Core Java

Course Introduction

Course contents -- Java 8

- Language features (Installation, Buzzwords, History, JRE/JDK/JVM, Compilation, Operators, Data types, Narrowing/Widening)
- Control Structures (if-else, loops, switch, recursion, wrapper classes, boxing, value/reference type)
- Input/Output: Commandline args, Scanner, Console
- OOP basics (OOP pillars, class/object, class methods, ctor chaining, overloading, final field, toString()), enum
- Static (static fields/method/block), Singleton, Package (classpath), static import
- Arrays (1-D & 2-D array, primitive and object array, java.util.Arrays)
- OOP advanced (aggregation/association/composition, Inheritance, super, Method overriding, instanceof, final method/class, Abstract class, Java 7 Interfaces, Marker interfaces)
- Java classes (java.lang.Object, Date/LocalDate/Calendar, String/StringBuilder/StringBuffer, etc)
- Exception Handling (try-catch-throw, throws, finally, try-with-resource/Closeable, errors, custom exception, checked/unchecked, overriding rules)
- Java Generics (methods, classes, interfaces), Comparable/Comparator interface, Java 8 Interfaces
- Nested classes, Functional Interfaces, Lambda expressions, Method references
- Java Collections (Hierarchy, Lists, Iterator, Sets, Maps, Queue/Stack, Fail-safe/Fail-fast Iterators, Legacy collections, Collections)
- Functional programming (Concepts, Java 8 interfaces, Stream characteristics, Stream operations, Collectors)
- Java IO (Binary vs Text streams, File IO, Stream chaining, Data stream, Serialization, Reader/Writers)
- Multi-threading (Thread vs Runnable, thread methods, Thread group, Synchronization, Deadlock)
- JVM Architecture, Reflection, Annotation, Garbage Collection

Reference Books

- Core Java Volume 1 & 2 - Horstmann
- Java Complete Reference - Herbert Schildt
- Java 8 Certification - Khalid Mughal
- Java Certification - Kathy Sierra

Java History

- Java goes back to 1991, when a group of Sun engineers, led by Patrick Naughton and James Gosling, wanted to design a small computer language that could be used for consumer devices like cable TV switchboxes.
- Since these devices do not have a lot of power or memory, the language had to be small and generate very tight code. Also, as different manufacturers may choose different CPU's, it was important that the language not be tied to any single architecture.
- The project was code-named "Green".

- The requirements for small, tight, and platform-neutral code led the team to design a portable language that generated intermediate code for a virtual machine.
- The Sun people came from a UNIX (Solaris) background, so they based their language on C++.
- Gosling decided to call his language "Oak". However, Oak was the name of an existing computer language, so they changed the name to Java.
- In 1992, the Green project delivered its first product, called "*7" - a smart remote control.
- Unfortunately, Sun was not interested in producing this, and the Green people had to find other ways to market their technology. However, none of the standard consumer electronics companies were interested either.
- The Green team spent all of 1993 and half of 1994 looking for sponsors to buy its technology.
- Meanwhile, the World Wide Web (WWW) was growing bigger and bigger. The key to the WWW was the browser translating hypertext pages to the screen.
- In 1994, most people were using Mosaic, a noncommercial web browser by University of Illinois.
- The Java language developers developed a cool browser - HotJava browser. This was client/server architecture-neutral application that was real-time and reliable.
- The developers made the browser capable of executing Java code inside web pages - called Applets. This POC was demonstrated at SunWorld on 23-May-1995.

Java versions

- JDK Beta - 1995
- **JDK 1.0 - January 23, 1996**
- JDK 1.1 - February 19, 1997
- **J2SE 1.2 - December 8, 1998**
    - Java collections
- J2SE 1.3 - May 8, 2000
- J2SE 1.4 - February 6, 2002
- **J2SE 5.0 - September 30, 2004**
    - enum
    - Generics
    - Annotations
- Java SE 6 - December 11, 2006
- Java SE 7 - July 28, 2011
- **Java SE 8 (LTS) - March 18, 2014**
    - Functional programming: Streams, Lambda expressions
- Java SE 9 - September 21, 2017
- Java SE 10 - March 20, 2018
- **Java SE 11 (LTS) - September 25, 2018**
- Java SE 12 - March 19, 2019
- Java SE 13 - September 17, 2019
- Java SE 14 - March 17, 2020
- Java SE 15 - September 15, 2020
- Java SE 16 - March 16, 2021
- **Java SE 17 (LTS) - September 14, 2021**
    - Jakarta SE 17
- Java SE 18 - March 22, 2022
- Java SE 19 - September 20, 2022

- Java SE 20 - March 21, 2023

Java platforms

- Java is not specific to any processor or operating system as Java platforms have been implemented for a wide variety of hardware and operating systems with a view to enable Java programs to run identically on all of them. Different platforms target different classes of device and application domains:
- **Java Card**: A technology that allows small Java-based applications to be run securely on smart cards and similar small-memory devices.
- **Java ME (Micro Edition)**: Specifies several different sets of libraries (known as profiles) for devices with limited storage, display, and power capacities. It is often used to develop applications for mobile devices, PDAs, TV set-top boxes, and printers.
- **Java SE (Standard Edition)**: Java Platform, Standard Edition or Java SE is a widely used platform for development and deployment of portable code for desktop environments
- **Java EE (Enterprise Edition)**: Java Platform, Enterprise Edition or Java EE is a widely used enterprise computing platform. The platform provides an API and runtime environment for developing and running enterprise software, including network and web services, and other large-scale, multi-tiered, scalable, reliable, and secure network applications.

Object Oriented

- Basic principles of Object-oriented Language
  - class
  - object
- class
  - User defined data type (similar to struct in C)
  - Has fields (data members) and methods (member functions)
    - static members: Accessed using class name directly
    - non-static members: Accessed using object
  - Defines the structure/blueprint of the created object/instance
  - Logical entity
- object
  - Instance of the class
  - One class can have multiple objects
  - Physical entity (occupies memory)

Hello World - Code, Compilation and Execution

- Code

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

- Explanation - main():
  - In Java, each variable/method must be in some class.
  - JVM calls main() method without creating object of the class, so method must be static.
  - main() doesn't return any value to JVM, so return type is void.
  - main() takes command line arguments - String args[]
  - main() should be callable from outside the class directly - public access.
- Explanation - System.out.println():
  - System is predefined Java class (java.lang.System).
  - out is public static field of the System class --> System.out.
  - out is object of PrintStream class (java.io.PrintStream).
  - println() is public non-static method of PrintStream class --> System.out.println("...");
- Compilation and Execution (in same directory)
  - terminal> javac Program.java
  - terminal> java Program

Entry point method

- main() is considered as entry poit method in java.

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

- JVM invokes main method.
- Can be overloaded.
- Can write one entry-point in each Java class.

java.lang.System class

```
```Java
public final class System
{
    //Fields
    public static final InputStream in;     // stdin
    public static final PrintStream out;    // stdout
    public static final PrintStream err;    // stderr
    //Methods
    public static Console console();
    public static void exit(int status);
    public static void gc();
    // ...
}
```
```

System.out.println()

- System: Final class declared in java.lang package and java.lang package is declared in rt.jar file.
- out: Object of PrintStream class declared as public static final field in System class.
- println(): Non-static method of PrintStream class.

C/C++ Program Compilation and Execution

- Main.cpp --> Compiler --> Main.obj --> Linker --> Main.exe
  - Main.cpp - Source code
  - Main.obj - Object code
  - Main.exe - Program = Executable code (contains machine level code)
- terminal> ./Main.exe
- Operating system creates a process to execute the program.
- Process sections
  - Text:
  - Data:
  - Rodata:
  - Stack:
  - Heap

Java Program Compilation and Execution

- Main.java --> Compiler --> Main.class --> JVM
  - Main.java --> Source code
  - Main.class --> Byte code (Intermediate Language code)
- terminal> javac Main.java
- terminal> java Main

JDK vs JRE vs JVM

- SDK is Software Development Kit required to develop application.
- SDK = Software Development Tools + Libraries + Runtime environment + Documentation + IDE.
  - Software Development Tools = Compiler, Debugger, etc.
  - Libraries = Set of functions/classes.
- JDK is Java platform SDK. It is a software development environment used for developing Java applications.
- JDK = Java Development Tools + JRE + Java docs.
  - Required to develop Java applications.
- JRE = Java API (Java class libraries) + Java Virtual Machine (JVM).
  - All core java fundamental classes are part of rt.jar file.
  - Required to develop and execute Java applications.

Hello World - Variations

- In STS Eclipse, classes are written under "src" directory. They are auto-compiled and generated .class files are placed under "bin" directory.
- One Java project can have multiple .java files. Each file can have main() method which can be executed separately.
- The main() method must be public static void. Missing any of them raise compiler error.

- The entry-point method must be main(String[] args). Otherwise, raise runtime error - main() method not found.
- The main() method can be overloaded i.e. method with same name but different parameters (in same class).
- If a .java file contains multiple classes, for each class a separate .class file is created.
- Name of (non-public) Java class may be different than the file name. The name of generated .class file is same as class name.
- Name of public class in Java file must be same as file-name. One Java file can have only one public class.

