators and Short-circuiting

Logical operators combine two conditions and return a boolean value. The following table shows logical operators.

Operator	Meaning
&	AND
|	OR
^	XOR (exclusive or)
||	Short-circuit OR
&&	Short-circuit AND
!	Unary NOT

Short-circuit Logical Operators (&& and ||)

Short-circuit operators check only the necessary conditions and do NOT check both the conditions always. Evaluation of condition stops once the outcome of the condition is known.

&&	Don't check the second condition if the first condition is false.
||	Don't check the second condition if the first condition is true.

```
if (a > b && a > c)...
```

Returns false if condition **a > b** is false and doesn't check the second condition (a > c).

```
if (a == b || a == c) ...
```

Returns true if condition a == b is true and doesn't check the second condition.

Bitwise Operators

Bitwise operators operate on bits in integers. They can be used only with integer data types such as int and long.

Operator	Meaning
~	Ones complement
&	Bitwise Anding
|	Bitwise Oring
^	Bitwise XOR (Exclusive OR)
<<	Shifts bits to left by n times by filling 0 on the right
>>	Shifts all bits right by n times, filling left side bits with sign bit
>>>	Shifts all bits right by n times, filling left side bits with 0

Control Statements

The following are conditional statements and looping structures available in Java.

if statement

The **if** statement is used to decide whether an operation is to be performed. If the condition is true then statement-1 is executed, otherwise statement-2 is executed. The **else** portion is optional.

```
if (condition)
    statement-1;
[else
   statement-2;]
```

NOTE: If multiple statements are to be executed then enclose statements in braces ({}).

It is possible to have nested if statements (an if statement within another if statement). It is also possible to have multiple if statements with if .. else.. if .. else and so on.

Switch statement

The switch statement is used to execute one among many alternatives.

```
switch(expression)
{
   case Label-1: statement; [break];
   [case Label-2 : statement; [break];]...
   [default: statement; ]
}
```

The following example takes the same action for code 1 and 2 and different action for 3, and default is used to take action for the rest of the codes.

```
switch(code) {
  case 1 : // falls through to next case
  case 2 : price = 1000; break;
  case 3 : price = 500; break;
  default: price = 100;
 }
```

For switch statements, the expression must be a **char**, **byte**, **short**, **int** or an **enum** type.

Starting from Java 7.0, switch supports **string** data type also.

```
String county;
// code to get value into country
 switch(country)
 {
     case "India": // code
     case "China": // code
     case "Japan": // code
     default: // code
 }
```

The case constant must evaluate to the same type as the expression. The case constant must be a compile time constant.

NOTE: The default case doesn't have to come only at the end of the switch statement; it can be given anywhere.

The switch expression

- A switch expression, introduced in Java 12, returns a value.
- Unlike switch statements, switch expression returns a value for each case using operator **->**.
- To return a value from a block of statements, we can use yield statements from Java 13.
- Switch expression also supports multiple labels for a single case.

The syntax for **case** changes in switch expression as follows:

```
case Label_1, Label_2,..., Label_n ->
         expression; | throw-statement; | block
```

The following switch returns discount rate based on product code:

```
int disrate =
     switch (code) {
          case 1 -> 10;
          case 2 > 20;
          case 3 > 25;
          default -> 5;
        };
```

Multiple labels

It is possible to use multiple labels for each case by separating them using comma (,).

The following switch expression returns the number of days based on value of month. It uses multiple values for cases.

```
int days = switch (month) {
      case 2 -> 28;
      case 4, 6, 9, 11 -> 30;
      default -> 31;
};
```

Using yield in switch

It is possible to use **yield** to return a value when a case has a set of statements (block).

The following example demonstrates how to use yield in a case when it is not possible to return value with -> alone.

```
int days = switch (month) {
  case 2 -> {
   if (year % 4 == 0 && year % 100 != 0 || year % 400
== 0)
      yield 29;
   else
      yield 28;
    }
   case 4, 6, 9, 11 -> 30;
   default -> 31;
};
```

The while loop

Repeatedly executes statement(s) as long as the condition is true.

```
while(condition)
     statements;
```

Following example displays all numbers from 1 to 10. To execute multiple statements, enclose statements in braces ({}).

```
int i = 1;
 while (i <= 10) {
    System.out.println(i);
    i ++;
}
```

The do .. while loop

Executes the body of the loop and then checks the condition thus guaranteeing the execution of the statement *at least for once*.

```
do {
  statement;
} while (condition);
```

The following program displays numbers from 1 to 10.

```
int i = 1;
 do {
System.out.println(i); i
    ++;
} while (i <= 10);
```

The for loop

The **for** loop executes statements as long as the condition is true. First it executes initialization, then checks whether the condition is true. If the condition is true then execute the statements and then increment the portion.

```
for ( initialization; condition; increment)
    Statement;
```

Variables can be declared within initialization and these variables are available only within the loop.

```
// Variable i is available only within for

for (int i = 1; i <= 10; i ++) {
// body
}
```

The break statement

The **break** statement is used to terminate the loop from inside the loop. Java provides labeled break and unlabeled break. When a break statement is used without any label, it terminates the current loop.

```
break[label]
```

Labeled break terminates the loop that is associated with the given label. In the following example, break terminates 1st loop as label first is associated with **i** loop.

```
01: first:
02: for(int i = 0; ... ){
03:   second:
04:   for (int j = 0; ...) {
05:   for (int k = 0; ...) {
06:       if (cond)
07:          break first;
08:       }
09:    }
10: }
```

The continue statement

Continue is used to stop the current iteration of the loop body and proceed with the next iteration.

```
continue[label]
```

The following code ignores remaining statements in the j loop when the condition is satisfied and starts the next iteration.

```
01: for(int i = 0; i < 10; i ++) {
02:   for(int j = 0; j < 10; j ++) {
03:         if(j < i)
04:             continue;
05:      }
06: }
```

Enhanced for loop

Java provides an enhanced for loop exclusively to process arrays and collections. It takes one value from the array and assigns it to the loop variable. Executes loop statements and repeats the process until the array is exhausted.

```
For ( datatype loopvariable : array)
           statement;
```

The following example displays all elements of an array using an enhanced for loop. The data type of the loop variable and data type of the array must be the same.

```
int a [] = {10,20,30,40,50}:
 for (int n : a)
   System.out.println(n);
```

Printing value using System.out.printf()

Function printf() is similar to the printf() function of the C language. The following are the conversion characters that can be used.

b - boolean, **c** - char, **d** - integer, **f** - floating point, **s** - string.

```
System.out.printf("int %d and float %f \n", i, f);
```

If conversion character and given data do not match then a runtime error occurs. The following will produce a runtime error.

```
System.out.printf("&d",10.50);
```

Reading input using Scanner

Input from users can be taken using Scanner class. The following snippet shows how to use Scanner to read an integer from a user.

```
Scanner s = new Scanner(System.in);
 int n = s.nextInt();
```

The following program reads a number from a user using Scanner and displays its factors.

```
01: import java.util.Scanner;
02:
03: public class Factors {
04:   public static void main(String[] args) {
05:       Scanner s = new Scanner(System.in);
06:       System.out.print("Enter a number :");
07:       //read an int from keyboard
08:       int num= s.nextInt();
09:
10:       for (int i = 2; i <= num/2; i ++) {
11:            if (num % i == 0)
12:                 System.out.println(i);
13:       }
14:    }
15: }
```

The **import** statement is required to make **Scanner** available to the current program.

Method	Description
nextInt()	Reads next integer
nextDouble()	Reads next double
nextLine()	Reads next line and returns as a string

Creating and using Arrays

- Array is a collection of elements.
- All elements of the array are of the same type.
- All elements are commonly referred to by a single name and individual elements are accessed using index.
- An array in Java is dynamic. We specify the size of the array at runtime.

```
int a1[];      //declare an array
int []a2;     //another syntax to declare an array

// Create an array of 10 elements

a1 = new int[10];
```

Alternatively, you can declare and allocate memory in one step.

```
// Declare and create an array
int a1[] = new int[10];

/* create an array where size of the array is
determined by the value of variable n */

int a2[] = new int[n];
```

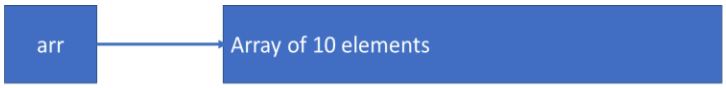
Getting array's length using length property

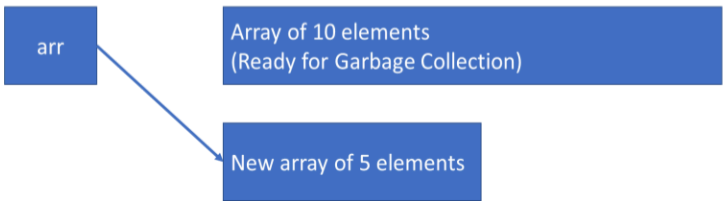
Property **length** of an array can be used to get the number of elements in an array.

```
System.out.printf("Length: %d", a.length);
```

An array name can be used to refer to an array of different sizes at runtime. The following snippet shows how array name **arr** points to an array of 10 elements first and then points to an array of 5 elements.

```
// create an array
int arr[] = new int[10];

//Array arr points to a new array of 5 elements
arr = new int[5];
```





Old array of 10 elements referred by **arr** is now eligible for garbage collection by Java.

NOTE: Java virtual machine has a garbage collector, which takes care of releasing unused memory blocks.

You can create a double dimensional array using two square brackets as follows:

```
Int a[][] = new  int[5][5];
```

Initializing an array

Java allows an array to be initialized. The following examples show how you can initialize an array in Java.

```
// initializes array a to given three values
int a[] = {10,20,30};
// initializes array b to given five values
int b[] = new int[] {10, 20, 30, 40,50};

// initializes double dimensional array - 2 X 3
int bb[][]= {{1,2,3},{4,5,6}};
```

NOTE: Elements of an array are automatically initialized to default value according to the data type of array.

Variable Number of Arguments

- Java allows a method to take a variable number of arguments.
- The argument is declared with **...** (ellipses) after data type and before the name.
- Must be the last argument in a method.
- Argument is treated as an array inside the method.

```
01: public class VaryingArguments {
02:   public static void main(String[] args) {
03:         System.out.println(getSum(10, 20,30));
04:         System.out.println(getSum(100, 200));
05:   }
06:   public static int getSum(int... nums) {
07:         int total = 0;
08:   for (int n : nums) {
09:   total += n;
10:   }
11:   return total;
12:   }
13: }
```

Command-line Parameters

- Parameters that are passed at the time of invoking a java program from the command line are called as command line parameters.
- These parameters can be used to provide input to the program.
- These parameters are accessible from ***main()*** through the **args** parameter, which is an array of strings.
- The ***length*** attribute of ***args*** returns the number of parameters passed on the command line.

```
01: // program to display command-line parameters
02: public class CLA {
03:  public static void main(String args[]) {
04:    System.out.printf("Args Count: %d\n",
args.length);
05:      // display parameters
06:       for(Strings: args)
07:       System.out.println(s);
08:   } //end of main
09: }
```

Call the previous program from command line as shown below:

```
c:\jdk\imaginnovate>java CLA One Two <Enter>
 Args Count: 2
 One
 Two
```

```
01: public class CLA {
02:   public static void main(String a[]) {
03:         for (String s : a)
04:            System.out.println(s);
05:   }
06: }
```

NOTE: Method main() must have a parameter of type String[]. However, the name of this parameter could be anything. The following code is the same as the code above.

Program - Prime.java

This program checks whether the number given on the command line is a prime number or not. Prime number is a number which has no factors other than 1 and itself.

```
01: // program to check whether a number is prime
02: public class Prime {
03:    public static void main(String[] args) {
04:         // check whether number is passed
05:         if (args.length == 0) {
06:            System.out.println("Number is missing!");
07:            return;
08:         }
09:         // convert args[0] to int
10:         int n = Integer.parseInt(args[0]);
11:         boolean prime = true;
12:
13:         for (int i = 2; i <= n/2; i ++) {
14:           if (n % i == 0) {
15:              prime = false;
```

```
16:             break;
17:           }
18:         } //end of for
19:
20:         if (prime)
21:           System.out.println("Prime Number");
22:         else
23:           System.out.println("Not a prime number");
24:     } //end of main
25: }
```

Prime program does the following:

1. It checks whether a command line argument is given. If not, it terminates **main**() by using return statements.
2. Converts first command line argument to an int.
3. Sets a loop that runs from 2 to n/2.
4. If a number in the range 2 to n/2 can divide the number without any remainder then it breaks the loop after setting the prime variable to false.
5. If no number in the range 2 to n/2 can divide a number without any reminder then the loop terminates but the prime variable's value remains true.
6. At the end it displays a message based on the value of the prime variable.

Object Oriented Programming

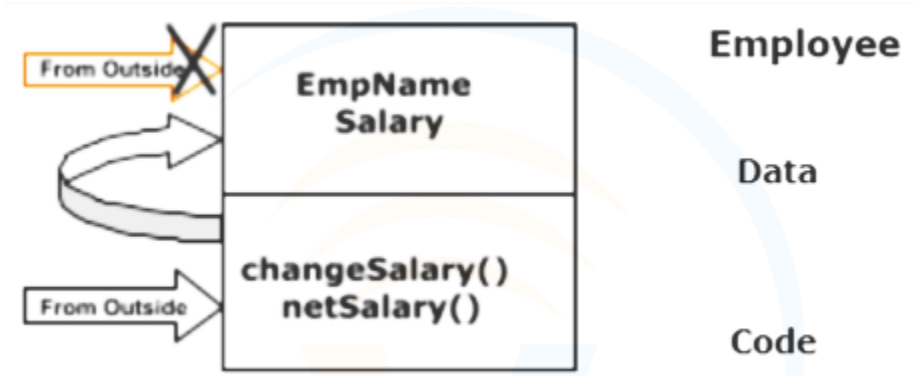
- Java is an object-oriented programming language. So, it is important to understand what object-oriented programming is.
- OOP allows programs to be divided into objects, where each object contains data and code to process data.
- These objects of the program map to real-world objects like an account in a bank, a student in college etc.

Three basic principles of OOP are:

- **Encapsulation**
- **Inheritance**
- **Polymorphism**

Encapsulation

- Binds or encapsulates data and code that processes data.
- Protects instance variables (data) by not allowing them to be accessed from outside the object.
- Provides a set of functions (methods) that are accessed from other objects.
- Increases control as each object is independent with its data, which is private, and a set of functions, which can be called from other objects.
- A class in Java defines data and code that is to be encapsulated.



In the above example the class **Employee** has two instance variables – EmpName and Salary and two methods – changeSalary() and netSalary().
Instance variables can be accessed only by methods that are part of the class – changeSalary() and netSalary().

Class vs. Object

Class is the description of a collection of similar objects. Class defines the instance variables and methods to be encapsulated.

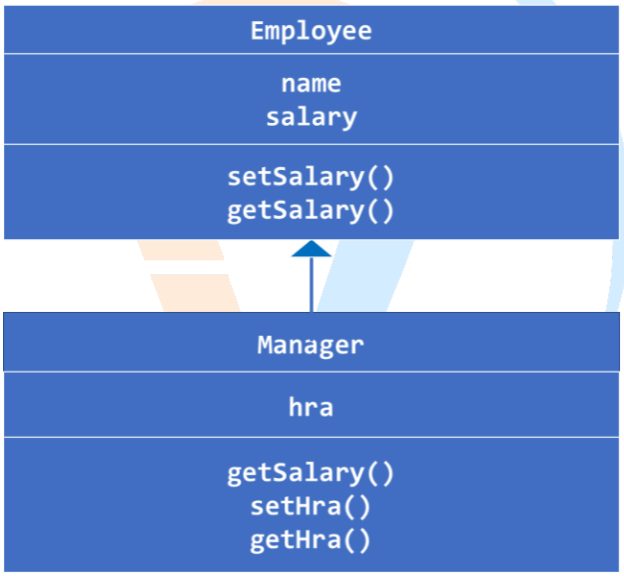
Class denotes a category of objects. It is a blueprint for creating objects. Example: Employee, Product, Student etc.

Instance variables are used to store data and **methods** perform operations on the data.

Object is an instance of the class. Memory is allocated only when an object of the class is created. Methods are called using an object. Example: An employee, a student etc.

Inheritance

- Allows a class to be created from another class thereby inheriting all members of another class.
- Allows reusability of existing classes.
- New classes may add new members and override existing members of the inherited class.
- Class being inherited is called a superclass and the class created from the existing class is called a subclass.
- Inheritance represents "IS A" relationship between classes. For example, Manager is extending Employee because Manager is an Employee. A SavingsAccount extends Account because SavingsAccount is an Account.
- Hierarchies can be built using inheritance



In the above example, **Manager** class inherits all the data members and methods of **Employee** class. It adds **hra** and **setHra**() members and overrides **getSalary**() method.

Overriding is the process where the subclass (Manager) is creating a method with the same signature as a method in the superclass (Employee).

Java allows a class to extend only one class – **single inheritance**.

Multiple inheritance, where a class can be created from multiple classes, is NOT permitted in Java.

Polymorphism

- Allows multiple methods performing the same operation to have the same name.
- Compiler invokes one of the multiple methods depending upon the context (parameters passed to method).
- Programmers need not remember multiple names.

In the following example, all three functions returning a maximum of two values are given the same name.

```
max(int ,int)
max(long, long)
max(String,String)
```

Depending upon which data type is passed as the parameter to **max** function, one of the three functions is invoked by the compiler.

Polymorphism may be implemented either at compile-time (method overloading) or at run-time (dynamic method dispatch).

Creating a Class

A class is like a user-defined data type. A class describes data to be stored and processed. The following is the syntax used to create a class.

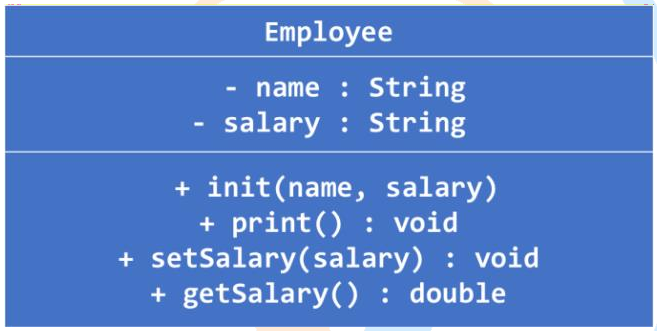
```
access class classname
 {
   [access] instance-variable-declaration;
   [[access instance-variable-declaration] ] . . .

   [access] returntype methodname ([parameterlist])
   {
      // body of the method
   }
   .
```

.

Access may be *private*, *public*, *protected* or *default* (no access) for instance variables and methods.

NOTE: Generally, class names are created using Pascal case naming convention - first letter of each word in uppercase and remaining in lowercase. Method names are created using Camel case naming convention – first word in lowercase and first letter of remaining words in uppercase and remaining letters in lowercase.

The following is the **UML** (Unified Modeling Language) diagram to represent the Employee class. Minus (-) sign denotes **private** access and plus (+) sign denotes **public** access.



NOTE: Name of .java file and public class name must be the same. A single .java file can have multiple classes (but only one of them can be public). When you compile. java file, each class in the file will have a corresponding .class file.

Employee.java

This file contains definitions for class Employees.

```
01: public class Employee {
02:   private String name;
03:   private int salary;
04:   public void init(String n, int sal) {
05:         name = n;
06:         salary = sal;
07:   }
```

```
08:   public void print() {
09:   System.out.println(name);
10:   System.out.println(salary);
11:   }
12:   public void setSalary (int sal) {
13:         salary = sal;
14:   }
15:   public int getSalary() {
16:         return salary;
17:   }
18: }
```

TestEmployee.java

This file contains the definition for class TestEmployee. Function main() in TestEmployee does the following:

- Creates an object reference of **Employee** class.
- Creates an object of Employee class.
- Calls methods of Employee class using object reference.

```
01: public class TestEmployee {
02:     public static void main(String[] args) {
03:       Employee e;         //object reference
04:       e = new Employee(); //create an object of
Employee
05:
06:       e.init("Jonathan", 35000);
07:       e.print();
08:       e.setSalary(55000);
09:
10:       if (e.getSalary() > 50000)
11:           System.out.println("Fat salary");
12:  }
13: }
```

If you compile **Employee.java**, **TestEmployee.java** and run, it generates the following output:

```
Jonathan
```

```
 3500
Fat salary
```

Object Reference Object

Constructor

- A method in the class with the same name as the class name.
- Invoked automatically whenever an object of the class is created.
- Used to initialize instance variables of the class.
- Doesn't contain any return type – not even void.

```
01: public class Number {
02:   private int num;
03:   // constructor
04:   public Number() {
05:         num = 1;
06:   }
07: }
```

In the above, **Number** class has a constructor that is used to initialize instance variable *num*. When an object of the **Number** class is created, Java automatically calls the constructor.

```
Number n = new Number(); // calls constructor
```

Constructors can take parameters and use those parameters for initializing instance variables.

```
01: public class Number {
02:   private int num;
03:   public Number(int n) {
04:         num = n;
05:   }
06: }
```

The above constructor is called whenever an object is created.

```
Number n = new Number(10);
```

NOTE: If no constructor is explicitly created then Java creates a constructor with no parameters (default constructor). But Java doesn't provide this constructor once a user-defined constructor is created in the class.

The following is **Employee** class with constructor instead of **init**() method :

```
01: class Employee {
02:   private String name;
03:   private int salary;
04:   public Employee (String n, int sal) {
05:         name = n;
06:         salary = sal;
07:   }
08:   // remaining code
09: }
```

```
01: class UseEmployee {
02:   public static void main(String[] args) {
03:         Employee e = new Employee("Jonathan",
25000);
04:   }
05: }
```

Overloading Methods

- It is possible to define two or more methods with the same name within the same class provided the parameters of the methods are different.
- Overloading methods is one way of implementing polymorphism.
- Methods can be overloaded based on type/number of parameters.
- Methods cannot be overloaded based on return type.

The following example demonstrates overloading methods in a class.

```
01: class Test {
02:    void m1(int n) { }
03:    void m1(String s) { }
04:    void m1(String s, int n) { }
05:    void m1(int n, String s) { }
06:    // Not allowed as methods differ only by return
type
07:    int m2 (int n) {}
08:    long m2 (int n) {}
09: }
```

Overloading Constructors

Just like how it is possible to overload methods of a class, it is also possible to overload constructors of the class.

```
01: public class Number {
02:    int num;
03:    public Number() {
04:          num = 1;
05:    }
06:    public Number(int n) {
07:    num = n;
08:    }
09: }
```

First constructor is called when an object is created without passing any parameter and second constructor is called when an object is created by passing a single **int** as parameter.

```
Number n1 = new Number();  //calls constructor Number()
Number n2 = new Number (100);    // calls Number(int)
```

NOTE: It is NOT possible to create an object unless Java can invoke one of the constructors.

```
01: class A {
02:   int n;
03:   public A(int num) {
04:         n = num;
05:   }
06:   public A(String s) {
07:     n=Integer.parseInt(s);
08:   }
09: }
```

Third line doesn't compile as Java cannot call any constructor.

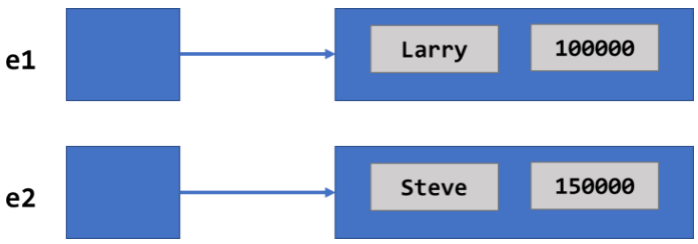
```
A  obj1 = new A(10);
A  obj2 = new A("99");
A  obj2 = new A();    // not possible
```

In order to create an object without passing any parameter at the time of creation, the class must have a constructor that takes no parameter or it must have a default constructor.

Object Reference

Object reference is a variable that references an object. An object in Java is accessed using object reference.

```
Employee e1; // create object reference of Employee
class
// create an object and make e1 referencing it
e1 = new Employee("Larry", 100000);

//create object reference and object
Employee e1 = new Employee("Larry", 100000);
Employee e2 = new Employee("Steve", 150000);
```



Comparing object references

If two object references of the same class are compared, only references are compared but NOT contents of the objects.

```
Employee e1 = new Employee("Jason", 25000);
 Employee e2 = new Employee("Jason", 25000);
if (el == e2) // only references are compared, NOT
objects
     System.out.println("Equal");
```

NOTE: If you compare two object references, only the references are compared and not the contents of the object that they point to.

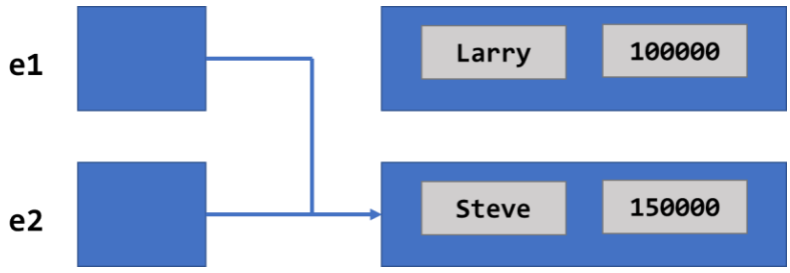
NOTE: Only == and != relational operators are allowed with object references.

Assignment between two object references

It is possible to assign one object reference to another. But what we effectively copy is not the contents of an object to another, instead one object reference to another.

The following example and picture demonstrate what happens when an object reference is copied to another.

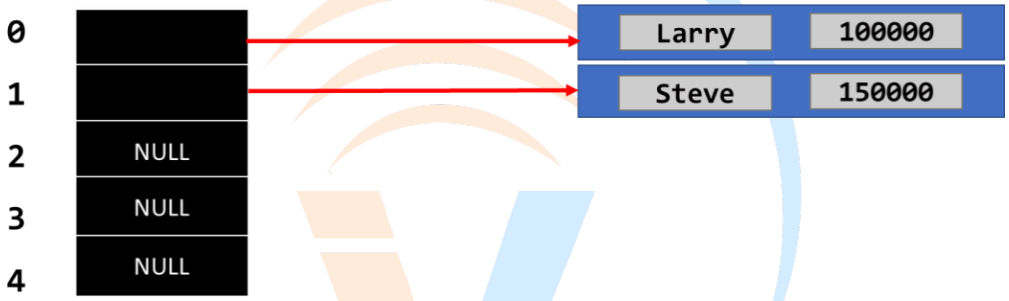
```
Employee e1, e2;

e1 = new Employee("Larry",100000);
e2 = new Employee("Steve",150000);
e1 = e1; // e1 and e1 point to same object
```



Array of objects

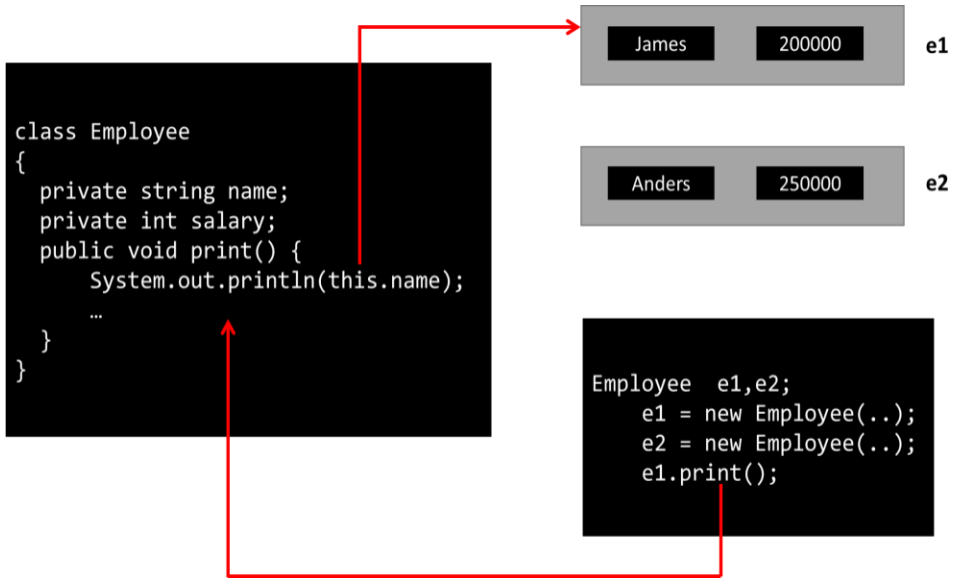
- Java supports arrays of objects of a class.
- When an array of objects is created only object references that can point to objects of class are created.
- The default value of each element is null and object reference must be made to point to an object explicitly before it is used.

```
Employee [] e;
// Create an array of  Employee object
references e = new Employee[5];
e[0] = new Employee("Larry",100000);
e[1] = new Employee("Larry",150000);
```



The this reference

- Keyword ***this*** references invoking object.
- Used by methods to implicitly access members of the calling object.
- Used to access instance variables when parameters and instance variables have the same names.
- The **this** reference is made available to each non-static method automatically.
- Used to call a constructor from another constructor.