PATH vs CLASSPATH

- Enviorment variables: Contains important information about the system e.g. OS, CPU, PATH, USER, etc.
- PATH: Contains set of directories separated by ; (Windows) or : (Linux).
    - When any program (executable file) is executed without its full path (on terminal/Run), then OS search it in all directories given in PATH variable.
    - terminal> mspaint.exe
    - terminal> notepad.exe
    - terminal> taskmgr.exe
    - terminal> java.exe -version
    - terminal> javac.exe -version
    - To display PATH variable
        - Windows cmd> set PATH
        - Linux terminal> echo $PATH
    - PATH variable can be modified using "set" command (Windows) or "export" command (Linux).
    - PATH variable can be modified permanently in Windows System settings or Linux ~/.bashrc.
- CLASSPATH: Contains set of directories separated by ; (Windows) or : (Linux).
    - Java's environment variable by which one can inform Java compiler, application launcher, JVM and other Java tools about the directories in which Java classes/packages are kept.
    - CLASSPATH variable can be modified using "set" command (Windows) or "export" command (Linux).
        - Windows cmd> set CLASSPATH=\path\to\set;%CLASSPATH%
        - Linux terminal> export CLASSPATH=/path/to/set:$CLASSPATH
    - To display CLASSPATH variable
        - Windows cmd> set CLASSPATH
        - Linux terminal> echo $CLASSPATH
- Compilation and Execution (source code in "src" directory and .class file in "bin" directory)
    - terminal> cd \path\of\src directory
    - terminal> javac -d ..\bin Program.java
    - terminal> set CLASSPATH=..\bin
    - terminal> java Program

Console Input/Output

- Java has several ways to take input and print output. Most popular ways in Java 8 are given below:
- Using java.util.Scanner and System.out

```
Scanner sc = new Scanner(System.in);
System.out.print("Enter name: ");
String name = sc.nextLine();
System.out.print("Enter age: ");
int age = sc.nextInt();
System.out.println("Name: " + name + ", Age: " + age);
System.out.printf("Name: %s, Age: %s\n", name, age);
```

Language Fundamentals

Naming conventions

- Names for variables, methods, and types should follow Java naming convention.
- Camel notation for variables, methods, and parameters.
  - First letter each word except first word should be capital.
  - For example:

```
public double calculateTotalSalary(double basicSalary, double
incentives) {
    double totalSalary = basicSalary + incentives;
    return totalSalary;
}
```

- Pascal notation for type names (i.e. class, interface, enum)
  - First letter each word should be capital.
  - For example:

```
class CompanyEmployeeManagement {
    // ...
}
```

- Package names must be in lower case only.
  - For example: javax.servlet.http;
- Constant fields must be in upper case only.
  - For example:

```
final double PI = 3.14;
final int WEEKDAYS = 7;
final String COMPANY_NAME = "Sunbeam Infotech";
```

Keywords

- Keywords are the words whose meaning is already known to Java compiler.

- These words are reserved i.e. cannot be used to declare variable, function or class.
- Java 8 Keywords
    1. abstract - Specifies that a class or method will be implemented later, in a subclass
    2. assert - Verifies the condition. Throws error if false.
    3. boolean- A data type that can hold true and false values only
    4. break - A control statement for breaking out of loops.
    5. byte - A data type that can hold 8-bit data values
    6. case - Used in switch statements to mark blocks of text
    7. catch - Catches exceptions generated by try statements
    8. char - A data type that can hold unsigned 16-bit Unicode characters
    9. class - Declares a new class
    10. continue - Sends control back outside a loop
    11. default - Specifies the default block of code in a switch statement
    12. do - Starts a do-while loop
    13. double - A data type that can hold 64-bit floating-point numbers
    14. else - Indicates alternative branches in an if statement
    15. enum - A Java keyword is used to declare an enumerated type. Enumerations extend the base class.
    16. extends - Indicates that a class is derived from another class or interface
    17. final - Indicates that a variable holds a constant value or that a method will not be overridden
    18. finally - Indicates a block of code in a try-catch structure that will always be executed
    19. float - A data type that holds a 32-bit floating-point number
    20. for - Used to start a for loop
    21. if - Tests a true/false expression and branches accordingly
    22. implements - Specifies that a class implements an interface
    23. import - References other classes
    24. instanceof - Indicates whether an object is an instance of a specific class or implements an interface
    25. int - A data type that can hold a 32-bit signed integer
    26. interface- Declares an interface
    27. long - A data type that holds a 64-bit integer
    28. native - Specifies that a method is implemented with native (platform-specific) code
    29. new - Creates new objects
    30. null - This indicates that a reference does not refer to anything
    31. package - Declares a Java package
    32. private - An access specifier indicating that a method or variable may be accessed only in the class it’s declared in
    33. protected - An access specifier indicating that a method or variable may only be accessed in the class it’s declared in (or a subclass of the class it’s declared in or other classes in the same package)
    34. public - An access specifier used for classes, interfaces, methods, and variables indicating that an item is accessible throughout the application (or where the class that defines it is accessible)
    35. return - Sends control and possibly a return value back from a called method
    36. short - A data type that can hold a 16-bit integer
    37. static - Indicates that a variable or method is a class method (rather than being limited to one particular object)

38. strictfp - A Java keyword is used to restrict the precision and rounding of floating-point calculations to ensure portability.
39. super - Refers to a class's base class (used in a method or class constructor)
40. switch - A statement that executes code based on a test value
41. synchronized - Specifies critical sections or methods in multithreaded code
42. this - Refers to the current object in a method or constructor
43. throw - Creates an exception
44. throws - Indicates what exceptions may be thrown by a method
45. transient - Specifies that a variable is not part of an object's persistent state
46. try - Starts a block of code that will be tested for exceptions
47. void - Specifies that a method does not have a return value
48. volatile - This indicates that a variable may change asynchronously
49. while - Starts a while loop
50. goto, const - Unused keywords (Reserved words)
51. true, false, null - Literals (Reserved words)

Data types

- Data type describes:
  - Memory is required to store the data
  - Kind of data memory holds
  - Operations to perform on the data
- Java is strictly type checked language.
- In java, data types are classified as:
  - Primitive types or Value types
  - Non-primitive types or Reference types

```
Data types
    |- Primitive types (Value types)
    |      |- Boolean: boolean
    |      |- Character: char
    |      |- Integral: byte, short, int, long
    |      |- Floating-point: float, double
    |
    |- Non-Primitive types (Reference types)
           |- class
           |- interface
           |- enum
           |- Array
```

1. boolean (size is not specified)
2. byte (size is 1 byte)
3. char (size is 2 bytes)
4. short (size is 2 bytes)
5. int (size is 4 bytes)
6. float (size is 4 bytes)
7. double (size is 8 bytes)

8. long (size is 8 bytes)

- primitive types(boolean, byte, char, short, int ,float, double, long) are not classes in Java.

```
Stack<int> s1 =  new Stack<int>( ); //Not OK
Stack<Integer> s1 =  new Stack<Integer>( ); //OK
```

Datatype	Detail	Default	Memory needed (size)	Examples	Range of Values
boolean	It can have value true or false, used for condition and as a flag.	false	1 bit	true, false	true or false
byte	Set of 8 bits data	0	8 bits	NA	-128 to 127
char	Used to represent chars	\u0000	16 bits	“a”, “b”, “c”, “A” and etc.	Represents 0-256 ASCII chars
short	Short integer	0	16 bits	NA	-32768-32768
int	integer	0	32 bits	0, 1, 2, 3, -1, -2, -3	-2147483648 to 2147483647-
long	Long integer	0	64 bits	1L, 2L, 3L, -1L, -2L, -3L	-922337203685477580 7 to 9223372036854775807
float	IEEE 754 floats	0.0	32 bits	1.23f, -1.23f	Upto 7 decimal
double	IEEE 754 floats	0.0	64 bits	1.23d, -1.23d	Upto 16 decimal

- Widening: We can convert state of object of narrower type into wider type. it is called as "widening".

```
int num1 = 10;
double num2 = num1; //widening
```

- Narrowing: We can convert state of object of wider type into narrower type. It is called "narrowing".

```
double num1 = 10.5;
int num2 = (int) num1; //narrowing
```

- Rules of conversion
  - source and destination must be compatible i.e. destination data type must be able to store larger/equal magnitude of values than that of source data type.
  - Rule 1: Arithmetic operation involving byte, short automatically promoted to int.

    - Rule 2: Arithmetic operation involving int and long promoted to long.
    - Rule 3: Arithmetic operation involving float and long promoted to float.
    - Rule 4: Arithmetic operation involving double and any other type promoted to double.
-  Type Conversions

Literals

- Six types of Literals:
    - Integral Literals
    - Floating-point Literals
    - Char Literals
    - String Literals
    - Boolean Literals
    - null Literal

Integral Literals

- Decimal: It has a base of ten, and digits from 0 to 9.
- Octal: It has base eight and allows digits from 0 to 7. Has a prefix 0.
- Hexadecimal: It has base sixteen and allows digits from 0 to 9 and A to F. Has a prefix 0x.
- Binary: It has base 2 and allows digits 0 and 1.
- For example:

```
int x = 65; // decimal const don't need prefix
int y = 0101; // octal values start from 0
int z = 0x41; // hexadecimal values start from 0x
int w = 0b01000001; // binary values start with 0b
```

- Literals may have suffix like U, L.
    - L -- represents long value.

```
long x = 123L; // long const assigned to long variable
long y = 123; // int const assigned to long variable -- widening
```

Floating-Point Literals

- Expressed using decimal fractions or exponential (e) notation.
- Single precision (4 bytes) floating-point number. Suffix f or F.
- Double precision (8 bytes) floating-point number. Suffix d or D.
- For example:

```
float x = 123.456f;
float y = 1.23456e+2;   // 1.23456 x 10^2 = 123.456
double z = 3.142857d;
```

Char Literals

- Each char is internally represented as integer number - ASCII/Unicode value.
- Java follows Unicode char encoding scheme to support multiple langauges.
- For example:

```
char x = 'A';       // char representation
char y = '\101';    // octal value
char z = '\u0041';  // unicode value in hex
char w = 65;        // unicode value in dec as int
```

- There are few special char literals referred as escape sequences.
  - \n -- newline char -- takes cursor to next line
  - \r -- carriage return -- takes cursor to start of current line
  - \t -- tab (group of 8 spaces)
  - \b -- backspace -- takes cursor one position back (on same line)
  - ' -- single quote
  - " -- double quote
  - \ -- prints single \
  - \0 -- ascii/unicode value 0 -- null character

String Literals

- A sequence of zero or more unicode characters in double quotes.
- For example:

```
String s1 = "Sunbeam";
```

Boolean Literals

- Boolean literals allow only two values i.e. true and false. Not compatible with 1 and 0.
- For example:

```
boolean b = true;
boolean d = false;
```

Null Literal

- "null" represents nothing/no value.
- Used with reference/non-primitive types.

```
String s = null;
Object o = null;
```

Variables

- A variable is a container which holds a value. It represents a memory location.
- A variable is declared with data type and initialized with another variable or literal.
- In Java, variable can be
    - Local: Within a method -- Created on stack.
    - Non-static/Instance field: Within a class - Accessed using object.
    - Static field: Within a class - Accessed using class-name.

Operators

- Java divides the operators into the following catgories:
    - Arithmetic operators: +, -, *, /, %
    - Assignment operators: =, +=, -=, etc.
    - Comparison operators: ==, !=, <, >, <=, >=, instanceof
    - Logical operators: &&, ||, !
        - Combine the conditions (boolean - true/false)
    - Bitwise operators: &, |, ^, ~, <<, >>, >>>
    - Misc operators: ternary ?:, dot .
        - Dot operator: ClassName.member, objName.member.