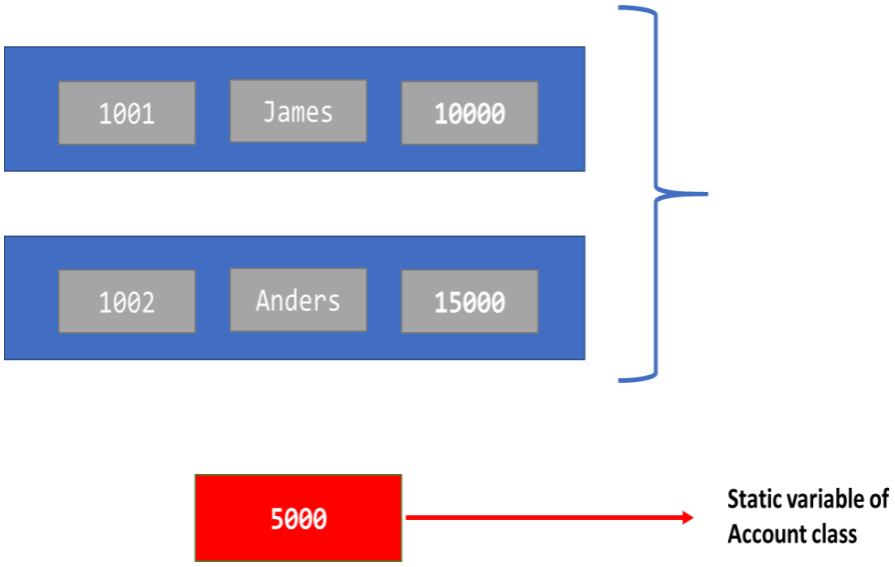
The following constructor uses this reference to access instance variables as parameters and instance variables have the same names.

NOTE: When instance variables and parameters are having the same name, parameters of the method take precedence over instance variables. In general, local variables always take precedence over non-local variables.

```
01: class Employee {
02:       private String name;
03:       private int salary;
04:       public Employee (String name, int salary) {
05:           this.name = name;
06:           this.salary = salary;
07:       }
08: }
```

Static Variables

- A static variable, otherwise known as class variable, is a variable that is associated with class and not an object.
- Defined using keyword **static.**
- Exist only for once for the entire class.
- Static variables that are declared as **final** are used as constants.



Static Methods

- Static methods are the same as normal methods except that they can be invoked with class names and need not be invoked by any object.
- Defined using keyword **static**.
- Do not have **this** reference.
- Static methods cannot invoke non-static methods and cannot access instance variables as they do not have access to any specific object (this reference) – refer to code below.
- Generally used to access and manipulate static variables.

```
01: public class StaticTest {
02:   private int iv;
03:   private static int sv;
04:   public static void main(String[] args) {
05:         iv = 10; // not valid
06:   StaticTest obj = new StaticTest();
07:   obj.iv = 10; //valid as we use object to access iv
08:   }
09: }
```

Why is main() a static method?

Function **main**() must be declared as static as it is called using the class name by JVM. For example, when you invoke

```
Java Hello
```

It will be converted to a call to main as follows:

```
Hello.main(parameter)
```

Other examples for static members are **Integer**.**parseInt**(), **Math**.**abs**() and **System**.**out**.

NOTE: Java does not allow any function to be defined outside the class. Methods that can be used without creating an object are declared static, so that they can be called using class names. This also saves the time taken to create and release an object.

	Variable	
Method	**Static**	**Instance**
Instance	✓	✓
Static	✓	✗

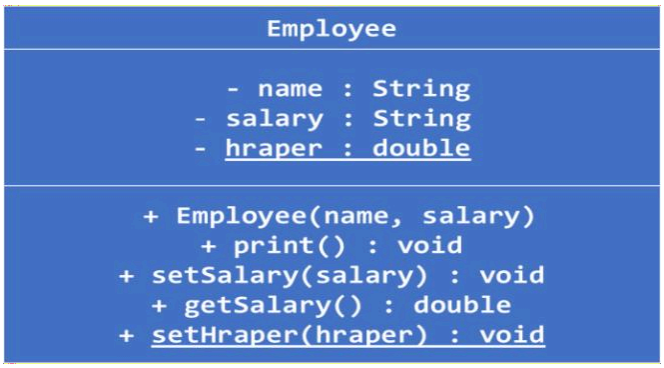
Final Variable

- Java allows variables to be declared as final using the final keyword.
- Final variables cannot change value once they are assigned a value.
- Final variables are generally static variables as it generally makes no sense to have the same value for a variable in each object of the class.
- Final variables must be assigned a value before the constructor of the object is completed.

```
Class Account {
  public static final int MINBAL = 500;
}
```

Variable MINBAL is accessed using class name from outside the class as it is a static and public variable. Minbal cannot be changed as it is final and initialized to 500.

```
System.out.println(Account.MINBAL);
```



In the above UML diagram, static members – **hraper** variable and **setHraper**() method are ***underlined*** to indicate that they are static members of the class.

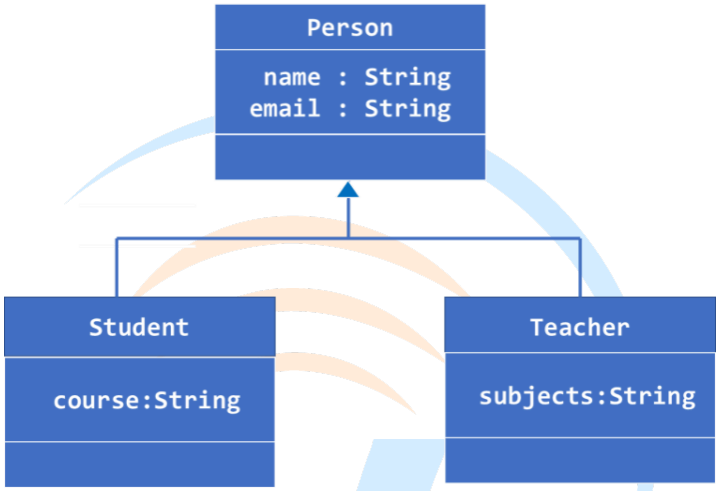
The following code shows how to implement Employee class with static variables – **hraper** and static method **setHraper**().

```
01: public class Employee {
02:   private String name;
03:   private double salary;
04:   private static double hraper = 20;
05:   public Employee (String name, double salary) {
06:         this.name = name;
07:         this.salary = salary;
08:   }
09:   public void setSalary (double salary) {
10:         this.salary salary;
11:   }
12:   public double getSalary() {
13:         return this.salary +
14:   this.salary * Employee.hraper / 100;
15:   }
16:   public static void setHraper (double hra) {
17:         Employee.hraper = hra;
18:   }
19: }
```

```
01: public class UseEmployee {
02:   public static void main(String[] args) {
03:     Employee e = new Employee("Jason", 30000);
04:     System.out.println(e.getSalary());
05:     Employee.setHraper (25);
06:     System.out.println(e.getSalary());
07:   }
08: }
```

Inheritance

- Allows a new class to be created from an existing class.
- Keyword **extends** is used to create a new class that extends an existing class.
- Existing class is called a superclass and the new class is called a subclass.
- Inheritance represents **Is A** relationship. For example, Student is a Person, Car is a Vehicle etc.

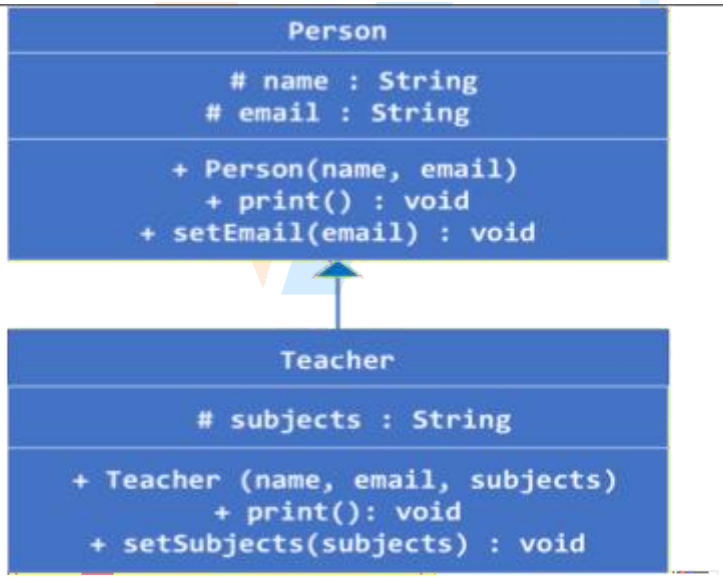


The following code demonstrates creating **Person** class and **Teacher** classes.

```
01: class Person {
02:   protected String name, email;
03:   public Person(String name, String email) {
04:         this.name = name;
05:         this.email = email;
06:   }
07:   public void print() {
08:   System.out.println(this.name);
09:   System.out.println(this.email);
10:   }
11:   public void setEmail (String email) {
12:         this.email = email;
13:   }
14: }
```

NOTE: Access **protected** allows the members to be accessed from subclasses also.

```
01: class Teacher extends Person{
02:   protected String subjects;
03:   public Teacher(String name,String email,
04:                          String subjects) {
05:     super(name,email); //must be first statement
06:         this.subjects = subjects;
07:   }
08:   @Override
09:   public void print() { //overrides print() of
Person
10:         super.print();
11:         System.out.println(subjects);
12:   }
13:   public void setSubjects(String subjects) {
14:           this.subjects = subjects;
15:   }
16: }
```



NOTE: Symbol # in the UML diagram specifies protected access.

```
01: class TestPerson {
02: public static void main(String args[]) {
03: Teacher t = new Teacher ("Kavitha",
04: "testmail@gmail.com", "Java, .Net");
05: t.print();
06: t.setEmail("testmail@gmail.com");
07: t.setSubjects ("Java, .Net, Python, DS");
08: }
09: }
```

Constructors in Inheritance

When an object of a subclass is created then Java automatically calls the default constructor of the super class from the constructor of the subclass.

But, when there is no default constructor in superclass, we must explicitly call the constructor of superclass using **super** keyword and required parameters (as we did in **Teacher** class).

Overriding Methods

When a method in the subclass has the same name and signature of a method in superclass, the method in subclass is said to be overriding the method in superclass.

Method **print**() of **Person** class is overridden in subclass **Teacher**. **Rules related to overriding**:

- The arguments list must match exactly. Otherwise, it becomes overloaded.
- Return type must be the same as, or a subclass of, the return type of the original method.
- Access level CANNOT be more restrictive than the original. For example, if the superclass method is public, then the subclass method cannot be protected.
- A final method cannot be overridden.
- Static method cannot be overridden.
- Only inherited methods can be overridden. Private methods of superclass cannot be overridden as they are not inherited.

@Override Annotation

Annotation **Override** indicates that a method declaration in a subclass is intended to override a method declaration in a superclass.

If a method annotated with this annotation does not override a superclass method, the compiler generates an error message.

Overloading vs. Overriding

When you create a method in a subclass with the same name as a method in a superclass but with a different set of parameters, it is called overloading and not overriding.

The following example shows the difference between overriding and overloading.

```
01: class C1{
02:   public void m1(String msg) {
03:   }
04:   public void m2 (int n) {
05:   }
06: }
07:
08: class C2 extends C1 {
09:     // overloads m1() in class C1
10:     public void m1() {
11:     }
12:
13:     // overrides m2() in class C1
14:     public void m2(int n) {
15:     }
16: }
```

Using super keyword to access superclass version

The **super** keyword in subclass is used to access superclass. The super keyword can be used to:

- Access a method of superclass that is overridden by a member in the subclass.
- Invoke the constructor of super class from the constructor of subclass.

- The super keyword can refer only to the immediate super class in the hierarchy.

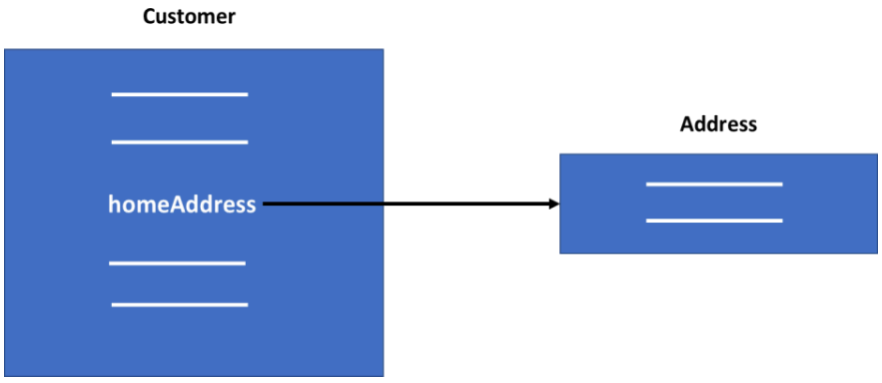
NOTE: While calling the constructor of the superclass from subclass's constructor, the call must be the first statement in the constructor of the subclass.

Has A" Relationship

When a class contains a variable that refers to another class, the class is said to have a "Has A" relationship with the other class.

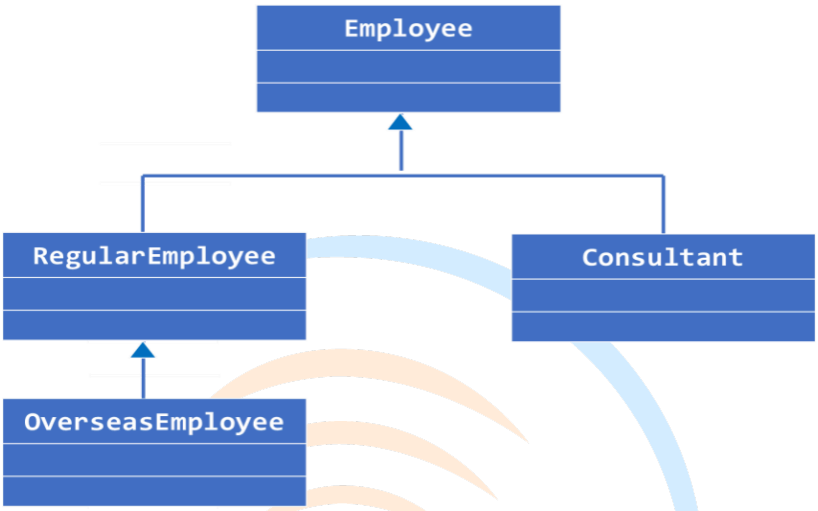
In the following example, **Customer** class has "has a" relationship with **Address** class as it has a variable of class **Address**.

```
class Address {
// instance variables & methods
 }
class Customer {
     private Address homeaddress;
// other variables and methods
 }
```



Multi-level inheritance

Java supports multi-level inheritance. In this, a class is extended by another class, which is again extended by another class. The following picture shows multi-level inheritance.



In the above example, **Employee** class is inherited into **RegularEmployee** class, which in turn is inherited into **OverseasEmployee**.

Constructors of classes in inheritance must be called in the order of inheritance.