- Operator precedence and associativity

Operator	Description	Associativity
++ --	unary postfix increment unary postfix decrement	right to left
++ -- + - ! ~ (*type*)	unary prefix increment unary prefix decrement unary plus unary minus unary logical negation unary bitwise complement unary cast	right to left
* / %	multiplication division remainder	left to right
+ -	addition or string concatenation subtraction	left to right
<< >> >>>	left shift signed right shift unsigned right shift	left to right
< <= > >= instanceof	less than less than or equal to greater than greater than or equal to type comparison	left to right
== !=	is equal to is not equal to	left to right
&	bitwise AND boolean logical AND	left to right

- Operator precedence and associativity

Operator	Description	Associativity
++ --	unary postfix increment unary postfix decrement	right to left
++ -- + - ! ~ (*type*)	unary prefix increment unary prefix decrement unary plus unary minus unary logical negation unary bitwise complement unary cast	right to left
* / %	multiplication division remainder	left to right
+ -	addition or string concatenation subtraction	left to right
<< >> >>>	left shift signed right shift unsigned right shift	left to right
< <= > >= instanceof	less than less than or equal to greater than greater than or equal to type comparison	left to right
== !=	is equal to is not equal to	left to right
&	bitwise AND boolean logical AND	left to right

Wrapper types

- In Java primitive types are not classes. So their variables are not objects.
- Java has wrapper class corresponding to each primitive type. Their variables are objects.
- All wrapper classes are final classes i.e we cannot extend it.
- All wrapper classes are declared in java.lang package.

```
Object
    |- Boolean
    |- Character
    |- Number
        |- Byte
        |- Short
        |- Integer
        |- Long
```

```
        |- Float
        |- Double
```

- For every primitive, we get class in Java. It is called Wrapper class.

1. boolean => java.lang.Boolean
2. byte => java.lang.Byte
3. char => java.lang.Character
4. short => java.lang.Short
5. int => java.lang.Integer
6. float => java.lang.Float
7. double => java.lang.Double
8. long => java.lang.Long

- Applications of wrapper classes
  - Use primitive values like objects

```
// int 123 converted to Integer object holding 123.
Integer i = new Integer(123);
```

  - Convert types

```
Integer i = new Integer(123);
byte b = i.byteValue();
long l = i.longValue();
short s = i.shortValue();
double d = i.doubleValue();
String str = i.toString();

String val = "-12345";
int num = Integer.parseInt(val);
```

  - Get size and range of primitive types

```
System.out.printf("int size: %d bytes = %d bits\n", Integer.BYTES,
Integer.SIZE);
System.out.printf("int max: %d, min: %d\n", Integer.MAX_VALUE,
Integer.MIN_VALUE);
```

  - Helper/utility methods

```
System.out.println("Sum = " + Integer.sum(22, 7));
System.out.println("Max = " + Integer.max(22, 7));
System.out.println("Min = " + Integer.min(22, 7));
```

Boxing

- Converting from value (primitive) type to reference type.

```
int x = 123;
Integer y = new Integer(x); // boxing
```

- Java 5 allows auto-conversion from primitive type to corresponding wrapper type. This is called as "auto-boxing".

```
int x = 123;
Integer y = x; // auto-boxing
```

Unboxing

- Converting from reference type to value (primitive) type.

```
Integer y = new Integer(123);
int x = y.intValue(); // unboxing
```

- Java 5 allows auto-conversion from wrapper type to corresponding value type. This is called as "auto-unboxing".

```
Integer y = new Integer(123);
int x = y; // auto-unboxing
```

Control Statements

- By default, Java statements are executed sequentially i.e. statements are executed in order in which they appear in code.
- Java also provide statements to control the flow of execution. These are called as control statements.
- Types of control flow statements.
    - Decision Making statements
        - if statements
        - switch statement
    - Loop statements
        - do while loop
        - while loop
        - for loop
        - for-each loop

        - labeled loop
    - Jump statements
        - break statement
        - continue statement
        - return statement
- Being structured programming language, control flow statements (blocks) can be nested within each other.

if statements

- In Java, conditions are boolean expressions that evaluate to true or false.
- Program execution proceeds depending on condition (true or false).
- Syntax:

```
if(condition) {
    // execute when condition is true
}
```

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

```
if(condition1) {
    // execute when condition1 is true
}
else if(condition2) {
    // execute when condition2 is true
}
else {
    // execute when condition no condition is true
}
```

switch statements

- Selects a code block to be executed based on given expression.
- The expression can be integer, character or String type.

```
switch (expression) {
    case value1:
        // executed when expression equals value1
    break;
```

```
    case value2:
        // executed when expression equals value2
    break;
    // ...
    default:
        // executed when expression not equals any case constant
}
```

- We can use String constants/expressions for switch case in Java (along with integer and char types).

```
String course = "DAC";
switch(course) {
case "DAC":
    System.out.println("Welcome to DAC!");
    break;
case "DMC":
    System.out.println("Welcome to DMC!");
    break;
case "DESD":
    System.out.println("Welcome to DESD!");
    break;
default:
    System.out.println("Welcome to C-DAC!");
}
```

do-while loop

- Executes loop body and then evaluate the condition.
- If condition is true, loop is repeated again.

```
do {
    // body
} while(condition);
```

- Typically used to implement menu-driven program with swicth-case.

while loop

- Evaluate the condition and repeat the body if condition is true.

```
initialization;
while(condition) {
    body;
    modification;
}
```

for loop

- Initialize, evaluate the condition, execute the body if condition is true and then execute modification statement.

```
for(initialization; condition; modification) {
    body;
}
```

for-each loop

- Execute once for each element in array/collection.

```
int[] arr = { 11, 22, 33, 44 };
for(int i: arr)
    System.out.println(i);
```

break statement

- Stops switch/loop execution and jump to next statement after the switch/loop.

```
initialization;
while(condition) {
    body;
    if(stop-condition)
        break;
    modification;
}
statements after loop;
```

continue statement

- Jumps to next iteration of the loop.

```
initialization;
while(condition) {
    if(skip-condition)
        continue;
    body;
    modification;
}
statements after loop;
```

Labeled loops

- In case of nested loop, break and continue statements affect the loop in which they are placed.
- Labeled loops overcome this limitation. Programmer can choose the loop to be affected by break/continue statement.
- Labels can be used with while/for loop.

```
outer: for(int i=1; i<=3; i++) {
    middle: for(int j=1; j<=3; j++) {
        inner: for(int k=1; k<=3; k++) {
            if(i==j && j==k && i==K)
                break middle;
            System.out.printf("%d, %d, %d\n", i, j, k);
        }
    }
}
```

```
2 1 1
2 1 2
2 1 3
2 2 1
3 1 1
3 1 2
3 1 3
3 2 1
3 2 2
3 2 3
3 3 1
3 3 2
```

Ternary operator

- Ternary operator/Conditional operator

```
condition? expression1 : expression2;
```

- Equivalent if-else code

```
if(condition)
    expression1;
else
    expression2;
```

- If condition is true, expression1 is executed and if condition is false, expression2 is executed.

```
a = 10;
b = 7;
max = (a > b) ? a : b;
```

```
a = 10;
b = 17;
max = (a > b) ? a : b;
```

Java methods

- A method is a block of code (definition). Executes when it is called (method call).
- Method may take inputs known as parameters.
- Method may yield a output known as return value.
- Method is a logical set of instructions and can be called multiple times (reusability).

```
class ClassName {
    public static ret-type staticMethod(parameters) {
        // method body
    }

    public ret-type nonStaticMethod(parameters) {
        // method body
    }

    public static void main(String[] args) {
        // ...
        res1 = ClassName.staticMethod(arguments);
        ClassName obj = new ClassName();
        res2 = obj.nonStaticMethod(arguments);
    }
}
```

Class and Object

- Class is collection of logically related data members ("fields"/attributes/properties) and the member functions ("methods"/operations/messages) to operate on that data.
- A class is user defined data type. It is used to create one or more instances called as "Objects".
- Class is blueprint/prototype/template of the object; while Object is an instance of the class.
- Class is logical entity and Object represent physical real-world entity.
- Class represents group of such instances which is having common structure and common behavior.

- class is an imaginary entity.
    - Example: Car, Book, Laptop etc.
    - Class implementation represents encapsulation.
    - e.g. Human is a class and You are one of the object of the class.

```
class Human {
    int age;
    double weight;
    double height;
    // ...
    void walk() { ... }
    void talk() { ... }
    void think() { ... }
    // ...
}
```

- Since class is non-primitive/reference type in Java, its objects are always created on heap (using new operator). Object creation is also referred as "Instantiation" of the class.

```
Human obj = new Human();
obj.walk();
```

Instance

```
- Definition
    1. In java, object is also called as instance
    2. An entity, which is having physical existance is called as instance.
    3. An entity, which is having state, behavior and identity is called as
instance.
- instance is a real time entity.
- Example: "Tata Nano", "Java Complete Reference", "MacBook Air" etc.
```

- Types of methods in a Java class.
    - Methods are at class-level, not at object-level. In other words, same copy of class methods is used by all objects of the class.
    - Parameterless Constructor
        - In Java, fields have default values if unintialized. Primitive types default value is usually zero (refer data types table) and Reference type default value is null. Constructor should initialize fields to the desired values.
        - Has same name as of class name and no return type. Automatically called when object is created (with no arguments).
        - If no construcor defined in class, Java compiler provides a default construcor (Parameterless).

```
Human obj = new Human();
```

    - Parameterized Constructor

        - Constructor should initialize fields to the given values.
        - Has same name as of class name and no return type. Automatically called when object is created (with arguments).

```
Human obj = new Human(40, 76.5, 5.5);
```

    - Inspectors/Getters
        - Used to get value of the field outside the class.

```
double ht = obj.getHeight();
```

    - Mutators/Setters
        - Used to set value of the field from outside the class.
        - It modifies state of the object.

```
obj.setHeight(5.5);
```

        - Getter/setters provide "controlled access" to the fields from outside the class.
    - Facilitators
        - Provides additional functionalities like Console IO, File IO.

```
obj.display();
```

    - Other methods/Business logic methods

```
obj.talk();
```

- Executing a method on object is also interpreted as
    - Calling member function/method on object.
    - Invoking member function/method on object.
    - Doing operation on the object.
    - Passing message to the object.
- Each object has
    - State: Represents values of fields in the object.
    - Behaviour: Represents methods in the class and also depends on Object state.
    - Identity: Represents uniqueness of the object.
- Object created on heap using new operator is anonymous.

```
new Human().talk();
```

- Assigning reference doesn't create new object.

```
Human h1 = new Human(...);
Human h2 = h1;
```

How to get system date in Java?