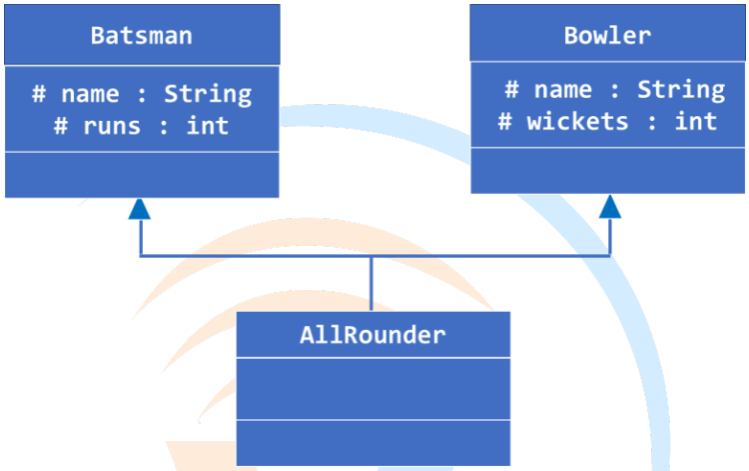
When you create an object of **OverseasEmployee**, the constructor of the Employee class is called first then constructor of **RegularEmployee** and then finally constructor of **OverseasEmployee**.

Java tries to call the constructor of a superclass from the subclass's constructor using the super() statement. However, if that is not adequate, you must explicitly call the constructor of superclass using super keyword by passing required parameters.

Multiple Inheritance

Java **does NOT** support multiple inheritance, where a single class extends two or more classes. It means, Java doesn't allow two or more classes to be inherited into a single class.

In the following diagram, **AllRounder** is extending **Batsman** and **Bowler**, which is not supported by Java.



The reason for Java to avoid multiple inheritance is the ambiguity caused by multiple inheritance. For example, **AllRounder** has two instances of the name variable as one comes from **Batsman** and another comes from **Bowler**. This leads to *ambiguity*.

Runtime Polymorphism

- It is the process by which a call to an overridden method is resolved at runtime instead of compile-time.
- It is based on the concept that an object reference of the super class can refer to an object of any subclass in the hierarchy.
- Runtime polymorphism occurs when an overridden method is called using an object reference of superclass.
- Java decides which method to call (superclass or subclass) based on the object pointed by object reference of superclass.
- Also called as ***late binding*** or ***dynamic method dispatch***.

```
class Person {
  public void print() {
          . . .
     }
 }
class Teacher extends Person {
   @Override
 public void print() {
          . . .
    }
 }
```

The following code shows how Java calls appropriate methods depending on the object pointed by object reference.

```
Person p;
p = new Person(...);
p.print();

// object ref. of superclass points to object of
subclass p = new Teacher (...);
/* call to print() method calls print() of subclass as p
points to object of subclass. */
p.print(); // calls print() of subclass
```

NOTE: Dynamic method dispatch is applied only to methods that are defined in superclass, overridden in subclass and invoked using an object reference of superclass.

Abstract Method

- Keyword **abstract** is used to create abstract methods.
- Abstract method doesn't contain the body.
- Must be overridden in subclass. Otherwise, subclasses must be declared abstract.
- If there is any abstract method in the class then class must be declared as abstract.
- Mutually exclusive with final method – a method cannot be declared as final and abstract.

Final Variable, Method, Class and Parameter

- Keyword **final** is used to declare final variables, methods and classes.
- A final variable is a variable whose value cannot be changed once the variable is assigned a value.
- Final method is a method that cannot be overridden in the subclass.
- Final class cannot be extended (inherited). There is no subclass for final class.
- You can declare parameters as **final** to ensure that the value of the parameter doesn't change during the execution of the method.

```
public void fun(int x, final int y) {}
```

The following example demonstrates how to use abstract method, abstract class, final method and runtime polymorphism.

```
01: abstract class Employee {
02:     protected String name, desg;
03:     public Employee (String name, String desg) {
04:         this.name = name;
05:         this.desg = desg;
06:     }
07:     final public String getDesg() {
08:         return desg;
09:     }
10:     final public void setDesg(String desg) {
11:         this.desg = desg;
12:     }
13:     final public String getName() {
14:         return name;
15:     }
16:     public abstract int getPay();
17: }
18: class RegularEmployee extends Employee {
19:   protected int salary;
20:   public RegularEmployee(String name, String desg,
21:         int salary) {
22:   super(name, desg);
23:         this.salary =salary;
24:   }
```

```
25:   @Override
26:   public int getPay() {
27:         return salary;
28:   }
29: }
30:
31: class Consultant extends Employee {
32:   protected int nohours, hourrate;
33:   public Consultant (String name, String desg,
34:         int nohours, int hourrate) {
35:    super(name, desg);
36:        this.nohours =nohours;
37:   this.hourrate=hourrate;
38:   }
39:   @Override
40:   public int getPay() {
41:      return nohours*hourrate;
42:   }
43: }
44: public class TestEmployee {
45:   public static void main(String[] args) {
46:         Employee employees [] =
47:   {new RegularEmployee("Steve", "Programmer",
25000),
48:         new Consultant ("Kevin", "DBA", 10,500),
49:   new RegularEmployee("Tom", "System Analyst",
45000)
50:   };
51:   for (Employee e: employees)
52:   System.out.printf("%s:%d\n",
53:   e.getName(), e.getPay());
54:   }
55: }
```

```
Steve:25000
 Kevi:5000
 Tom:45000
```

Initialization Blocks

- Object initialization block is executed when an object is created.
- Static initialization block is executed when the class is loaded.
- Initialization block is run after the super-constructor has run.
- Blocks are run in the order in which they appear in the class.
- Initialization blocks cannot have parameters and cannot return any value.
- Object initialization blocks are executed irrespective of which constructor is executed. So, they can be used to contain code that all constructors need to share.

```
01: class A {
02:     // Object initialization block
03:     {
04:         System.out.println("Init Block In A");
05:     }
06:     public A() {
07:         System.out.println("Constructor of A");
08:     }
09:
10:     // Static initialization block
11:     static {
12:         System.out.println("Static Init Block Of A");
13:     }
14: }
15:
16: class B extends A
17: {
18:     public B() {
19:         System.out.println("Constructor of B");
20:     }
21:     public B(String message) {
22:         System.out.println("Constructor with message: "
23:                                                 + message);
24:     }
25:     // Object initialization block
26:     {
27:          System.out.println("First Init Block In B");
28:     }
29:
```

```
30:     {
31:          System.out.println("Second Init Block In B");
32:     }
33:
34:     static {
35:        System.out.println("Static Init Block Of B");
36:    }
37: }
38:
39: public class InitBlockDemo {
40:   public static void main(String[] args) {
41:         B obj = new B();
42:    System.out.println("=========================");
43:   B obj2 = new B("Second Object");
44:   }
45: }
```

When you run the above program, you will get the following output:

```
Static Init Block Of A
 Static Init Block Of B
 Init Block In A
 Constructor of A
First Init Block In B
 Second Init Block In B
 Constructor of B
 ============================
Init Block In A
 Constructor of A First
 Init Block In B Second
 Init Block In B
Constructor with message: Second Object
```

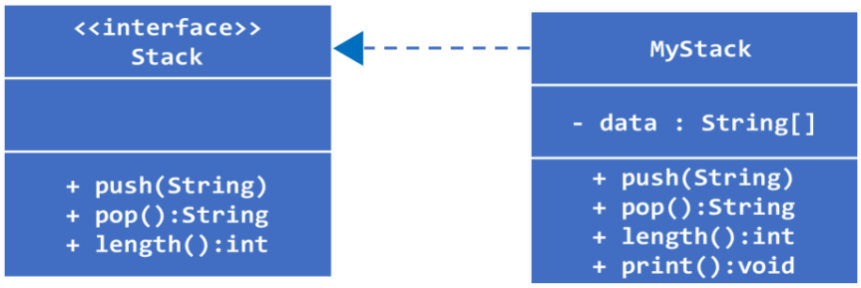
Interfaces

- An interface is a collection of methods that must be implemented by the implementing class.
- An interface defines a contract regarding what a class must do, without saying anything about how the class will do it.
- Interfaces can contain declaration of methods and variables.
- Implementing class must define all the methods declared in the interface.
- If a class implements an interface and does not implement all the methods then the class itself must be declared as abstract.
- Variables in interface automatically become static and final variables of the implementing class.
- Members of interface are implicitly public, so need not be declared as public.

```
access interface name [ extends interface [,interface] ]
{
   returntype method([parameters]);
   datatype finalvariable = value;
}
```

NOTE: In UML, the interface resembles a class. One way to differentiate an interface from a class is to use **<<interface>>** stereotype as shown above.

The following example shows how to declare an interface and how a class implements the interface.




```
interface  Stack {
void push(String value);
String pop();
int length();
 }
```

Stack is a data structure that stores data in such a way that the last value pushed is the first value to be popped (LIFO).

Implementing an Interface

The following class - **MyStack**, implements **Stack** interface and provides definition for methods in the interface. It also adds a new method – print() to print all values in the stack.

```
01:  class MyStack implements Stack {
02:  private String data[] = new String[10];
03:  private int top = 0;
04:  public void push(String value) {
05:        data[top] = value;
06:  top++;
07:  }
08:  public String pop() {
09:        top --;
10:  return data[top];
11:  }
12:  public int length() {
13:        return top;
14:  }
15:  public void print() {
16:  for (int i = 0; i < top; i ++) {
17:        System.out.println(data[i]);
18:  }
19:  }
20: }
```

```
01:  class MyStack implements Stack {
02:  private String data[] = new String[10];
03:  private int top = 0;
04:  public void push(String value) {
05:        data[top] = value;
06:  top++;
07:  }
08:  public String pop() {
```

```
09:         top --;
10:   return data[top];
11:   }
12:   public int length() {
13:         return top;
14:   }
15:   public void print() {
16:   for (int i = 0; i < top; i ++) {
17:         System.out.println(data[i]);
18:   }
19:   }
20: }
```

The following **main**() creates an object of **MyStack** class and calls its methods.

```
01: class UseMyStack {
02:    public static void main(String[] args) {
03:          MyStack ms = new MyStack();
04:    ms.push("First");
05:    ms.push("Second");
06:    ms.print();
07:    System.out.println(ms.pop());
08:    }
09: }
```

NOTE: A class can neither narrow the accessibility of an interface method nor specify new exceptions in the method's ***throws*** clause. For example, a method declared in interface cannot be overridden with any access narrower than public as it is implicitly declared as public in interface.

Using object reference of an Interface

An object reference of an interface can refer to an object of its implementing class or any subclass of implementing class.

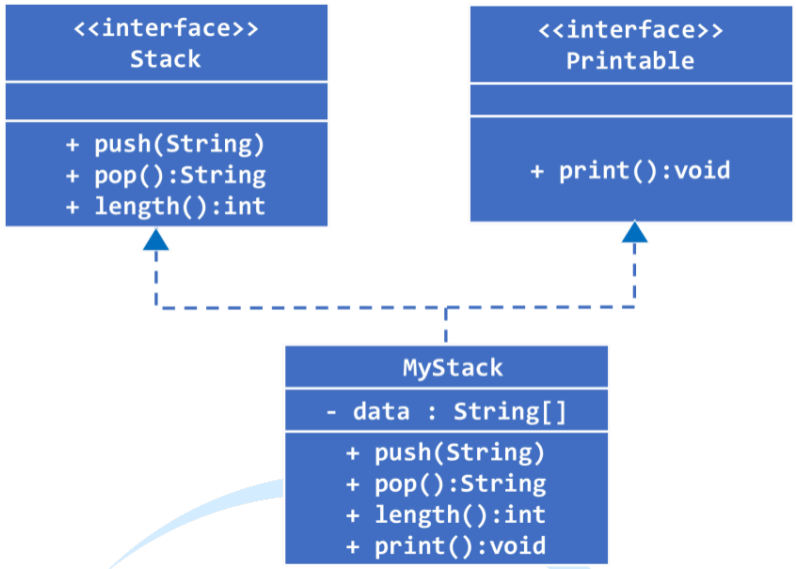
For example, an object reference of **Stack** interface can refer to an object of **MyStack** class as it implements Stack interface. However, only methods that are present in the interface can be called using object reference of interface even though the reference refers to the object of implementing class.

The following code snippet shows how to use object reference of Stack interface.

```
Stack s;
s = new MyStack();
 s.push("10");
 System.out.println(s.pop());
// Cannot call print() as it is NOT a method of
interface
 Stack
s.print();
```

NOTE: An object reference of an interface can reference an object of implementing class. However, only methods in interface can be called using object reference of interface.

A single class can implement multiple interfaces. *Multiple inheritance* is supported using interfaces.




```
interface  Printable {
    void print();
}
class Mystack implements Stack, Printable {
...
}
```

An Interface extending another

- An interface can extend **one or more** interfaces.
- A class implementing the interface must define all the methods in the entire inheritance chain.
- The interface being extended is called a superinterface and the one extending is called a subinterface.

```
interface A {
   void m1();
}
```

```
interface B extends A {
   void m2();
}
class c1 implements B {
// implements both m1 and m2
 }
```

Default Methods in Interface

- Default methods enable you to add new functionality to the interfaces and ensure binary compatibility with code written for older versions of those interfaces.
- You specify that a method definition in an interface is default with the **default** keyword at the beginning of the method signature.
- All method declarations in an interface, including default methods, are implicitly public, so you can omit the public modifier.
- When you extend an interface that contains a default method, you can do the following:
  - ✔ Not mention the default method at all, which lets your extended interface inherit the default method.
  - ✔ Redeclare the default method, which makes it abstract.
  - ✔ Redefine the default method, which overrides it.

Static Methods in Interface

- You can define static methods in interfaces using **static** keyword.
- You can keep static methods specific to an interface, in the same interface rather than in a separate class.
- Static methods of interface can be called using interface name followed by method name. Ex. Counter.printAuthor()
- To call static method we don't need any implementation of interface or inheritance.

```
01: public interface Counter {
02:   void increment();
03:   void decrement();
```

```
04:   int getValue();
05:   default int getIncrement() {
06:         return 1;
07:   }
08:   static void printAuthor() {
09:   System.out.println("Imaginnovate");
10:   }
11: }
```

```
01: public class OneCounter implements Counter {
02:   private int value;
03:   @Override
04:   public void increment() {
05:         value += getIncrement();
06:   }
07:   @Override
08:   public void decrement() {
09:         value -= getIncrement();
10:   }
11:   @Override
12:   public int getValue() {
13:         return value;
14:   }
15: }
```

```
01: public class TenCounter implements Counter {
02:   private int value;
03:   @Override
04:   public void increment() {
05:         value += getIncrement();
06:   }
07:   @Override
08:   public void decrement() {
09:         value -= getIncrement();
10:   }
11:   // Override default implementation
12:   @Override
13:   public int getIncrement() {
```

```
14:         return 10;
15:   }
16:   @Override
17:   public int getValue() {
18:         return value;
19:   }
20: }
```

```
01: public class TestCounter {
02:     public static void main(String[] args) {
03:        Counter.printAuthor();
04:
05:        Counter c = new OneCounter();
06:        c.increment();
07:        System.out.println(c.getIncrement());
08:        System.out.println(c.getValue());
09:
10:        c = new TenCounter();
11:        c.increment();
12:        System.out.println(c.getValue());
13:     }
14: }
```

Functional Interface

- An interface that contains only one abstract method is called a functional interface.
- Functional interface can have default methods, but must have only one abstract method.
- An interface can be explicitly marked as a functional interface using **@FunctionalInterface** annotation. However, the compiler will treat any interface meeting the definition of a functional interface as a functional interface regardless of whether or not a FunctionalInterface annotation is present on the interface declaration.
- Instances of functional interfaces can be created with lambda expressions, method references, or constructor references.
- Examples are **Runnable**, **Comparable** and **Comparator**.

```
01: @FunctionalInterface
02: public interface Printable {
03:   void print(String mgs);
04:   default String newline() {
05:         return "\n";
06:   }
07: }
```

Abstract Class vs. Interface

- Both interface and abstract class cannot have objects, but can have object references.
- In the case of interface, object reference can reference an object of implementing class. An object reference of abstract class can reference an object of subclass.
- However, abstract classes can have instance variables, but interfaces cannot have instance variables.