- Using Calender class:

```
Calendar c = Calendar.getInstance();
//int day = c.get( Calendar.DAY_OF_MONTH );
int day = c.get( Calendar.DATE );
int month = c.get( Calendar.MONTH) + 1;
int year = c.get( Calendar.YEAR );
```

Initialization

```
int num1 = 10;      //Initialization
int num2 = num1;    //Initialization
```

- Initialization is the process of storing value inside variable during its declaration.
- We can initialize any variable only once.

Assignment

```
int num1 = 10;  //Initialization
//int num1 = 20;  //Not OK
num1 = 20;  //OK:   Assignment
num1 = 30;  //OK:   Assignment
```

- Assignment is the process of storing value inside variable after its declaration.
- We can do assignment multiple times.

Constructor

- It is a method of class which is used to initialize instance.
- Constructor is a special syntax of Java because:
    1. Its name and class name is always same.
    2. It doesn't have any return type
    3. It is designed to call implicitly.
    4. In the lifetime on instance, it gets called only once.

```
public Date(  ){    //Constructor of the class
    System.out.println("Inside constructor");
    Calendar c = Calendar.getInstance( );
    this.day = c.get( Calendar.DATE );
    this.month = c.get( Calendar.MONTH ) + 1;
    this.year = c.get( Calendar.YEAR );
}
```

- We can not call constructor on instance explicitly.

```
Date date = new Date(); //OK
date.Date( );   //Not OK
```

- We can use any access modifer on constructor.
- If constructor is public then we can create instance of a class inside method of same class as well as different class.
- If constructor is private then we can create instance of class inside method of same class only.
- Types of constructor in Java:
    1. Parameterless constructor.
    2. Parameterized constructor.
    3. Default constructor.

Parameterless constructor

- A constructor which do not have any parameter is called parameterless constructor.

```
    public Date(  ){   //Parameterless constructor
        Calendar c = Calendar.getInstance( );
        this.day = c.get( Calendar.DATE );
        this.month = c.get( Calendar.MONTH ) + 1;
        this.year = c.get( Calendar.YEAR );
    }
```

- If we create instance W/O passing arguments then parameterless constructor gets called.

```
Date dt1 = new Date( );  //OK
```

- In the above code, parameterless constructor will call.

Parameterized constructor

- Constructor of a class which take parameters is called parameterized constructor.

```
public Date( int day, int month, int year ){    //Parameterized constructor
    this.day = day;
    this.month = month;
    this.year = year;
}
```

- If we create instance by passing arguments then parameterized constructor gets called.

```
Date date = new Date( 23, 7, 1983 );    //OK
```

- Constructor calling sequence depends on order of instance creation.

Default constructor

- If we do not define any constructor(no parameterless, no parameterized) inside class then compiler generate one constructor for the class by default. It is called default constructor.
- Compiler generated default constructor is zero parameter i.e parameterless constructor.
- Compiler never generate default parameterized constructor. It means that, if we want to create instance with arguments then we must define parameterized constructor inside class.

Constructor chaining

- In Java, we can call constructor from another constructor. It is called constructor chaining.
- For constructor chaining, we should use this statement.
- this statement must be first statement inside constructor body.

```
class Date{
    private int day;    //Default value is 0
    private int month;  //Default value is 0
    private int year;   //Default value is 0

    public Date(  ){    //Parameterless Constructor
        this( 12, 8, 2022);    //Constructor chaining
    }
    public Date( int day, int month, int year ){    //Parameterized constructor
        this.day = day;
        this.month = month;
        this.year = year;
    }
}
```

- Using constructor chaning, we can reduce developers efforts.

this reference

- If we call non static method on instance then non static method get this reference.

- this reference contains reference of current/calling instance.
- Using this reference, non static method and non static field can communicate with each other. Hence this reference is considered as a link/connection between them.
- this reference is considered as a first parameter to the method. Hence it gets space once per method call on Java Stacks.
- We can not use this reference to access local variable. We should use this reference to access non static field/method.
- Use of this reference to access non static field/method is optional.
- If name of the local variable and name of field is same then to refer field we should use this reference.

```
class Complex{
    private int real;
    public void setReal( int real ){
        this.real = real;
    }
}
```

- Definition:
    - this reference is implicit parameter available inside every non static method of the class which is used to store reference of current/calling instance.

OOPS concepts

- Following members do not get space inside instance
    1. Method parameter(e.g this reference, args etc)
    2. Method local variable
    3. Static field
    4. Static as well as non static method
    5. Constructor
- Only non static field get space inside instance.
- If we declare non static field inside class then it gets space once per instance according their order of declaration.

```
class Test{
    private int num1;
    private int num2;
    private int num3;
}
class Program{
    public static void main(String[] args) {
        Test t1 = new Test( );
        Test t2 = new Test( );
        Test t3 = new Test( );
    }
}
```

- Method do not get space inside instance. All the instances of same class share single copy of method declared inside class.
- Instances can share method by passing reference of the instance as argument to the method.

Object Oriented Programming Structure / System(OOPS)

- OOPS is not a syntax. Rather it is a process or a programming methodology that we can use to solve real world use cases / problems. In other words, oops is an object oriented thought process.
- Alan Kay is inventor of oops.
- Ref: https://medium.com/javascript-scene/the-forgotten-history-of-oop-88d71b9b2d9f
- Object Oriented Analysis and Design with application: Grady Booch
- According to Grady Booch, there are 4 major and 3 minor pillars of oops.

4 Major pillars / parts / elemets

1. Abstraction : To achive simplicity
2. Encapsulation : To achive data hiding and data security
3. Modularity : To minimize module dependency
4. Hierarchy : To achive reusability

- By major, we mean that a language without any one of these elements is not object oriented.

3 Minor pillars / parts / elemets

1. Typing : To minimize maintenance of the system.
2. Concurrency : To utilize hardware resources efficiently.
3. Persistence : To maintain state of the instance on secondary storage

- By minor, we mean that each of these elements is a useful, but not essential.

Abstraction

- It is a major pillar of oops.
- Abstraction is the process of getting essential things from system/object.
- Abstraction focuses on the essential characteristics of som object, relative to the perspective of viewer. In simple word, abstraction changes from user to user.
- Abstraction describes external behavior of an instance.
- Creating instance and calling methods on it represents abstraction programatically.

```
Scanner sc = new Scanner( System.in );
String name = sc.nextLine( );
int empid = sc.nextInt( );
float salary = sc.nextFloat( );
```

```
Complex c1 = new Complex();
c1.acceptRecord( );
c1.printRecord( );
```

- Using abstraction, we can achive simplicity.

Encapsulation

- It is a major pillar of oops.
- Definition:
    1. To achive abstraction, we need to provide some implementation. This implementation of abstraction is called encapsulation.
    2. Binding of data and code together is called as encapsulation.
- Hiding represents encapsulation.

```
class Complex{
    /* Data*/
    private int real;
    private int imag;
    public Complex( ){
        this.real = 0;
        this.imag = 0;
    }
    /*Code*/
    public void acceptRecord( ){
        Scanner sc = new Scanner( System.in)
        //TODO: accept record for real and imag
    }
    public void printRecord( ){
        //TODO: print state of real and imag
    }
}
class Program{
    public static void main(String[] args) {
        //Abstraction
        Complex c1 = new Complex();
        c1.acceptRecord( );
        c1.printRecord( );
    }
}
```

- Abstraction and encapsulation are complementary concepts: Abstraction focuses on the observable behavior of an object, whereas encapsulation focuses on the implementation that gives rise to this behavior.
- Class implementation represents encapsulation.
- Use of encapsulation:
    1. To achive abstraction
    2. To achive data hiding(also called as data encapsulation).
- Process of declaring field private is called as data hiding.
- Data hiding help to achive data security.

Modularity

- It is a major pillar of oops
- Modularity is the process of designing and developing complex system using small parts/modules.
- Main purpose of modularity is to minimize module dependancy.
- Using .jar file, we can achive modularity in Java.

Hierarchy

- It is a major pillar of oops.
- Level/order/ranking of abstraction is called as hierarchy.
- Using hierarchy, we can achive code reusability.
- Application of reusability:
    1. To reduce development time.
    2. To reduce development cost
    3. To reduce develpers effort.
- Types of hierarchy:
    1. Has-a => Association
    2. Is-a => Generalization(is also called as inheritance)
    3. Use-a => Dependancy
    4. Create-a => Instantiation

Typing

- It is a minor pillar of oops.
- It is also called as polymorphism(poly(many) + morphism(forms/behavior)).
- polymorphism is a Greek word.
- An ability of any instance to take multiple forms is called as polymorphism.
- Using typing/polymorphism, we can reduce maintenance of the application.
- In Java, we can achive polymorphism using two ways:
    1. Method overloading (It represents compile time polymorphism)
    2. Method overriding (It represents run time polymorphism)
- In Java, runtime polymorphism is also called as dynamic method dispatch

Concurrency

- It is a minor pillar of oops.
- Concurrency is the process of executing multiple task simultaneously.
- Using thread, we can achive Concurrency in Java.
- Using Concurrency, we can utilize H/W resources efficiently.

Persistance

- It is a minor pillar of oops.
- Process of maintaining state of instance inside file / database is called as Persistance.
- We can implement it using file handing(serialization / derserialization) and database programming(JDBC).

Core Java

"this" reference

- "this" is implicit reference variable that is available in every non-static method of class which is used to store reference of current/calling instance.
- Whenever any non-static method is called on any object, that object is internally passed to the method and internally collected in implicit "this" parameter.
- "this" is constant within method i.e. it cannot be assigned to another object or null within the method.
- Using "this" inside method (to access members) is optional. However, it is good practice for readability.
- In a few cases using "this" is necessary.
    - Unhide non-static fields from local fields.

```
double height; // "height" field
void setHeight(double height) { // "height" parameter
    height = height; // ???
}
```

    - Constructor chaining
    - Access instance of outer object.

null reference

- In Java, local variables must be initialized before use; otherwise it raise compiler error.

```
int a;
a++; // compiler error
Human obj;
obj.walk(); // compiler error
```

- In Java, reference can be initialized to null (if any specific object is not yet available). However reference must be initialized to appropriate object before its use; otherwise it will raise NullPointerException at runtime.

```
Human h = null; // reference initialized to null
h = new Human(); // reference initialized to the object
h.walk(); // invoke the method on the object
```

```
Human h = null;
h.walk(); // NullPointerException
```

Constructor chaining

- Constructor chaining is executing a constructor of the class from another constructor (of the same class).

```
Human(int age, double height, double weight) {
    this.age = age;
    this.height = height;
    this.weight = weight;
}
Human() {
    this(0, 1.6, 3.3);
    // ...
}
```

- Constructor chaining (if done) must be on the very first line of the constructor.

OOP concepts

Abstraction

- Abstraction is getting essential details of the system.
- Abstraction change as per perspective of the user.
- Abstraction represents outer view of the system.
- Abstraction is based on interface/public methods of the class.

Encapsulation

- Combining information/state and complex logic/operations so that it will be easier to use is called as Encapsulation.
- Fields and methods are bound in a class so that user can create object and invoke methods (without looking into complex implementations).
- Encapsulation represents inner view of the system.
- Encapsulation and abstraction are complementary to each other.

Information hiding

- Members of class not intended to be visible outside the class can be restricted using access modifiers/specifiers.
- The "private" members can be accessed only within the class; while "public" members are accessible in as well as outside the class.
- Usually fields (data) are "private" and methods (operations) are "public".

Packages

- Packages makes Java code modular. It does better organization of the code.
- Package is a container that is used to group logically related classes, interfaces, enums, and other packages.

- Package helps developer:
    - To group functionally related types together.
    - To avoid naming clashing/collision/conflict/ambiguity in source code.
    - To control the access to types.
    - To make easier to lookup classes in Java source code/docs.
- Java has many built-in packages.
    - java.lang * --> Integer, System, String, ...
    - java.util --> Scanner, ArrayList, Date, ...
    - java.io --> FileInputStream, FileOutputStream, ...
    - java.sql --> Connection, Statement, ResultSet, ...
    - java.util.function --> Predicate, Consumer, ...
    - javax.servlet.http --> HttpServlet, ...
- Package Syntax
    - To define a type inside package, it is mandatory write package declaration statement inside .java file.
    - Package declaration statement must be first statement in .java file.
    - Types inside package called as packaged types; while others (in default package) are unpackaged types.
    - Any type can be member of single package only.
- It is standard practice to have multi-level packages (instead of single level). Typically package name is module name, dept/project name, website name in reverse order.

```
package com.sunbeaminfo.modular.corejava;
```

```
package com.sunbeaminfo.dac;
```

- Packages can be used to avoid name clashing. In other words, two packages can have classes/types with same name.

```
package com.sunbeam.tree;

class Node {
    // ...
}

public class Tree {
    // ...
}
```

```
package com.sunbeam.list;

class Node {
    // ...
}

public class List {
    // ...
}
```

Access modifiers (for types)

- default: When no specifier is mentioned.
  - The types are accessible in current package only.
- public: When "public" specifier is mentioned.
  - The types are accessible in current package as well as outside the package (using import).

Access modifiers (for type members)

- private: When "private" specifier is mentioned before the member field/method.
  - The members are accessible in current class (member OR this.member).
- default: When no specifier is mentioned before the member field/method.
  - The members are accessible in current class (member OR this.member).
  - The members are accessible in all classes in same package (obj.member OR ClassName.member).
- protected: When "protected" specifier is mentioned before the member field/method.
  - The members are accessible in current class (member OR this.member).
  - The members are accessible in all classes in same package including its sub-classes (super.member, obj.member OR ClassName.member).
  - The members are accessible in sub classes outside the package including its sub-classes (super.member).
- public: When "public" specifier is mentioned before the member field/method.
  - The members are accessible in current class (member OR this.member).
  - The members are accessible in all other classes (super.member, obj.member OR ClassName.member).
- Scopes
  - private (lowest)
  - default
  - protected
  - public (highest)

	default	private	protected	public
same class	yes	yes	yes	yes
same package subclass	yes	no	yes	yes
same package non-subclass	yes	no	yes	yes
different package subclass	no	no	yes	yes
different package non-subclass	no	no	no	yes

-

Array

- Array is collection of similar data elements. Each element is accessible using indexes (0 to n-1).
- In Java, array is non-primitive/reference type i.e. its object is created using new operator (on heap).
- Array size is fixed - given while creating array object. It cannot be modified later.
- Java array object holds its length and data elements.
- The array of primitive type holds values (0 if uninitialized) and array of non-primitive type holds references (null if uninitialized).
- Array of primitive types

```
int[] arr1 = new int[5];

int[] arr2 = new int[5] { 11, 22, 33 };
// error

int[] arr3 = new int[] { 11, 22, 33, 44, 55 };

int[] arr4 = { 11, 22, 33, 44 };
```

- Array of non-primitive types

```
Human[] arr5 = new Human[5];

Human[] arr6 = new Human[] {h1, h2, h3};
// h1, h2, h3 are Human references

Human[] arr7 = new Human[] {
    new Human(...),
    new Human(...),
    new Human(...)
};
```

- In Java, checking array bounds is responsibility of JVM. When invalid index is accessed, ArrayIndexOutOfBoundsException is thrown.
- Array types are
  - 1-D array
  - 2-D/Multi-dimensional array
  - Ragged array

1-D array

```
int[] arr = new int[5];

int[] arr = new int[4] { 10, 20, 30, 40 };
// error

int[] arr = new int[] { 11, 22, 33, 44, 55 };

int[] arr = { 11, 22, 33, 44 };

Human[] arr = new Human[5];

Human[] arr = new Human[] {h1, h2, h3};
```

- Individual element is accesses as `arr[i]`.

Arrays

2-D/Multi-dimensional array

```
double[][] arr = new double[2][3];

double[][] arr = new double[][]{ { 1.1, 2.2, 3.3 }, { 4.4, 5.5, 6.6 } };

double[][] arr = { { 1.1, 2.2, 3.3 }, { 4.4, 5.5, 6.6 } };
```

- Internally 2-D arrays are array of 1-D arrays. "arr" is array of 2 elements, in which each element is 1-D array of 3 doubles.
- Individual element is accesses as `arr[i][j]`.

Ragged array

- Ragged array is array of 1-D arrays. Each 1-D array in the ragged array may have different length.

```
int[][] arr = new int[4][];
arr[0] = new int[] { 11 };
arr[1] = new int[] { 22, 33 };
```

```
arr[2] = new int[] { 44, 55, 66 };
arr[3] = new int[] { 77, 88, 99, 110 };
for(int i=0; i<arr.length; i++) {
    for(int j=0; j<arr[i].length; j++) {
        System.out.print(arr[i][j] + ", ");
    }
}
```

Variable Arity Method

- Methods with variable number of arguments. These arguments are represented by ... and internally collected into an array.

```
public static int sum(int... arr) {
    int total = 0;
    for(int num: arr)
        total = total + num;
    return total;
}
public static void main(String[] args) {
    int result1 = sum(10, 20);
    System.out.println("Result: " + result1);
    int result2 = sum(11, 22, 33);
    System.out.println("Result: " + result2);
}
```

- If method argument is `Object... args`, it can take variable arguments of any type.
- Pre-defined methods with variable number of arguments.
    - PrintStream class: PrintStream printf(String format, Object... args);
    - String class: static String format(String format, Object... args);

Method overloading

- Methods with same name and different arguments in same scope - Method overloading.
- Arguments must differ in one of the follows
    - Count

```
static int multiply(int a, int b) {
    return a * b;
}
static int multiply(int a, int b, int c) {
    return a * b * c;
}
```

    - Type

```
static int square(int x) {
    return x * x;
}
static double square(double x) {
    return x * x;
}
```

- Order

```
static double divide(int a, double b) {
    return a / b;
}
static double divide(double a, int b) {
    return a / b;
}
```

- Constructors have same name (as of class name) and different arguments. This is referred as "Constructor overloading".
- Note that return type is NOT considered in method overloading. Following code cause error.

```
static int divide(int a, int b) {
    return a / b;
}
static double divide(int a, int b) {
    return (double)a / b;
}
```

```
int result1 = divide(22, 7);
double result2 = divide(22, 7);
// collecting return value is not mandetory
divide(22, 7);
```

Method arguments

- In Java, primitive values are passed by value and objects are passed by reference.
- Pass by reference -- Stores address of the object. Changes done in called method are available in calling method.

```
public static void testMethod(Human h) {
    h.setHeight(5.7);
}
public static void main(String[] args) {
    Human obj = new Human(40, 76.5, 5.5);
```

```
    obj.display();  // age=40, weight=76.5, height=5.5
    testMethod(obj);
    obj.display();  // age=40, weight=76.5, height=5.7
}
```

- Pass by value -- Creates copy of the variable. Changes done in called method are not available in calling method.

```
public static void swap(int x, int y) {
    int t = x;
    x = y;
    y = t;
}
public static void main(String[] args) {
    int num1 = 11, num2 = 22;
    System.out.printf("num1=%d, num2=%d\n", num1, num2);
    swap(num1, num2);
    System.out.printf("num1=%d, num2=%d\n", num1, num2);
}
```

- Pass by reference for value/primitive types can be simulated using array.

Command line arguments

- Additional data/information passed to the program while executing it from command line -- Command line arguments.

```
terminal> java pkg.Program Arg1 Arg2 Arg3
```

- These arguments are accessible in Java application as arguments to main().

```
package pkg;
class Program {
    public static void main(String[] args) {
        // ... args[0] = Arg1, args[1] = Arg2, args[2] = Arg3
    }
}
```

Core Java

Arrays

2-D/Multi-dimensional array

```
double[][] arr = new double[2][3];

double[][] arr = new double[][]{ { 1.1, 2.2, 3.3 }, { 4.4, 5.5, 6.6 } };

double[][] arr = { { 1.1, 2.2, 3.3 }, { 4.4, 5.5, 6.6 } };
```

- Internally 2-D arrays are array of 1-D arrays. "arr" is array of 2 elements, in which each element is 1-D array of 3 doubles.
- Individual element is accesses as `arr[i][j]`.