Null Interface

- Java allows interfaces to be empty. An empty interface contains no variables and methods. It is used to tag or mark a class as having certain properties or behavior.
- For example,a **Serializable** interface is a null interface.
- Null interface is also known as **marker interface**.

Private Methods in Interface

- Interfaces can have private methods that can be called from other methods of the interface.
- Private methods in interface cannot be called from implementing class.
- Provides more reusability in interfaces as these private methods can be called from other default methods.
- Must be declared with a **private** keyword and must contain a body.

```
01: public interface Logger {
02:  private void log(String message, String target) {
03:      // log message to target
04:  }
```

```
05:
06:  default void logToFile(String message) {
07:      log(message, "file");
08:  }
09:
10:  default void logToDatabase(String message) {
11:      log(message, "database");
12:  }
13: }
```

```
01: class TLC {
02:  private int v = 10;
03:
04:  public void somemethod() {
05:   class LocalClass {
06:    public void print() {
07:     System.out.printf("Local class.
08:                        Outer class value %d\n",
v);
09:    }
10:  }
11:   // use class here
12:   LocalClass obj = new LocalClass();
13:   obj.print();
14:  } // end of method
15: } // end of TLC
16:
17: public class Local Test {
18:   public static void main(String[] args) {
19:         TLC obj = new TLC();
20:   obj.somemethod();
21:   }
22: }
```

```
Local class. Outer class value 10
```

Packages

- Package is a collection of classes and interfaces.

- Package allows you to store your classes and interface without name collision as two packages may have same names for classes and interfaces.
- Packages are stored in a hierarchical manner. A package may be nested in another package. For example, **java.util**, **java.lang** etc. In **java.util**, **util** is a package that is in another package **java**.
- A package is a *folder* in the underlying operating system.

NOTE: All the classes that are not explicitly stored in any package are stored in **default package** or unnamed package, which is the current directory. And these classes are accessed directly without any package prefix.

Creating a Package and placing a Class in it

- Use a package statement to specify the package to which classes and interfaces belong to.
- The **package** statement is applied to all classes and interfaces in the source file.
- A folder, in the file system of Operating System, with the same name as the package is to be created.
- The .class file must be stored in the package (directory).
- Once a class is placed in a package, it must always be accessed using package name as prefix – *packagename.classname*.

```
package <fully qualified package name>
```

The following example shows how to create a class in a package called **library**. The package **library** must be created by creating a folder with that name.

```
01: package library;  //Book class belongs to library package
02: public class Book   {
03:  //class code
04: }
```

NOTE: If a package statement is used then it must be the first statement in the source file.

Class **Book** belongs to the package **library**. Apart from using package statements in .java file (as shown above), **Book.class** must be placed in the folder **library** to make **Book** class belong to the library package.

NOTE: A class with **public** access may be accessed from outside the package, whereas a class with **default** access (no access) is confined to the package in which it is defined.

If you want two public classes to belong to the same package then each class must be placed in a different .java file as shown below.

```
01: //Book.java
02: package library;
03: public class Book   {
04:   //body of Book class
05: }
```

```
01: //Member.java
02: package library;
03: public class Member  {
04:   //body of Member class
05: }
```

Accessing Classes of a Package

You can access a member of the package by using the package name followed by dot (.). A member of the package must be prefixed with the package name. The following code shows how to access the Book class of the library package.

```
01: public class UseLibrary  {
02:     public static void main(String[] args)  {
03:         library.Book b = new library.Book();
04:     }
05: }
```

A package can be accessed only from its parent directory (unless CLASSPATH is used). In the above example, the package library can be accessed only from the programs folder.

Import Statement

Use **import** statement to import specified classes or packages into the current source file.

```
import pkg1[.pkg2...].{classname/*}
```

***** means all classes of the package are to be imported, but it does not import subpackages of the package.

NOTE: The **import** statement must be given after the **package** statement in the source file if both exist.

The following examples import all members of package **st**, **Date** class in **java.util** package and all classes of **java.text** package**.**

```
import st.*;
import java.util.Date;
import java.text.*;
```

NOTE: Importing a package only imports classes in the package, but not nested packages in the package. So, classes in each nested package are to be explicitly imported.

NOTE: Package java.lang is automatically imported. So, its classes like String, System, Integer etc. can be used without importing them.

CLASSPATH variable

- Contains the list of directories where Java will look for classes and packages.
- Java compiler and runtime use CLASSPATH variable to get the list of folders and .**jar** files that are to be searched for the required classes and packages.
- By default, java searches for the specified classes in the current directory and in the standard library.

- But once CLASSPATH is set then Java uses only the directories in CLASSPATH and standard library to search for classes and does NOT search in the current directory.

The following example sets CLASSPATH to the current directory (**.**) and **c:\jdk\Imaginnovate** folder.

```
c:set classpath=.;c:\jdk\Imaginnovate
```

SET command is an operating system command. **CLASSPATH** is a variable used by Java compilers and JVM to locate packages and classes.

JAR (Java Archive) File

A JAR file contains packages, classes and interfaces. By including a .jar file in CLASSPATH we can make its packages, classes and interfaces available to Java.

JAR utility, which is provided with JDK, is used to create and extract JAR files. This tool is invoked from the command prompt.

The following command creates a JAR file to contain all .class files in the library folder. Flag **c** stands for create, **f** stands for file and **v** stands for verbose.

```
C:\jdk\Imaginnovate>jar cfv library.jar library\*.class
```

You can see the content of a JAR file by using flag **t** (table of content) as follows:

```
c:\jdk\Imaginnovate>jar ft library.jar
```

You need not extract the JAR file to use its packages and classes. Just including the .jar file in CLASSPATH will do.

So, to use **library.jar** that is placed in **c:\java** folder, we need to set classpath as follows:

```
c:set
classpath=.;c:\jdk\Imaginnovate;c:\java\library.jar
```

Access Modifiers

Access modifiers specify from where a member can be accessed. The following are the available access modifiers for members of a class and their meaning.

Access Modifier	Meaning
private	Allows members to be accessed only from the class in which it is defined.
protected	Allows member to be accessed from the class and all its subclasses and also from any class in the same package.
default	Allows members to be accessed throughout the package to which the class belongs to. If no access modifier is given then Java uses *default* as access modifier.
public	Allows a member to be accessed from anywhere.

NOTE: For top level classes, only **public** and **default** access modifiers are applicable. However, inner classes can have **private** and **protected** modifiers also.

The following example demonstrates how access modifiers influence the access of members from subclasses, other classes in the same package and other packages.

The following table summarizes access methods and their impact.

Where	Private	Default	Protected	Public
Same class	Yes	Yes	Yes	Yes
Subclass in same package	No	Yes	Yes	Yes
Non subclass in same package	No	Yes	Yes	Yes
Subclass in different package	No	No	Yes	Yes
Non-subclass in different package	No	No	No	Yes

A.java

```
01: package p1;
02: public class A {
03:     private int v1;
04:     protected int v2;
```

```
05:     public int v3;
06:     int v4;            // default access
07: }
```

B.java

```
01: package p1;
02: public class B extends A {
03: public void print() {
04:   // can access v2, v3, v4
05: }
06: }
07: class C {
08:   public void print() {
09:         A obj = new A();
10:   //can access v2, v3, v4 of obj
11:   }
12: }
```

C.java

```
01: package p2;
02: import p1.A;
03: public class D extends A {
```

```
04:   public void print() {
05:         //can access v2, v3
06:   }
07: }
08: class E {
09:   public void print() {
10:     A obj = new A();
11:     // can access v3 of obj only
12:   }
13: }
```

Static Import

- It allows static members of a class to be imported.
- Once the static members are imported, they may be used without qualification.
- Fully qualified names of the members must be used in import static statements.
- Should be used sparingly, otherwise it causes loss of readability.

```
01: import static java.lang.Math.PI;
02: import static java.lang.Math.abs;
03: import static java.lang.System.out;
04:
05: public class StaticImport {
06:     public static void main(String[] args) {
07:         out.println (PI);
08:         out.println (abs(-100));
09:     }
10: }
```

Enumeration

- Enumeration is a collection of constants. Java provides support for enumeration through keyword **enum**.
- An **enum** is internally converted to a full-fledged class.
- Values in enumeration can be compared and assigned.

- Enumeration can be used with switch statements and enhanced for loop.

```
01: enum Languages {
02:   C, CPP, JAVA, CSHARP
03: };
04: public class EnumDemo {
05:   public static void main(String[] args) {
06:         Languages lang;
07:
08:         lang = Languages.JAVA;
09:         if (lang == Languages.JAVA)
10:               System.out.println("Next is Java EE");
11:        System.out.println("Value of lang = " + lang);
12:         switch (lang) {
13:         case C:
14:               System.out.println("Java or C++");
15:               break;
16:         case CPP:
17:               System.out.println("Java");
18:               break;
19:         case JAVA:
20:               System.out.println("Java EE");
21:               break;
22:               case CSHARP:
23:               System.out.println("ASP.NET");
24:               break;
25:         }
26:   } // end of main
27:  } //end of class
```

```
Next is Java EE
 Value of lang = JAVA
 Java EE
```

Abstract Class

When a class contains one or more abstract methods, it must be defined as an abstract class using keyword abstract.

- Defined using **abstract** keywords.

- Cannot instantiate (create) objects of abstract class.
- However, an object reference of abstract class can be declared.
- Generally, it contains one or more abstract methods.
- A class can be declared as abstract even though it doesn't contain any abstract methods.
- Can contain other non-abstract members such as methods and instance variables.

Sealed Classes

- Sealed classes let you restrict class hierarchies to only certain classes.
- Sealed classes are declared with modifier **sealed** and specify which classes are permitted to extend the class using **permits** modifier.
- Only the classes defined after the keyword **permits** are allowed to extend the sealed class.
- Classes mentioned after permits must be in the same package or module.
- Every permitted subclass must explicitly extend the sealed class.
- Every permitted subclass must use one of the modifiers - **final**, **sealed**, or **non-sealed**.
- Concrete subtypes are implicitly **final**.

```
01: sealed class Course permits OnlineCourse,
OfflineCourse{
02:
03: }
04:
```

```
05: final class OnlineCourse extends Course {
06:
07: }
08:
09: non-sealed class offlineCourse extends Course {
10:
11: }
12:
13: class WeekendCourse extends OfflineCourse {
14:
15: }
```

Nested Classes

Nested class is a class defined in another class. Nested class may be any one of the following types:

- **Static member class**
- **Non-static member class**
- **Method-local inner class**
- **Anonymous inner class**

Static member Class

A class created in another class with a static modifier is called a static member class. It can be used just like a normal class.

However, to access it from outside of the enclosing class, we have to use the name of the enclosing class as the prefix for static member class.

```
01: // Top level class
02: class TLC {
03:   // static member class
04:    static public class SMC {
05:         public void print(){
```

```
06:            System.out.println("SMC");
07:           }
08:    }
09:    public void print() {
10:    System.out.println("TLC");
11:    }
12: }
13: public class TLCClient {
14:   public static void main(String... args) {
15:         TLC.SMC srec = new TLC.SMC();
16:   srec.print();
17:   }
18: }
```

Non-static member class (inner class)

A class defined within another class without a static modifier is called a non-static member class. It is also called the inner class.

Inner class has access to all of the variables and methods of its outer class and can refer to them directly. Inner classes were introduced in version 1.1 of Java, mainly to help handle events in AWT and Swing applications.

```
m1 = 10, m2 = 20
```

```
01: class OuterClass {
02: private int m1 = 10;
03: class InnerClass {
04:  private int m2 = 20;
```

```
05:  public void print() {
06:        System.out.printf("m1 = %d, m2 = %d",
m1,m2);
07:  }
08: }
09: public void print() {
10:  InnerClass obj = new InnerClass();
11:  obj.print();
12: }
13: }
14: public class InnerTest {
15:  public static void main(String args[]) {
16:        OuterClass obj = new OuterClass();
17:        obj.print();
18:  }
19: }
```

Method-local Inner Class

A class declared inside a method is called a local class. Only the method, in which the class is defined, can create an instance of the class.

Just like any other inner class, a local inner class can access members of its outer class directly.

```
Local class. Outer class value 10
```

```
01: class TLC {
02:  private int v = 10;
03:
04:  public void somemethod() {
```

```
05:   class LocalClass {
06:    public void print() {
07:     System.out.printf("Local class.
08:                         Outer class value %d\n",
v);
09:    }
10:   }
11:   // use class here
12:   LocalClass obj = new LocalClass();
13:   obj.print();
14:  } // end of method
15: } // end of TLC
16:
17: public class Local Test {
18:   public static void main(String[] args) {
19:         TLC obj = new TLC();
20:   obj.somemethod();
21:   }
22: }
```

Anonymous Inner Class

Anonymous inner class combines the process of defining an inner class and creating an object of the inner class into one step. It is a class without any name (anonymous).

```
new <superclass or interface>  (optional arguments) {
   Members definition
}
```

The optional arguments are passed to the constructor of super class.

Anonymous class can extend either a class or implement an interface. No **extends** or **implements** keyword is required. The name given after keyword **new** is taken as class or interface.

If an anonymous class implements an interface, then no arguments are passed as interfaces do not have constructors.

NOTE: Anonymous classes cannot have any constructors as they don't have a name.

```
01: interface Task {
02:   void process();
03: }
04:
05: public class AnonymousDemo {
06:   public static void main(String[] args) {
07:
08:      Task task= new Task() {
09:          public void process() {
10:            System.out.println("Do the process!");
11:          }
12:      };
13:
14:      task.process();
15:   }
16: }
```

Exception Handling

- An Exception is an abnormal condition (runtime error).
- When an exceptional event occurs, Java throws an exception.
- An exception is an object of a class that is created from **java**.**lang**.**Exception** class.
- Depending upon the error (exceptional event), an object of the exception class is created and thrown.
- If the exception is not handled then the program is terminated abnormally.
- Exceptions can be thrown by Java Runtime System or by the code.

The following code demonstrates what happens when you run code that throws an exception.

```
01: public class Test {
02:     public static void main(String[] args) {
03:     int n;
04:
```

```
05:       n = Integer.parseInt(args[0]);
06:       if (99% n == 0) {
07:          System.out.printf("%d is a factor of 99",
n);
08:       }
09:    }
10: }
```

If the above program is run with a command line parameter that cannot be converted to int (such as *abc*) then exception **NumberFormatException** is thrown.

```
java Test abc
Exception in thread "main"
java.lang.NumberFormatException:
For input string: "ABC" at
java.lang.NumberFormatException.forInputString(Unknown
Source) ...
```

When an exception is thrown, the program is terminated at the line that caused the exception.

The following are the keywords related to exception handling in Java.

Keyword	Meaning
try	Specifies the block in which you want to monitor for exceptions.
catch	Catches an exception and takes necessary action.
finally	Contains the code that must be executed before leaving the try block.
throws	Specifies the exceptions that a method might throw.
throw	Throws the given exception.

The following is the general syntax for try, catch and finally blocks.

```
try {
// try block
```

```
 }
catch (<exceptiontype>  <parameter>) {
//catch block
 }
finally {
// finally block
 }
```

NOTE: After try block either catch or finally or both may be given. But you cannot avoid both.

Handling exception using try and catch

If you want to handle the exception that is likely to be thrown then place the code in the try block and handle the exception in the catch block as shown below.

```
01: public class Test {
02: public static void main(String[] args) {
03:   int n;
04:   try{
05:   n = Integer.parseInt(args[0]);
06:   if (99 % n == 0)
07:         System.out.printf("%d is a factor of 99",
n);
08:   }
09:   catch (NumberFormatException ex) {
10:   System.out.println("Invalid number!");
11:   }
12: } // end of main
13: } // end of class
```

Handling multiple exceptions

It is also possible to handle more than one exception. If the code in the try block is likely to throw more than one exception then you have to handle all potential exceptions using multiple catch blocks.

However, note that a try block can throw only one exception at a time. Because once an exception is thrown, the code after the statement causing the exception will not be executed.

The following code shows how to handle two exceptions – *NumberFormatException* and *ArrayIndexOutOfBoundsException*.

```
01: public class Test {
02:   public static void main(String[] args) {
03:         int n;
04:   try{
05:   n = Integer.parseInt(args[0]);
06:   if (99 % n == 0)
07:         System.out.printf("%d is a factor of 99",
n);
08:   }
09:   catch (NumberFormatException ex) {
10:   System.out.println("Invalid number!");
11:   }
12:   catch (ArrayIndexOutOfBoundsException ex) {
13:    System.out.println("Number parameter is
missing");
14:    }
15:   }
16: }
```

In the code above, try block will throw *ArrayIndexOutOfBoundsException* if no parameter is passed on command line (as args[0] is not available). The other exception that will be thrown if string in args[0] cannot be converted to an integer is *NumberFormatException.*

Multi-Catch

Prior to Java 7, we had to catch only one exception type in each catch block. So whenever we needed to handle more than one specific exception, but take the same

action for all exceptions, then we had to have more than one catch block containing the same code.

In the following code, we have to handle two different exceptions, but take the same action for both. So, we need two different catch blocks as of Java 6.0.

```
try {
int v = Integer.parseInt(num); int
    result = 100/v;
    System.out.println(result);
 }
catch (NumberFormatException ex) {
// take action
 }
catch (ArithmeticException ex) {
// take action
 }
```

Starting from Java 7.0, it is possible for a single catch block to catch multiple exceptions by separating exception with **|** (pipe) symbol in catch block.

```
try {
 int v = Integer.parseInt(num);
  int result = 100/v;
    System.out.println(result);
}
// multi-catch
catch (NumberFormatException ex | ArithmeticException
ex) {
// take action
}
```

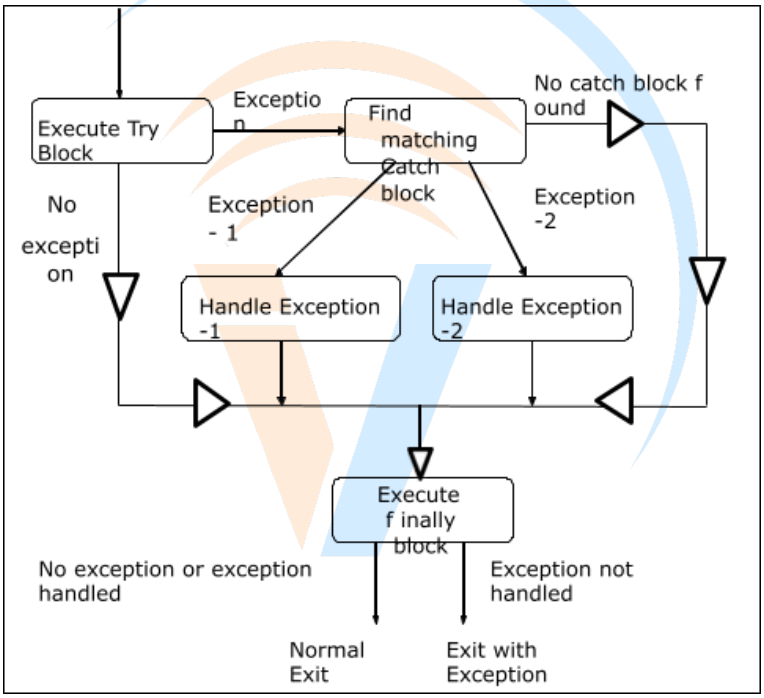
A catch block that handles multiple exception types creates no duplication in the bytecode generated by the compiler; the bytecode has no replication of exception handlers.

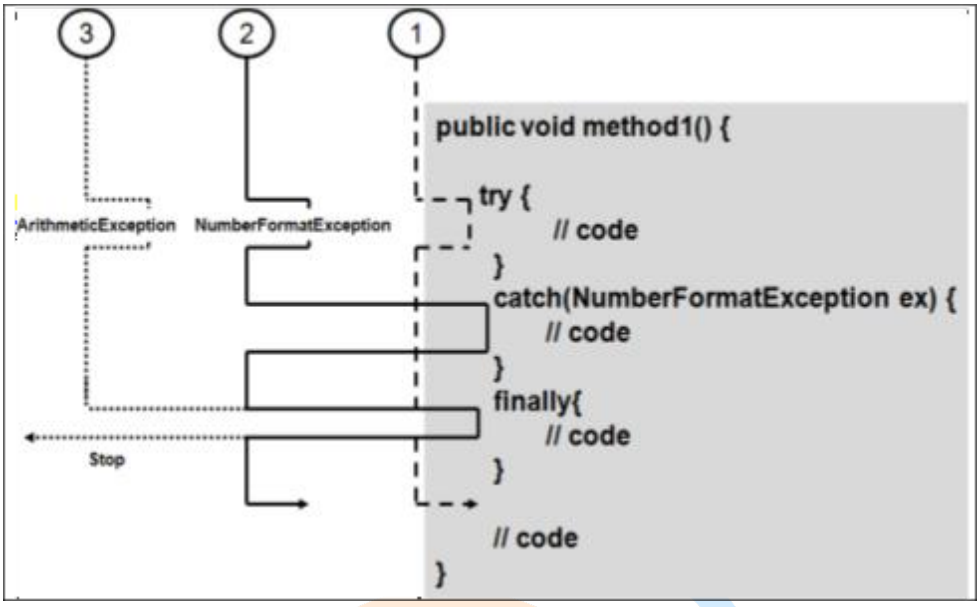
The finally block

- Used to execute code that is to be executed after a try/catch block has completed and before the code following try/catch is executed.

- Can be used to perform any operation that must be done before you leave the try/catch block – clean up code.
- The **finally** clause is optional. If it is given, it must be given either after catch block(s), if it exists, or after try block.

```
try {
 //  code
}
finally {
 //  code
}
```





The following snippet demonstrates how to use **finally** block.

```
Public void m1()  {
 try {
// code
  }
catch (NumberFormatException ex) {
// exception code
  }
finally {
// place cleanup code
  }
}
```

The following are the cases in which the finally block in the above snippet is executed:

- When try block is successfully executed without any exceptions then control goes to finally block after try block.
- When a NumberFormatException is thrown in the try block, the exception is handled by the catch block and then control goes to the finally block.
- When try block throws an uncaught exception then also control goes to finally block and then program is terminated. But in this case the program is terminated after the finally block is executed.

Run the code below and see the output (shown below) to understand flow of control with try, catch and finally blocks.

```
01: public class FinallyTest {
02:     public static void main(String[] args) {
03:        method("5");
04:        method("abc");
05:        method("0");
06:     } // end of main()
07:
08:     public static void method (String value) {
09:      try {
10:        int n = Integer.parseInt(value);
11:        System.out.println(100 /n);
12:      }
13:      catch (NumberFormatException ex) {
14:        System.out.println("Invalid number!");
15:      }
16:      finally {
17:        System.out.println("Finally block.");
18:      }
19:
20:      System.out.println("**Method Completed**");
21:     } //end of method()
22: }
```

```
20
Finally block.
**Method Completed**
 Invalid number!
Finally block.
**Method Completed**
 Finally block.
Exception in thread "main"
java.lang.ArithmeticException: / by zero
at FinallyTest.method (FinallyTest.java:13) at
         FinallyTest.main(FinallyTest.java:8)
```

Exceptions may be thrown by JVM or by methods in classes (API). In the above program ***NumberFormatException*** is thrown by **parseInt**() of ***Integer*** class, which is Java API.

Exceptions ArithmeticException and ArrayIndexOutOfBoundsException are thrown by JVM.

Types of Exceptions – checked and unchecked

There are two types of exceptions - Unchecked or Implicit exceptions and Checked or Explicit exceptions.

Unchecked Exceptions

Handling unchecked exceptions is not mandatory. Some of these exceptions are thrown by JVM and others by Java API. However, you may want to handle unchecked exceptions to prevent programs from getting terminated and handle exceptional events.

All exceptions that extend **RuntimeException** class are unchecked exceptions.

Checked Exceptions

Checked exceptions are thrown by methods. These exceptions must be handled by the code calling the method using try and catch blocks.

In method declaration, we have to specify the checked exceptions that the method is likely to throw using **throws** clause.

Methods that might throw a checked exception must be called from try block where catch block handles the checked exception. Or the calling method must declare a checked exception that might be thrown by the called method in its **throws** clause.

NOTE: Checked exceptions are always thrown from methods and never by JVM

The following code snippet shows how to handle or propagate checked exceptions.

```
01: public class TestChecked {
02:     public void m1() throws AmountException {
03:       // code
04:     }
05:     public void m2() throws AmountException {
06:       m1();
07:     }
08:     public void m3() {
09:       try {
10:         m1();
11:       }
12:       catch (AmountException ex) {
13:       }
14:     }
15: }
```

The following are the points related to above program:

- Method m1() is likely to throw a checked exception, **AmountException,** and it must be handled by methods calling m1().
- Method m2() compiles even though it calls m1() without handling AmountException as it declares **AmountException** in its throws clause.
- Method m3() handles **AmountException** that is likely to be thrown by m1() using try and catch blocks.

Invalid vs. Valid order of catch blocks

Understanding the hierarchy of exception classes is important. Most specific exceptions must be handled first and then less specific.

The following snippet shows invalid order for exception handling and will result in compilation error.

```
try {
   ...
}
catch(Exception ex) { }
catch(RuntimeException ex)  { }
catch(NumberFormatException ex) { }
```

Any exception is a match to the Exception class. So giving catch block with Exception class first will not allow other catch blocks to ever get control and makes those catch blocks unreachable.

Valid order of catch blocks

The following is the valid order of catch blocks. Most specific exception – *NumberFormatException* is handled first and most general, *Exception,* is handled last.

```
try {
   ....
}
catch(NumberFormatException ex) {  }
catch(RuntimeException ex)  {  }
catch(Exception ex) {  }
```

Creating user-defined exception

It is possible to create a new exception by creating a class. User-defined exceptions can be thrown and caught just like ready-made exceptions. However, unlike JVM exceptions, which are thrown by Java Runtime, user-defined exceptions are to be explicitly thrown by methods.

To create a user-defined exception, create a class that extends **Exception** class directly or indirectly.

The exception class need not create any method of its own since it inherits methods of **Exception** class. The following code shows how to create a user-defined exception and how to throw and catch.

```
01: class StackFullException extends Exception {
02:    public StackFullException() {
03:      super("Stack is full"); // exception message
04:    }
05: }
06: class Stack {
07:    public void push (int v) throws Stack
FullException {
08:      if(condition)
09:            throw new StackFullException();
10:    }
11: }
```

User-defined exception *StackFullException* will be thrown by the push method in case of an error. Any code calling the **push** method must make sure it catches *StackFullException* as it is given in the **throws** clause of push method. The following snippet shows how to handle the push method.

```
try {
 s.push(10);
      ...
}
catch(StackFullException ex) {
    System.out.println (ex.getMessage());
 }
```

Exception class

The following are the methods available in *Exception* class. In fact, all these methods are actually derived from *Throwable* class and *Exception* class has no methods of its own.

Method	Meaning
getMessage()	Returns description of the exception. The message is passed to the Exception class by its subclass using the constructor of Exception class.
printStackTrace()	Displays the stack trace providing all methods that are invoked from the beginning to the point of error.