Ragged array

- Ragged array is array of 1-D arrays. Each 1-D array in the ragged array may have different length.

```
int[][] arr = new int[4][];
arr[0] = new int[] { 11 };
arr[1] = new int[] { 22, 33 };
arr[2] = new int[] { 44, 55, 66 };
arr[3] = new int[] { 77, 88, 99, 110 };
for(int i=0; i<arr.length; i++) {
    for(int j=0; j<arr[i].length; j++) {
        System.out.print(arr[i][j] + ", ");
    }
}
```

Variable Arity Method

- Methods with variable number of arguments. These arguments are represented by ... and internally collected into an array.

```
public static int sum(int... arr) {
    int total = 0;
    for(int num: arr)
        total = total + num;
    return total;
}
public static void main(String[] args) {
    int result1 = sum(10, 20);
    System.out.println("Result: " + result1);
```

```
    int result2 = sum(11, 22, 33);
    System.out.println("Result: " + result2);
}
```

- If method argument is `Object... args`, it can take variable arguments of any type.
- Pre-defined methods with variable number of arguments.
  - PrintStream class: PrintStream printf(String format, Object... args);
  - String class: static String format(String format, Object... args);

Method overloading

- Methods with same name and different arguments in same scope - Method overloading.
- Arguments must differ in one of the follows
  - Count

```
static int multiply(int a, int b) {
    return a * b;
}
static int multiply(int a, int b, int c) {
    return a * b * c;
}
```

  - Type

```
static int square(int x) {
    return x * x;
}
static double square(double x) {
    return x * x;
}
```

  - Order

```
static double divide(int a, double b) {
    return a / b;
}
static double divide(double a, int b) {
    return a / b;
}
```

- Constructors have same name (as of class name) and different arguments. This is referred as "Constructor overloading".
- Note that return type is NOT considered in method overloading. Following code cause error.

```
static int divide(int a, int b) {
    return a / b;
}
static double divide(int a, int b) {
    return (double)a / b;
}
```

```
int result1 = divide(22, 7);
double result2 = divide(22, 7);
// collecting return value is not mandetory
divide(22, 7);
```

Method arguments

- In Java, primitive values are passed by value and objects are passed by reference.
- Pass by reference -- Stores address of the object. Changes done in called method are available in calling method.

```
public static void testMethod(Human h) {
    h.setHeight(5.7);
}
public static void main(String[] args) {
    Human obj = new Human(40, 76.5, 5.5);
    obj.display();  // age=40, weight=76.5, height=5.5
    testMethod(obj);
    obj.display();  // age=40, weight=76.5, height=5.7
}
```

- Pass by value -- Creates copy of the variable. Changes done in called method are not available in calling method.

```
public static void swap(int x, int y) {
    int t = x;
    x = y;
    y = t;
}
public static void main(String[] args) {
    int num1 = 11, num2 = 22;
    System.out.printf("num1=%d, num2=%d\n", num1, num2);
    swap(num1, num2);
    System.out.printf("num1=%d, num2=%d\n", num1, num2);
}
```

- Pass by reference for value/primitive types can be simulated using array.

Command line arguments

- Additional data/information passed to the program while executing it from command line -- Command line arguments.

```
terminal> java pkg.Program Arg1 Arg2 Arg3
```

- These arguments are accessible in Java application as arguments to main().

```
package pkg;
class Program {
    public static void main(String[] args) {
        // ... args[0] = Arg1, args[1] = Arg2, args[2] = Arg3
    }
}
```

Object/Field initializer

- In C++/Java, constructor is used to initialize the fields.
- In Java, field initialization can be done using
  - Field initializer
  - Object initializer
  - Constructor
- Example:

```
class InitializerDemo {
    int num1 = 10;
    int num2;
    int num3;

    InitializerDemo() {
        num3 = 30;
    }
    // ...

    public static void main(String[] args) {
        InitializerDemo obj = new InitializerDemo();
        System.out.printf("num1=%d, num2=%d, num3=%d\n", num1, num2, num3);
    }
}
```

final variables

- In Java, const is reserved word - but not used.
- Java has final keyword instead. It can be used for

    - final variables
    - final fields
    - final methods
    - final class
- The final local variables and fields cannot be modified after initialization.
- The final fields must be initialized any of the following.
    - Field initializer
    - Object initializer
    - Constructor
- Example:

```
class FinalDemo {
    final int num1 = 10;
    final int num2;
    final int num3;

    {
        num2 = 20;
    }

    FinalDemo() {
        num3 = 30;
    }
    public void display() {
        System.out.printf("num1=%d, num2=%d, num3=%d\n", num1, num2, num3);
    }

    public static void main(String[] args) {
        final int num4 = 40;

        final FinalDemo obj = new FinalDemo();
        obj.display();
    }
}
```

static keyword

- In OOP, static means "shared" i.e. static members belong to the class (not object) and shared by all objects of the class.
- Static members are called as "class members"; whereas non-static members are called as "instance memebers".
- In Java, static keyword is used for
    - static fields
    - static methods
    - static block
    - static import
- Note that, static local variables cannot be created in Java.

Static fields

- Copies of non-static/instance fields are created one for each object.
- Single copy of the static/class field is created (in method area) and is shared by all objects of the class.
- Can be initialized by static field initializer or static block.
- Accessible in static as well as non-static methods of the class.
- Can be accessed by class name or object name outside the class (if not private). However, accessing via object name is misleading (avoid it).

Static methods

- Methods can be called from outside the class (if not private) using class name or object name. However, accessing via object name is misleading (avoid it).
- When needs to call a method without object, then make it static.
- Since static methods are designed to be called on class name, they do not have "this" reference. Hence, cannot access non-static members in the static method (directly). However, we can access them on an object reference.
- Applications
  - To initialize/access static fields.
  - Helper/utility methods

```
import java.util.Arrays;
// in main()
int[] arr = { 33, 88, 44, 22, 66 };
Arrays.sort(arr);
System.out.println(Arrays.toString(arr));
```

  - Factory method - to create object of the class

```
import java.util.Calendar;
// in main()
//Calendar obj = new Calendar(); // compiler error
Calendar obj = Calendar.getInstance();
System.out.println(obj);
```

Static field initializer

- Similar to field initializer, static fields can be initialized at declaration.

```
// static field
static double price = 5000.0;
```

static keyword

Static Method

- If we want to access non static members of the class then we should define non static method inside class.

```
class Test{
    private int num1;
    public int getNum1( ){
        return this.num1;
    }
    public void setNum1( int num1 ){
        this.num1 = num1;
    }
}
```

- If we want to access static members of the class then we should define static method inside class.

```
class Test{
    private static int num2;
    public static int getNum2( ){
        return Test.num2;
    }
    public void setNum2( int num2 ){
        Test.num2 = num2;
    }
}
```

Why static method do not get this reference?

```
- If we call non static method on instance then method gets this reference.
- Static method is designed to call on class name.
- Since static method is not designed to call on instance, it doesn't get this
reference.
```

- Since static method do not get this reference, we can not access non static members inside static method.
- In other words, static method can access only static members of the class directly.
- If we want to access non static members inside static method then we need to use instance of the class.

```
class Program{
    int num1 = 10;
    static int num2 = 20;
    public static void main( String[] args ){
        //System.out.println("Num1    :   "+num1);    //Compiler error
        Program p = new Program( );
        System.out.println("Num1    :   "+p.num1);    //OK: 10
        System.out.println("Num1    :   "+new Program().num1);    //OK: 10
```

```
        System.out.println("Num2    :   "+num2);    //OK: 20
        System.out.println("Num2    :   "+Program.num2);    //OK:20
    }
}
```

- Inside non static method, we can access static as well as non static members directly.

```
class Test{
    private int num1 = 10;
    private static int num2 = 20;
    public void printRecord( ){
        System.out.println("Num1    :   "+this.num1);   //OK
        System.out.println("Num2    :   "+Test.num2);   //OK
    }
}
```

- Inside method, if there is a need to use this reference then we should declare method non static otherwise we should declare method static.

```
class Math{
    public static int power( int base, int index ){
        int result = 1;
        for( int count = 1; count <= index; ++ count ){
            result = result * base;
        }
        return result;
    }
}
class Program{
    public static void main(String[] args) {
        int result = Math.power(2, 3);
        System.out.println("Result  :   "+result);
    }
}
```

- Method local variable get space once per method call.
- We can declare, method local variable final but we can not declare it static.
    - static variable is also called as class level variable.
    - class level variables should exist at class scope.
    - Hence we can not declare local variable static. But we can declare field static.

```
class Program{
    private static int number;  //OK
    public static void print( ){
        //static int number = 0;  //Not OK
        number = number + 1;
```

```
        System.out.println("Number  :   "+number);
    }
    public static void main(String[] args) {
        Program.print();    //1
        Program.print();    //2
        Program.print();    //3
    }
}
```

Static block

- Like Object/Instance initializer block, a class can have any number of static initialization blocks, and they can appear anywhere in the class body.
- Static initialization blocks are executed in the order their declaration in the class.
- A static block is executed only once when a class is loaded in JVM.
- Example:

```
class Program {
    static int field1 = 10;
    static int field2;
    static int field3;
    static final int field4;

    static {
        // static fields initialization
        field2 = 20;
        field3 = 30;
    }

    static {
        // initialization code
        field4 = 40;
    }
}
```

Static import

- To access static members of a class in the same class, the "ClassName." is optional.
- To access static members of another class, the "ClassName." is mandetory.
- If need to access static members of other class frequently, use "import static" so that we can access static members of other class direcly (without ClassName.).

```
import static java.lang.Math.*;
class Program {
   public static double calcArea(double rad) {
    return Math.PI * rad * rad;
}
}
```

Singleton class

- Design patterns are standard solutions to the well-known problems.
- Singleton is a design pattern.
- It enables access to an object throughout the application source code.
- Singleton class is a class whose single object is created throughout the application.
- To make a singleton class in Java
    - step 1: Write a class with desired fields and methods.
    - step 2: Make constructor(s) private.
    - step 3: Add a private static field to hold instance of the class.
    - step 4: Initialize the field to single object using static field initializer or static block.
    - step 5: Add a public static method to return the object.
- Code:

```
public class Singleton {
    // fields and methods
    // since ctor is declared private, object of the class cannot be created
outside the class.
    private Singleton() {
        // initialization code
    }
    // holds reference of "the" created object.
    private static Singleton obj;
    static {
        // as static block is executed once, only one object is created
        obj = new Singleton();
    }
    // static getter method so that users can access the object
    public static Singleton getInstance() {
        return obj;
    }
}
```

```
class Program {
    public static void testMethod() {
        Singleton obj2 = Singleton.getInstance();
        // ...
    }

    public static void main(String[] args) {
        Singleton obj1 = Singleton.getInstance();
        // ...
```

```
    }
}
```

Association

- If "has-a" relationship exist between the types, then use association.
- To implement association, we should declare instance/collection of inner class as a field inside another class.
- There are two types of associations
  - Composition
  - Aggregation
- Example 1:

```
public class Engine {
    // ...
}
```

```
public class Person {
    private String name;
    private int age;
    // ...
}
```

```
public class Car {
    private Engine engine;
    private Person driver;
    // ...
}
```

- Example 2:

```
public class Wall {
    // ...
}
```

```
public class Person {
    // ...
}
```

```
public class Classroom {
    private Wall[] walls = new Wall[4];
    private ArrayList<Person> students = new ArrayList<>();
    // ...
}
```

Composition

- Represents part-of relation i.e. tight coupling between the objects.
- The inner object is essential part of outer object.
    - Engine is part of Car.
    - Wall is part of ClassRoom

Aggregation

- Represents has-a relation i.e. loose coupling between the objects.
- The inner object can be added, removed, or replaced easily in outer object.
    - Car has a Driver.
    - Company has Employees.

Inheritance

- If "is-a"/"kind-of" relationship exist between the types, then use inheritance.
- Inheritance is process -- generalization to specialization.
- All members of parent class are inherited to the child class.
- Example:
    - Manager is a Employee
    - Mango is a Fruit
    - Triangle is a Shape
- In Java, inheritance is done using extends keyword.

```
class SubClass extends SuperClass {
    // ...
}
```

- Java doesn't support multiple implementation inheritance i.e. a class cannot be inherited from multiple super-classes.
- However Java does support multiple interface inheritance i.e. a class can be inherited from multiple super interfaces.

super keyword

- In sub-class, super-class members are referred using "super" keyword.
- Calling super class constructor

- By default, when sub-class object is created, first super-class constructor (param-less) is executed and then sub-class constructor is executed.
- "super" keyword is used to explicitly call super-class constructor.

```
class Person {
    // ...
    public Person(String name, int age) {
        // ...
    }
}
class Student extends Person {
    // ...
    public Student(String name, int age, int roll, double marks) {
        super(name, age); // calls parameterized ctor of super class -- must
be first line only
        // ...
    }
}
```

- Accessing super class members
  - Super class members (non-private) are accessible in sub-class directly or using "this" reference. These members can also be accessed using "super" keyword.
  - However, if sub-class method signature is same as super-class signature, it hides/shadows method of the super class i.e. super-class method is not directly visible in sub-class.
  - The "super" keyword is mandetory for accessing such hidden members of the super-class.

```
class Person {
    // ...
    public String getName() {
        // ...
    }
    public int getAge() {
        // ...
    }
    public void display() {
        // display name and age
    }
}
class Student extends Person {
    // ...
    public void display() {
        System.out.println(this.getName()); // getName() is inherited from
super-class
        System.out.println(getAge()); // getAge() is inherited from super-
class
        super.display(); // Person.display() is hidden due to
Student.display()
            // must use super keyword to call hidden method of super class.
        // display roll and marks
```

```
    }
}
```

Inheritance

Types of inheritances

- Single inheritance

```
class A {
    // ...
}
class B extends A {
    // ...
}
```

- Multiple inheritance

```
class A {
    // ...
}
class B {
    // ...
}
class C extends A, B // not allowed in Java
{
    // ...
}
```

```
interface A {
    // ...
}
interface B {
    // ...
}
class C implements A, B // allowed in Java
{
    // ...
}
```

- Hierarchial inheritance

```
class A {
    // ...
}
```

```
class B extends A {
    // ...
}
class C extends A {
    // ...
}
```

- Multi-level inheritance

```
class A {
    // ...
}
class B extends A {
    // ...
}
class C extends B {
    // ...
}
```

- Hybrid inheritance: Any combination of above types

Up-casting & Down-casting

- Up-casting: Assigning sub-class reference to a super-class reference.
    - Sub-class "is a" Super-class, so no explicit casting is required.
    - Using such super-class reference, super-class methods overridden into sub-class can also be called.

```
Employee e = new Employee();
Person p = e; // up-casting
```

- Down-casting: Assigning super-class reference to sub-class reference.
    - Every super-class is not necessarily a sub-class, so explicit casting is required.

```
Person p1 = new Employee();
Employee e1 = (Employee)p1; // down-casting - okay - Employee reference will
point to Employee object
```

```
Person p2 = new Person();
Employee e2 = (Employee)p2; // down-casting - ClassCastException - Employee
reference will point to Person object
```

Polymorphism

- Poly=Many, Morphism=Forms
- Polymorphism is taking many forms.
- OOP has two types of Polymorphism
    - Compile-time Polymorphism / Static Polymorphism
        - Implemented by "Method overloading".
        - Compiler can identify which method to be called at compile time depending on types of arguments. This is also referred as "Early binding".
    - Run-time Polymorphism / Dynamic Polymorphism
        - Implemented by "Method overriding".
        - The method to be called is decided at runtime depending on type of object. This is also referred as "Late binding" or "Dyanmic method dispatch".
    - Process of calling method of sub class using reference of super class is called as dynamic method dispatch. ###Access Modifier
    - private (lowest)
    - default
    - protected
    - public (highest)

Method overriding

- Redefining a super-class method in sub-class with exactly same signature is called as "Method overriding".
- Programmer should override a method in sub-class in one of the following scenarios
    - Super-class has not provided method implementation at all (abstract method).
    - Super-class has provided partial method implementation and sub-class needs additional code. Here sub-class implementation may call super-class method (using super keyword).
    - Sub-class needs different implementation than that of super-class method implementation.
- Rules of method overriding in Java
    - Each method in Java can be overridden unless it is private, static or final.
    - Sub-class method must have same or wider access modifier than super-class method.

```
class SuperClass {
    protected Number calculate(Integer i, Float f) {
        // ...
    }
}
class SubClass extends SuperClass {
    /*protected or*/ public Number calculate(Integer i, Float f) {
        // ...
    }
}
```

    - Arguments of sub-class method must be same as of super-class method. The return-type of sub-class method can be same or sub-class of the super-class's method's return-type. This is called as "covarient" return-type.

```
class SuperClass {
    public Number calculate(Integer i, Float f) {
        // ...
    }
}
class SubClass extends SuperClass {
    // Double is inherited from Numer (i.e. return-type of super-class
method)
    public Double calculate(Integer i, Float f) {
        // ...
    }
}
```

- Checked exception list in sub-class method should be same or subset of exception list in super-class method.

```
class SuperClass {
    public void testMethod() throws IOException, SQLException {
        // ...
    }
}
class SubClass extends SuperClass {
    public void testMethod() throws IOException {
        // ...
    }
}
```

- If these rules are not followed, compiler raises error or compiler treats sub-class method as a new method.

```
class A {
    void method1() {
    }
    void method2() {
    }
}
```

```
class B extends A {
    void method1() {
        // overridden
    }
    void method2(int x) {
        // treated as new method
    }
}
```

```
// In main()
A obj = new B();
obj.method1();  // B.method1() -- overridden
obj.method2();  // A.method2() -- method2() is not overridden
```

- Java 5.0 added @Override annotation (on sub-class method) informs compiler that programmer is intending to override the method from the super-class.
- @Override checks if sub-class method is compatible with corresponding super-class method or not (as per rules). If not compatible, it raise compile time error.
- Note that, @Override is not compulsory to override the method. But it is good practice as it improves readability and reduces human errors.

final variables/fields

- Cannot be modified once initialized.
- Refer earlier notes.

final method

- If implementation of a super-class method is logically complete, then the method should be declared as final.
- Such final methods cannot be overridden in sub-class. Compiler raise error, if overridden.
- But final methods are inherited into sub-class i.e. The super-class final methods can be invoked in sub-class object (if accessible).

final class

- If implementation of a super-class is logically complete, then the class should be declared as final.
- The final class cannot be extended into a sub-class. Compiler raise error, if inherited.
- Effectively all methods in final class are final methods.
- Examples of final classes
    - java.lang.Integer (and all wrapper classes)
    - java.lang.String
    - java.lang.System

Object class

- Non-final and non-abstract class declared in java.lang package.
- In java, all the classes (not interfaces) are directly or indirectly extended from Object class.
- In other words, Object class is ultimate base class/super class.
- Object class is not inherited from any class or implement any interface.
- It has a default constructor.
    - Object o = new Object();

Object class methods (read docs)

- Parameter less constructor
    - public Object();
- Returns string representation of object state
    - public String toString();
- Comparing current object with another object
    - public boolean equals(Object);
- Used while storing object into set or map collections
    - public native int hashCode();
- Create shallow copy of the object
    - protected native Object clone() throws CloneNotSupportedException;
- Called by garbage collector (like C++ destructor)
    - protected void finalize() throws Throwable;
- Get metadata about the class
    - public final native Class<?> getClass();

- For thread synchronization
  - public final native void notify();
  - public final native void notifyAll();
  - public final void wait() throws InterruptedException;
  - public final native void wait(long) throws InterruptedException;
  - public final void wait(long, int) throws InterruptedException;

toString() method

- Non-final method of java.lang.Object class.
  - public String toString();
- Definition of Object.toString():

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

- To return state of Java instance in String form, programmer should override toString() method.
- The result in toString() method should be a concise, informative, and human-readable.
- It is recommended that all subclasses override this method.
- Example:

```
class Person {
    // ...
    @Override
    public String toString() {
        return "Name=" + this.name + ", Age=" + this.age;
    }
}
```

Inheritance vs Association

- Inheritance: is-a relation
  - Book is-a Product
  - Album is-a Product
  - Labor is-a Employee
  - Employee is-a Person
  - Batter is-a Player
  - ...
- Association: has-a relation
  - Employee has-a joining Date
  - Person has-a birth Date
  - Cart has Products
  - Bank has Accounts
  - ...

abstract keyword

- In Java, abstract keyword is used for
  - abstract method
  - abstract class

Fragile base class problem

- If changes are done in super-class methods (signatures), then it is necessary to modify and recompile all its sub-classes. This is called as "Fragile base class problem".
- This can be overcomed by using interfaces.

```
class A{
    public void print( ){
        //System.out.print("Hello,");
        System.out.print("Good Morning,");
    }
}
class B extends A{
    @Override
    public void print( ){
        super.print();
        System.out.println("Have a nice day!!");
    }
}
class C extends A{
    @Override
    public void print( ){
        super.print();
        System.out.println("Good day!!");
    }
}
class Program{
    public static void main(String[] args) {
        A a = null;
        a = new B();    a.print();  //Good Morning,,Have a nice day!!
        a = new C();    a.print();  //Good Morning,,Good day!!
    }
}
```