User-defined exceptions example

Here is a complete example to demonstrate creating, throwing and handling user-defined exceptions using Account class.

```
01: public class InvalidAmountException extends
Exception {
02:  public InvalidAmountException() {
03:        super("Invalid Transaction Amount!");
04:  }
05:  public InvalidAmountException(String msg) {
06:        super(msg);
07:  }
08: }
```

```
01: public class InsufficientAmountException
02:   extends Exception {
03:   public Insufficient AmountException() {
04:         super("Insufficient Balance!");
05:   }
06:   public Insufficient AmountException(String msg) {
07:         super(msg);
08:   }
09: }
```

```
01: public class Account {
02: String acno;
03: double balance;
04: public Account (String acno, double balance) {
05:   this.acno= acno;
06:   this.balance = balance;
07: }
08: public void deposit (double amount)
09:         throws InvalidAmountException {
10:   if (amount < 100) {
11:         throw new InvalidAmountException();
12:   }
13:   balance += amount;
14: }
15: public void withdraw(double amount) throws
16:   InvalidAmountException, Insufficient
AmountException{
17:   if (amount < 100) {
18:         throw new InvalidAmountException();
19:   }
20:   if (balance < amount) {
21:         throw new Insufficient AmountException();
22:   }
23:   balance -= amount;
24: }
25: }
```

```
01: public class UseAccount {
02:   public static void main(String[] args) {
03:       Account a = new Account ("1002", 10000);
04:        try {
05:   a.deposit (1000);
06:   System.out.println("Deposited 1000");
07:   a.deposit (50);
08:   }
09:   catch (InvalidAmountException ex) {
10:   System.out.println(ex.getMessage());
11:   }
12:   try {
```

```
13:   a.withdraw(5000);
14:   System.out.println("Withdrew 5000");
15:   a.withdraw(8000);
16:   System.out.println("Withdrew 8000");
17:   }
18:   catch (Exception ex) {
19:          ex.printStackTrace();
20:   }
21:   }
22: }
```

```
Deposited 1000
Invalid Transaction Amount!
 Withdrew 5000
 Insufficient AmountException: Insufficient Balance!
         at Account.withdraw(Account.java:26)
at UseAccount.main(UseAccount.java:24)
```

Assertions

- Assertions can be used to validate assumptions made about state of the program.
- Each assertion is associated with a Boolean condition. If a condition fails it means the assumption fails and assertion error occurs.
- Assertions are **disabled** by default. So, you must enable assertions at the time of running the program.
- Assertions are part of the development process. They should not be included in the final version of your code.

```
assert  <boolean expression>
assert  <boolean expression>  : message
```

When a boolean expression results in false, Java throws **AssertionError**. If it results in true, execution of the program continues normally.

If a message is given then it is displayed in stack trace when assertion error occurs. The following code shows how to use assertion to validate parameters of a private method.

```
01: public class AssertionTest {
02:
03:     private void print(int count) {
04:      assert count > 0: "Invalid count for print()";
05:     }
06:
07:     public static void main(String[] args) {
08:         AssertionTest t = new AssertionTest();
09:         t.print(1);
10:         t.print(0);
11:     }
12: }
```

If you run the previous program by enabling assertions using **–ea** flag of JVM as shown below then the following will result:

```
>java  -ea AssertionTest
Exception in thread "main" java.lang.AssertionError:
Invalid
 Count for print()
  ...
```

NOTE: Assertions must be used in such a way that, even though assertions are disabled, the functionality of the program does not change. And also remember assertions are only debugging aids.

Collections Framework

Collections framework provides classes to implement useful data structures (such as linked list) and algorithms (such as sorting). The following are advantages of using Collections Framework:

- **Reduces programming effort** by providing useful data structures like linked list and hash table, and algorithms like searching and sorting, so that you don't have to write them yourself. This saves a lot of programmer's time allowing programmers to spend more time on business logic and rules.
- **Increases performance** by providing high-performance implementations of useful data structures and algorithms.
- **Reduces the effort required to learn APIs** by eliminating the need to learn multiple ad hoc collection APIs.

The following are the core interfaces provided by Collections Framework and their implementing classes (concrete classes):

Interface	Meaning	Implementing classes
Collection	Provides basic operations required to a collection of objects.	
Set	Extends Collection and provides unique set of values.	HashSet
SortedSet	Extends Set and ensures elements are in sorted order.	TreeSet
List	Extends Collection and allows elements to be accessed by position.	ArrayList Vector LinkedList
Map	Provides method to maintain a collection of pairs, where each pair contains a key and value.	HashMap HashTable
SortedMap	Extends Map interface and ensures pairs are sorted on key.	TreeMap
Queue	Extends Collection and adds methods to insert, extract and inspect.	LinkedList PriorityQueue

Collection Interface

- This is the most basic interface. Contains methods to add, remove, search and iterate over list.
- Provides methods that collection related classes should provide.

Method	Meaning
boolean add(Object)	Adds an object to the collection.
boolean addAll (Collection col)	Adds all elements of the collection.
void clear()	Clears all elements from the collection.
boolean contains (Object obj)	Returns true if the given object is found in the collection.
boolean containsAll (Collection col)	Returns true if all elements of parameter collection are present in the current collection.
boolean isEmpty()	Returns true if the collection is empty.
Iterator iterator()	Returns an iterator, which is used to iterate over items of collection.
boolean remove (Object obj)	Removes the given object.
boolean removeAll (Collection col)	Removes all elements from the collection that are present in parameter collection.
boolean retainAll (Collection col)	Retains only the elements in the list that are in the parameter collection.
int size()	Returns number of elements.
Object [] toArray()	Returns an array containing all the elements of the collection.
boolean removeIf (Predicate)	Removes all elements that satisfy given predicate.
Stream<E> stream()	Returns a sequential Stream.
Stream<E> parallelStream()	Returns a parallel Stream.

NOTE: Methods removeAll(), retainAll(), addAll() return true if source list is changed as the result of the operation, otherwise they return false.

NOTE: Though by default, parameters are of type Object for many methods, using generics, we can specify the type of objects the parameters must support.

List Interface

- Extends **Collection** interface.
- An ordered collection (also known as a *sequence*).
- Allows elements to be accessed by their position (index).
- Classes ArrayList, LinkedList and Vector implement List interface.

Method	Meaning
void add(int index, Object value)	Inserts the given object at the specified position.
Object get(int index)	Returns object from the specified position in this list.
int indexOf(Object value)	Returns the index of the first occurrence of the specified object or -1 if object is not found.
ListIterator listIterator()	Returns a list iterator of the elements in this list.
Object set(int index, Object newvalue)	Replaces the object at the specified position with the new object.
void replaceAll(UnaryOperator<E>)	Replaces each element of the list with the result of the operator.
void sort(Comparator)	Sorts the list according to order induced by Comparator.

List<E> subList(int from, int to)	Returns a list between from (inclusive) and to (exclusive) elements.
Object remove(int index)	Removes element at the specified position.

ArrayList Class

- Implements **List** interface.
- Each **ArrayList** instance has a *capacity*. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an *ArrayList*, its capacity grows automatically.
- One of the constructors allows you to specify the initial capacity of the list.

```
ArrayList()
ArrayList(Collection c)
ArrayList(int initialCapacity)
```

NOTE: It is important for classes whose objects are used with collections to override equals() method.

```
01: import java.util.*;
02: public class ArrayListDemo {
03:    public static void main(String args[]) {
04:    ArrayList nl = new ArrayList();
05:    nl.add("Ronaldo");
06:    nl.add("Messi");
07:    nl.add("Modric");
08:    nl.add(1, "Hazard");
09:    // display the list
10:    for (Object obj: nl)
11:        System.out.println(obj);
12:    nl.remove("Ronaldo");
13:    System.out.println(nl.indexOf("Messi"));
14:   }
15:
16: }
```

Set Interface

- **Set** interface represents a collection that contains no duplicate elements.
- It extends **Collection** interface**.**
- This doesn't add any new methods to *Collection* interface. A class implementing Set interface must make sure the elements are unique.
- **HashSet** and **LinkedHashSet** classes implement Set interface.

HashSet Class

- Implements the **Set** interface.
- It does not guarantee the order of elements.

```
HashSet()
HashSet(Collection c)
```

NOTE: Class LinkedHashSet is same as HashSet except that it orders elements according to insertion order.

NOTE: For objects that are used with HastSet, LinkedHashSet and TreeSet, it is important to override hashCode() method.

The following program demonstrates usage of **LinkedHashSet**. It displays unique lines (ignoring duplicates) of a file.

```
01: import java.nio.file.Files;
02: import java.nio.file.Path;
03: import java.util.LinkedHashSet;
04:
05: public class UniqueLines {
06: Public static void main(String[] args) throws
Exception{
07:    Path path = Path.of("c:\\classroom\\names.txt");
08:
09:    var lines = Files.readAllLines(path);
10:    var uniqueLines = new
LinkedHashSet<String>(lines);
11:     for (var line : uniqueLines)
12:     System.out.println(line);
13:    }
14: }
```

SortedSet Interface

- **SortedSet** extends **Set** and ensures the elements are always in the ascending order.
- It provides the following methods to retrieve elements from first, last and subset from head, tail or middle**.**
- Class **TreeSet** implements **SortedSet** interface.

Method	Meaning
Object first()	Returns first value in the list.
Object last()	Returns last value in the list.
SortedSet headSet (Object toElement)	Returns a list consisting of values that are less than toElement.
SortedSet tailSet (Object fromElement)	Returns a list consisting of values that are greater than or equal to fromElement.
SortedSet subset (Object fromElement, Object toElement)	Returns a list consisting of values that are from fromElement, inclusive, to toElement, exclusive.

The following program is to display common lines (intersection) of two files.

```
01: import java.nio.file.Files;
02: import java.nio.file.Path;
03: importjava.nio.file.StandardCopyOption;
04: import java.util.LinkedHashSet;
05:
06: public class IntersectFiles {
07:  public static void main(String[]args) throws
Exception{
08: Path
oldNamesPath=Path.of("c:\\classroom\\names.txt");
09: Path new
NamesPath=Path.of("c:\\classroom\names2.txt");
10:  var oldNames = new LinkedHashSet<String>
11: (Files.readAllLines(oldNamesPath));
12:  var commonNames = new LinkedHashSet<String>();
13:  for(String name :
Files.readAllLines(newNamesPath)){
14:  if (oldNames.contains(name))
```

```
15:             commonNames.add(name);
16:    }
17:    for(String name:commonNames)
18:    System.out.println(name);
19:    }
20: }
```

TreeSet Class

- This class implements the SortedSet interface.
- It is based on a balanced tree data structure.
- Objects being used must be comparable (by implementing **Comparable** interface, which contains **compareTo**() method).

```
TreeSet()
TreeSet(Comparator)
```

It ensures objects are sorted according to natural order. In case, you specify *comparator* (using second constructor), which is an object of a class that implements **Comparator** interface, the objects are sorted according to order specified by *Comparator.*

```
01: import java.util.TreeSet;
02: class Person implements Comparable<Person>{
03: private String name;
04: private int age;
05: public Person(String name, int age){
06:   this.name = name;
07:   this.age =age;
08: }
09: public int compareTo(Person other) {
10: return this.name.compareTo(other.name);
11: }
12: public String toString() {
13:   return name + " - " +age;
14: }
15: }
16:
```

```
17: public class SortPersons {
18: public static void main(String args[]){
19:   var people = new TreeSet<Person>();
20:
21:  people.add(new Person("Larry", 34));
22:  people.add(new Person("Bill", 44));
23:  people.add(new Person("Bill", 44));
24:  people.add(new Person("Mark",25));
25:   for (var p : people)
26:     System.out.println(p);
27: }
28: }
```

Comparator<T> Interface

Comparator allows you to implement a compare method so that the program can specify the precise order for sorting.

```
int compare<T>(T o1, T o2)
```

The return value of **compare**() method must be one of the following.

Return value	What does it mean?
Zero (0)	Both the objects are equal.
Greater than 0 (> 0)	First object is greater than the second object.
Less than 0 (< 0)	First object is less than the second object.

Queue Interface

- This interface extends Collection interface and provides methods to add and retrieve values from queue.
- Classes **LinkedList** and **PriortyQueue** implement **Queue** interface.

Method	Meaning
Object element()	Retrieves, but does not remove, the head of the queue. Throws exception – NoSuchElementException – if queue is empty.
boolean offer (Object)	Inserts an element into queue, if possible.
Object poll()	Retrieves and removes the head of the queue or returns null if queue is empty.
Object remove()	Retrieves and removes the head of this queue. Throws exception if queue is empty.
Object peek()	Retrieves, but does not remove, the head of this queue, returns null if this queue is empty.

LinkedList Class

- Doubly-Linked list implementation of the **List** interface.
- Can be used as a stack, queue or double-ended queue (deque).
- Starting from Java 5.0, it implements **Queue** interface also.
- This implementation is not synchronized.

```
LinkedList()
LinkedList(Collection c)
```

Apart from methods in **List** and **Queue** interfaces, it provides the following extra methods.

Method	Meaning
void addFirst(Object o)	Inserts the given element at the beginning of the list.
void addLast(Object o)	Appends the given element to the end of this list.
Object getFirst()	Returns the first element of this list.
Object getLast()	Returns the last element of this list.
Object removeFirst()	Removes and returns the first element from this list.
Object removeLast()	Removes and returns the last element from this list.

NOTE: Class PriorityQueue implements Queue interface and orders elements by natural order. Elements of PriorityQueue must be comparable – implement Comparable interface, which contains compareTo() method.

Map Interface

- Allows keys to map values.
- A map does not allow duplicate keys. Each key can map to at most one value.
- **HashMap** and **HashTable** implement **Map** interface.

Method	Meaning
void clear()	Clears the map.
boolean containsKey(Object key)	Returns true if the given key exists.
Object get(Object key)	Returns the value of key.
boolean isEmpty()	Returns true if this map is empty.
Set keySet()	Returns a set view of the keys contained in this map.
Object put (Object key, Object value)	Places the key and the value.
Object remove (Object key)	Removes the mapping for this key from this map if present.
boolean remove (Object key, Object value)	Removes value for the key if it contains the given value.
int size()	Returns the number of keys.
Collection values()	Returns a collection view of the values contained in this map.
void forEach(BiConsumer<K,V>)	Performs the given action for each element.
V getOrDefault (Object key, V defaultValue)	Returns the value for key or *defaultValue* if key not found.
V putIfAbsent(K key, V value)	If key is not associated with a value, then puts value and returns null, else returns current value.
V replace(K key, V value)	Replaces value of given *key* to *value*.
boolean replace (K key, V oldValue, V newValue)	Replaces the value of key to new value if it contains old value.

HashMap Class

- This class implements the Map interface.
- It does not guarantee the order of keys.

```
HashMap()
HashMap(Map)
```

NOTE: HashTable class is same as HashMap class, but it is synchronized. LinkedHashMap class extends HashMap class and arranges keys in the order of insertion.

SortedMap Interface

This interface extends the Map interface to provide sorted keys.

The following are the methods that are added in the SortedMap interface compared with **Map** interface.

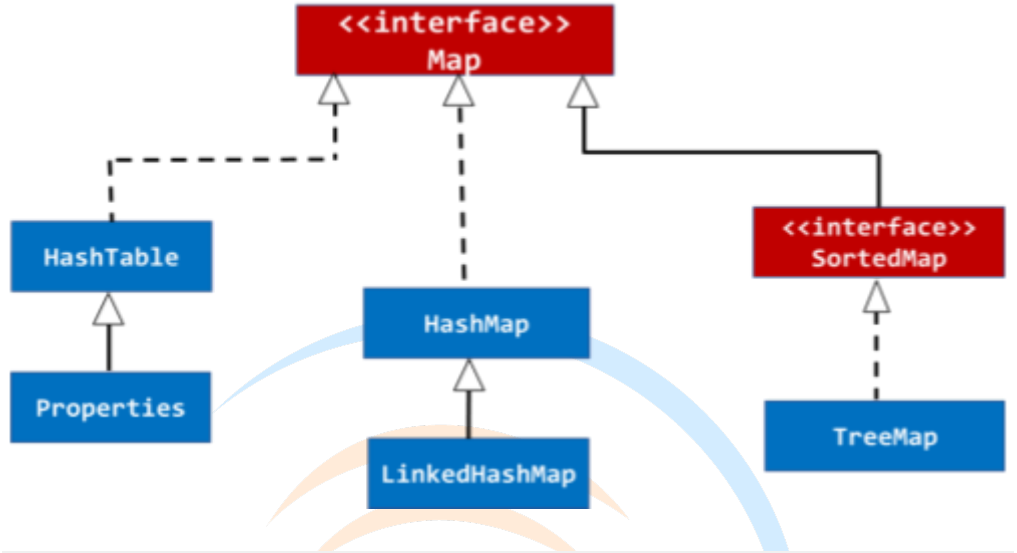
Method	Meaning
Object first()	Returns the first element.
Object last()	Returns the last element.
SortedSet headMap (Object toKey)	Returns list up to the specified element and not including it.
SortedSet tailMap (Object fromKey)	Returns list from the specified element up to end of the list.
SortedSet subMap(object fromKey, object toKey)	Returns list that contains elements from the given *fromKey* to *toKey* but not including it.

TreeMap Class

- This class implements the SortedMap interface. It arranges keys in the sorted order.
- Order of keys is either natural order or specified by **Comparator.**

```
TreeMap()
TreeMap(Comparator)
```

The following program uses HashMap, LinkedHashMap and TreeMap to show the way these classes arrange keys.




```
01: import java.util.HashMap;
02: import java.util.LinkedHashMap;
03: import java.util.TreeMap;
04:
05: public class MapDemo {
06:   public static void main(String[] args) {
07: HashMap<String, String> hm = new HashMap<>();
08:         hm.put("java", "Language");
09:   hm.put("dotnet", "Framework");
10:   hm.put("c#", "Language");
11: System.out.println("HashMap Output");
12:         for (String key : hm.keySet())
13:   System.out.printf("%s:%s\n",key,hm.get(key));
14:
15:      //Arranges keys in insertion order
16:   LinkedHashMap<String,String> lhm
17:               = new LinkedHashMap<>();
18:   lhm.put("java", "Language");
19:   lhm.put("dotnet", "Framework");
20:   lhm.put("c#", "Language");
21:      System.out.println("LinkedHashMap Output");
22:   for (String key : lhm.keySet())
23:   System.out.printf("%s:%s\n",key,lhm.get(key));
```

```
24:
25:   // Arranges keys in ascending order
26:  TreeMap<String, String> tm = new TreeMap<>();
27:        tm.put("java", "Language");
28:   tm.put("dotnet", "Framework");
29:   tm.put("c#", "Language");
30: System.out.println("TreeMap Output");
31:        for (String key : tm.keySet())
32:   System.out.printf("%s:%s\n",key,tm.get(key));
33:   }
34: }
```

Collections Class

- It contains a collection of static methods that operate on collections provided as parameters of the collection.
- Implements various algorithms related to collections.

Method	Meaning
int binarySearch(List l, Object v)	Implements binary search algorithm to search.
void copy(List dest, List src)	Copies content of one list to another.
void fill(List list, Object obj)	Fills the list with the given value.
Object max (Collection c)	Returns maximum value in collection.
Object min (Collection c)	Returns minimum value in collection.
void reverse(List l)	Reverses the list.
void shuffle(List l)	Shuffles the list with default randomness.
void sort(List l)	Sorts the list.
void sort(List l, Comparator c)	Sorts the list with order specified by Comparator.

```
01: import java.util.*;
02: public class CollectionsDemo {
03: public static void main(String args[]) {
04:  ArrayList<String> names = new ArrayList<>();
05:   names.add("James Goodwill");
06:   names.add("Jason Hunter");
07:   names.add("Roman");
08:   Collections.sort(names);
```

```
09:   printList(names);
10:   int pos = Collections.binarySearch(names,
"Roman");
11:   System.out.printf("Roman is found at: %d\n", pos);
12:
System.out.printf("Max:%s\n",Collections.max(names));
13:
14:   System.out.println("Reverse Order");
15:   Collections.reverse(names);
16:   printList(names);
17: }
18: public static void printList(List<String> list){
19:   for (String s: list) {
20:   System.out.println(s);
21:   }
22: }
23: }
```

Immutable Collections

- Collection is immutable i.e., we can't add, delete or modify elements.
- Nulls are not allowed.
- Serializable if all elements are serializable.
- We can create a list of up to 10 elements as there are 10 parameters for the of() method.
- List.<E>of (E…elements) can take any number of parameters including an array.

The following are methods to create immutable collections:

```
List.of()
List.of(values)
Set.of()
Set.of(values)
Map.of()
Map.of(key,  value, key, value, .....)
Mpa.ofEntries(Map.Entry ... entries)
```

```
// Immutable Empty List List<String>
emptyList = List.of();

// Immutable List
List<String> immutablelist= List.of("one", "two",
"three");
```

```
// Immutable Empty Map
Map<String, String> emptyMap = Map.of();

// Immutable Map using Map.of()
Map<String, String> phones =
     Map.of("Bill","39393393988","Larry","9988776655");

// Immutable Map using Map.ofEntries()
Map<Integer, String> map =

Map.ofEntries(Map,entry(1,"One"),Hap.entry(2,"Two"));
```

Generics

- Allow you to specify to compiler the type of object you want to store in a collection.
- Generics make collection classes type-safe by ensuring a collection supports only one type of object.
- At the time of creating an object of a collection class, we can specify which type of objects the collection supports.
- Eliminates the need to typecast the object taken from collection – makes it convenient and faster.

```
01: // Code without Generics
02: ArrayList al = new ArrayList();
03: al.add("First");
04: al.add(new Integer(100));
05: String st;
06: st = (String) al.get(0); // typecasting is required
07: System.out.println(st.toUpperCase());
08: st = (String) al.get(1); // results in Exception
```

In the above code, when you try to typecast *al.get(1)* to *String*, it results in **ClassCastException**, because *al.get(1)* actually refers to an object of type *Integer* and it cannot be converted to *String*.

The following example demonstrates how generics make collections type-safe and convenient to use.

```
01: // Code using Generics
02: ArrayList<String> al = new ArrayList<String>();
03: al.add("First");
04: // al.add(new Integer (100)); // compile-time error
05: al.add("Second");
06: String st;
07: st = al.get(0); // typecasting is NOT required
08: System.out.println(st.toUpperCase());
09: // can be used with enhanced for loop
10: for (String s : al)
11: System.out.println(s.toLowerCase());
```

Generics allow you to specify the type of object you want a data structure like **ArrayList** to support. Compiler checks whether objects used with ArrayList **al** are of type **String**. If any other type of object other than **String** is used with **al** then it is taken as a compile-time error.

Generic Methods

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (<T> in this example).
- Note that type parameters can represent only reference types—not primitive types (like int, double and char).
- Type parameters can be declared only once in the type parameter section but can appear more than once in the method's parameter list.

Bounded Type Parameter

- To declare a bounded type parameter, list the type parameter's name, followed by **extends** keyword, followed by upper bound.
- Note that, in this context, **extends** is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

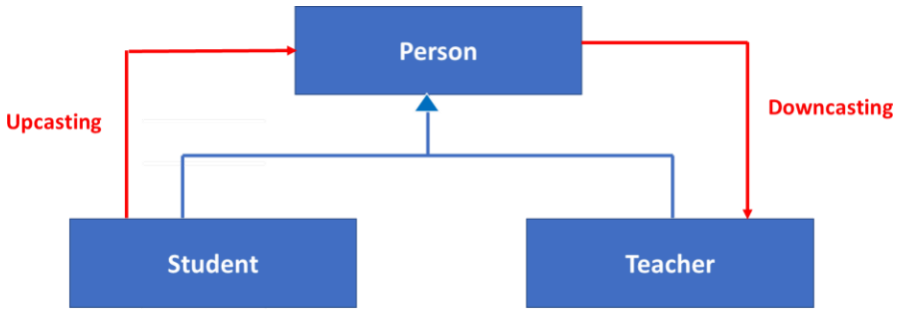
```
01: import java.util.*;
02: public class GenericMethods {
03:   public static <T> void print(T a[]){
04:         for(T v : a)
05:   System.out.println(v);
06:   }
07:  public static<T extends Comparable<T>>
08:         boolean contains(T a[], T v) {
09:   for (T value: a)
10:   if(value.compareTo(v) == 0)
11:   return true;
12:   return false;
13:   }
14:   public static void main(String args[]) {
15:   String names[]={"Steve","Joe","Bob","James"};
16:   print(names);
17:   System.out.println(contains(names,"Joe"));
18:   }
19: }
```

Downcasting and Upcasting

Java allows an object reference of superclass to refer to an object of subclass. This is the heart of runtime-polymorphism in Java.

Superclass is a more general form of subclass. Subclass is a more specific form of superclass. Person is a more general form of Student. In other words, a Student is always a Person. So, wherever Java expects Person, a Student can be used.

NOTE: Remember superclass and subclass share IS A relationship. A Student is always a Person.



Upcasting is casting an object to a class higher in the inheritance tree. When an object of the Teacher is assigned to a Person, it is called upcasting. Upcasting is implicit in Java.

```
Teacher f = new Teacher ("Hunter",
"hunter@servlets.com",
"servlets, jsp");
Person p = f; // Upcasting is done implicitly
Person p = (Person) f; // explicit casting, but NOT
required
```

Downcasting is casting an object to a class down in the ***hierarchy tree***. When an object of Person is cast to Teacher, it is called Downcasting.

Downcasting must be done explicitly using type casting syntax as follows:

```
Person p = new
Teacher ("Hunter", "hunter@servlets.com", "Java");
Teacher f = p; // doesn't compile
Teacher f = (Teacher) p; // Downcasting explicitly
```

NOTE: Downcasting is explicit and Upcasting is implicit. Downcasting is possible between a superclass and subclass only. For example, it is not possible to convert a String object to Person object as String and Person classes are not related.

The instanceof operator

Operator **instanceof** is used to check whether an object reference refers to an object of specified class or one of its subclasses.

```
object-reference instanceof class
```

```
 Person p = new Teacher("abc", "abc@gmail.com",
"java");
 System.out.println(p instanceof Teacher);
 System.out.println(p instanceof Person);
 System.out.println(p instanceof Student);
```

Pattern matching for instanceof

- Pattern matching is a feature finally standardized in Java 16, though it was introduced as a preview feature earlier.
- It converts superclass object reference to subclass if it points to specified subclass.
- Variables created by **instanceof** can be used only in the current block.

The following program checks whether object reference p points to Student object and if so, converts it to Student object:

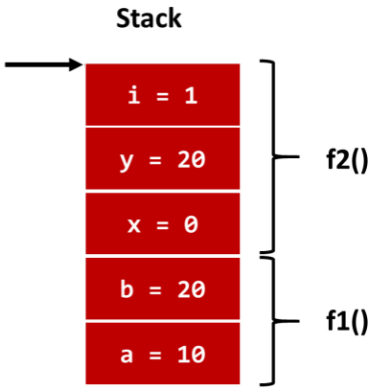
```
Person p = new Teacher ("Mike",
"mike@gmail.com","Java");
if (p instanceof Teacher t) {
 t.setSubjects("Java, Java EE");
}
```

Standard Data Types vs. Objects

Java treats standard data types (also called as primitive types) and objects of classes differently.

Standard data types are allocated memory on the **stack.** Parameters of primitive type are always **passed by value**.

```
public void f1() {
   int  a = 10, b = 20;

     f2(a, b);
}

public void f2(int x, int y) {
   int i = 1;

     x = 0;
}
```



In the above picture, variables a and b are created on stack when control enters into f1(). When function f2() is called, formal parameters x and y are created and assigned values from actual parameters – a and b.

However, changes to x and y will not change a and b as they occupy different locations in memory.

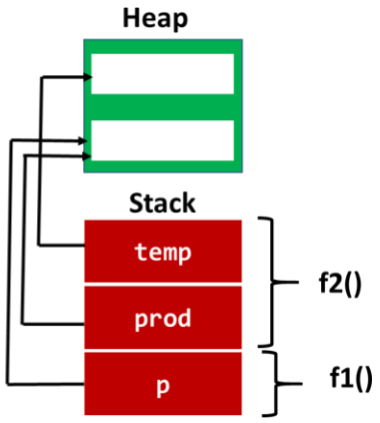
NOTE: Passing parameters of standard data types is always pass-by-value.

How objects are handled by Java?

Objects of classes are allocated memory dynamically on heap and they are released by garbage collector when they are not required.

Object is allocated memory on the heap. However, an object reference is allocated memory on stack. When an object is passed to a function, only reference of the object is passed but not the object itself.

```
public void f1() {
  Product p;
     p = new Product(…);
     f2(p);
}

public void f2(Product prod) {
  Product temp;
     temp = new Product(…);
     prod.setPrice(1000);
}
```



NOTE: An object is always passed by reference.

In the above figure, object references p in f1() and prod in function f2() both refer to the same object. Object reference temp points to a new object.

When control comes out of f2(), an object pointed by temp will be ready for garbage collection as it is no longer pointed by any object reference.

Wrapper Classes

Primitive types such as int and double are not objects in Java. Wrapper classes are used to convert primitive types to objects. Java provides a wrapper class for each primitive type (8 in total).

NOTE: Wrapper classes are declared as final and their values cannot be changed.

The following are the wrapper classes provided by Java and their corresponding primitive (standard) type.

Standard Type	Wrapper Class	Method to convert object to primitive type
boolean	Boolean	booleanValue()
byte	Byte	byteValue()
short	Short	shortValue()
int	Integer	intValue()
long	Long	longValue()
float	Float	floatValue()
double	Double	doubleValue()
char	Character	charValue()

Boxing and Unboxing

The following code shows how to use wrapper class Integer to convert an int to object (boxing) and back to int (unboxing).

```
01:  int x= 100;
02:     // convert int to an object
03:     Integer iobj = new Integer(x);   //boxing
04:
05:      //  get int from object
06:       x = iobj.intValue();
```

Autoboxing and Autounboxing

When a primitive data type is to be used as an object, it is to be boxed to an object of corresponding wrapper class. When the primitive value is to be taken from a wrapper class object then the value is to be unboxed from the object by using a method of wrapper class.

Starting from Java 5.0, Java does boxing and unboxing automatically. The following example shows how boxing and unboxing take place.

```
01:  public static void main(String[] args) {
02:  int i = 100;
03:     Integer iobj = i;        //autoboxing
04:    System.out.println(iobj.toString());
05:    i = iobj;                 // autounboxing
06:    System.out.println(i);
07:  }
```