Interface (Java 7 or Earlier)

- Interfaces are used to define standards/specifications. A standard/specification is set of rules.
- Interfaces are immutable i.e. once published interface should not be modified.
- Interfaces contains only method declarations. All methods in an interface are by default abstract and public.
- They define a "contract" that is must be followed/implemented by each sub-class.

```
interface Displayable {
    public abstract void display();
}
```

```
interface Acceptable {
    abstract void accept(Scanner sc);
}
```

```
interface Shape {
    double calcArea();
    double calcPeri();
}
```

- Interfaces enables loose coupling between the classes i.e. a class need not to be tied up with another class implementation.
- Interfaces cannot be instantiated, they can only be implemented by classes or extended by other interfaces.
- Java 7 interface can only contain public abstract methods and static final fields (constants). They cannot have non-static fields, non-static methods, and constructors.
- Examples:
    - java.io.Closeable / java.io.AutoCloseable
    - java.lang.Runnable
    - java.util.Collection, java.util.List, java.util.Set, ...
- Example 1: Multiple interface inheritance is allowed.

```
interface Displayable {
    void display();
}
interface Acceptable {
    void accept();
}

class Person implements Acceptable, Displayable {
    // ...
    public void accept() {
        // ...
    }
    public void display() {
        // ...
    }
}
```

- Example 2: Interfaces can have public static final fields.

```
interface Shape {
    /*public static final*/ double PI = 3.142;

    /*public abstract*/ double calcArea();
    /*public abstract*/ double calcPeri();
}

class Circle implements Shape {
    private double radius;
    // ...
    public double calcArea() {
        return PI * this.radius * this.radius;
    }
    public double calcPeri() {
        return 2 * Shape.PI * this.radius;
    }
}
```

- Example 3: If two interfaces have same method, then it is implemented only once in sub-class.

```
interface Displayable {
    void print();
}
interface Showable {
    void print();
}
class MyClass implements Displayable, Showable {
    // ...
    public void print() {
        // ...
    }
}
class Program {
    public static void main(String[] args) {
        Displayable d = new MyClass();
        d.print();
        Showable s = new MyClass();
        s.print();
        MyClass m = new MyClass();
        m.print();
    }
}
```

Types of inheritance in OOPS

- Interface inheritance
  - Single inheritance [Allowed in Java]
  - Multiple inheritance [Allowed in Java]
  - Hierarchical inheritance [Allowed in Java]

    - Multilevel inheritance [Allowed in Java]
- Implementation inheritance
    - Single inheritance [Allowed in Java]
    - Multiple inheritance [Not Allowed in Java]
    - Hierarchical inheritance [Allowed in Java]
    - Multilevel inheritance [Allowed in Java]

- Interface syntax
    - Interface : I1, I2, I3
    - Class : C1, C2, C3
    - class C1 implements I1 // okay
    - class C1 implements I1, I2 // okay
    - interface I2 implements I1 // error
    - interface I2 extends I1 // okay
    - interface I3 extends I1, I2 // okay
    - class C2 implements C1 // error
    - class C2 extends C1 // okay
    - class C3 extends C1, C2 // error
    - interface I1 extends C1 // error
    - interface I1 implements C1 // error
    - class C2 implements I1, I2 extends C1 // error
    - class C2 extends C1 implements I1,I2 // okay

abstract method

- If implementation of a method in super-class is not possible/incomplete, then method is declared as abstract.
- Abstract method does not have definition/implementation.

```
// Employee class
abstract double calcTotalSalary();
```

- If class contains one or more abstract methods, then class must be declared as abstract. Otherwise compiler raise an error.
- The super-class abstract methods must be overridden in sub-class; otherwise sub-class should also be marked abstract.
- The abstract methods are forced to be implemented in sub-class. It ensures that sub-class will have corresponding functionality.
- The abstract method cannot be private, final, or static.
- Example: abstract methods declared in Number class are:
    - abstract int intValue();
    - abstract float floatValue();
    - abstract double doubleValue();
    - abstract long longValue();

abstract class

- If implementation of a class is logically incomplete, then the class should be declared abstract.
- If class contains one or more abstract methods, then class must be declared as abstract.
- An abstract class can have zero or more abstract methods.
- Abstract class object cannot be created; however its reference can be created.
- Abstract class can have fields, methods, and constructor.
- Its constructor is called when sub-class object is created and initializes its (abstract class) fields.
- If object of a class is not logical (corresponds to real-world entity), then class can be declared as abstract.
- Example:
    - java.lang.Number
    - java.lang.Enum

class vs abstract class vs interface

- class
    - Has fields, constructors, and methods
    - Can be used standalone -- create objects and invoke methods
    - Reused in sub-classes -- inheritance
    - Can invoke overridden methods in sub-class using super-class reference -- runtime polymorphism
- abstract class
    - Has fields, constructors, and methods
    - Cannot be used independently -- can't create object
    - Reused in sub-classes -- inheritance -- Inherited into sub-class and must override abstract methods
    - Can invoke overridden methods in sub-class using super-class reference -- runtime polymorphism
- interface
    - Has only method declarations
    - Cannot be used independently -- can't create object
    - Doesn't contain anything for reusing (except static final fields)
    - Used as contract/specification -- Inherited into sub-class and must override all methods
    - Can invoke overridden methods in sub-class using super-class reference -- runtime polymorphism
    - Java support multiple interface inheritance

What is the difference between abstract class and interface? / When we should use abstract class and interface?

Abstract class

```
abstract class Shape{
    public abstract void calculateArea( );
}
class Rectangle extends Shape{
    @Override
    public void calculateArea(){
```

```
        //TODO
    }
}
class Circle extends Shape{
    @Override
    public void calculateArea(){
        //TODO
    }
}
class Triangle extends Shape{
    @Override
    public void calculateArea(){
        //TODO
    }
}
```

- If "is-a" relationship is exist between super type & sub type and if we want to maintain same method signature/design in all the sub clases then we should declare super type abstract.

```
Shape[] arr = new Shape[ 3 ];
arr[ 0 ] = new Rectangle();
arr[ 1 ] = new Circle();
arr[ 2 ]  = new Triangle();
```

- Using abstract class, we can group instances of related type together.
- Abstract class can extend only one abstract class / concrete class. In other words, using abstract class, we can not achieve multiple inheritance.
- We can define constructor inside abstract class.
- Abstract class may / may not contain abstract method.
- In General, if state is involved in super type then super type should be abstract class.

Interface

```
interface Printable{
    void printRecord( );
}
class Complex implements Printable{
    @Override
    public void printRecord(){
        System.out.println("Print complex number");
    }
}
class Point implements Printable{
    @Override
    public void printRecord(){
        System.out.println("Print point");
    }
```

```
}
class Date implements Printable{
    @Override
    public void printRecord(){
        System.out.println("Print date");
    }
}
```

- If "is-a" relationship is not exist(can-do relationship is exist) between super type & sub type and if we want to maintain same method design in all the sub classes then we should declare super type interface.

```
Printable[] arr = new Printable[ 3 ];
arr[ 0 ] = new Complex();
arr[ 1 ] = new Point();
arr[ 2 ] = new Date();
```

- Using interface, we can group instances of unrelated type together.
- Interface can extend more than one interfaces. In other words, using interface, we can achive multiple inheritance.
- We can not define constructor inside interface.
- Interface methods are by default abstract.
- In General, if state is not involved in super type then super type should be interface.

equals() method

- Non-final method of java.lang.Object class.
    - public boolean equals(Object other);
- Definition of Object.equals():

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

- To compare the object contents/state, programmer should override equals() method.
- This equals() must have following properties:
    - Reflexive: for any non-null reference value x, x.equals(x) should return true.
    - Symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
    - Transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
    - Consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
    - For any non-null reference value x, x.equals(null) should return false.

- Example:

```
class Employee {
    // ...
    @Override
    public boolean equals(Object obj) {
        if(obj == null)
            return false;

        if(this == obj)
            return true;

        if(! (obj instanceof Employee))
            return false;

        Employee other = (Employee) obj;
        if(this.id == other.id)
            return true;
        return false;
    }
}
```

Cloneable interface

- Enable creating copy/clone of the object.
- If a class is Cloneable, Object.clone() method creates a shallow copy of the object. If class is not Cloneable, Object.clone() throws CloneNotSupportedException.
- A class should implement Cloneable and override clone() to create a deep/shallow copy of the object.

```
class Date implements Cloneable {
    private int day, month, year;
    // ...
    // shallow copy
    public Object clone() throws CloneNotSupportedException {
        Date temp = (Date)super.clone();
        return temp;
    }
}
```

Garbage collection

- Garbage collection is automatic memory management by JVM.
- If a Java object is unreachable (i.e. not accessible through any reference), then it is automatically released by the garbage collector.
- An object become eligible for GC in one of the following cases:
    - Nullify the reference.

```
MyClass obj = new MyClass();
obj = null;
```

    - Reassign the reference.

```
MyClass obj = new MyClass();
obj = new MyClass();
```

    - Object created locally in method.

```
void method() {
    MyClass obj = new MyClass();
    // ...
}
```

- GC is a background thread in JVM that runs periodically and reclaim memory of unreferenced objects.
- Before object is destroyed, its finalize() method is invoked (if present).
- One should override this method if object holds any resource to be released explicitly e.g. file close, database connection, etc.

```
class MyClass {
    private Connection con;
    public MyClass() throws Exception {
        con = DriverManager.getConnection("url", "username", "password");
    }
    // ...
    @Override
    public void finalize() {
        try {
            if(con != null)
                con.close();
        }
        catch(Exception e) {
```

```
        }
    }
}
class Main {
    public static void method() throws Exception {
        MyClass my = new MyClass();
        my = null;
        System.gc(); // request GC
    }
    // ...
}
```

- GC can be requested (not forced) by one of the following.
    - System.gc();
    - Runtime.getRuntime().gc();
- JVM GC internally use Mark and Compact algorithm.
- GC Internals: https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

Marker interfaces

- Interface that doesn't contain any method declaration is called as "Marker interface".
- These interfaces are used to mark or tag certain functionalities/features in implemented class. In other words, they associate some information (metadata) with the class.
- Marker interfaces are used to check if a feature is enabled/allowed for the class.
- Java has a few pre-defined marker interfaces. e.g. Serializable, Cloneable, etc.
    - java.io.Serializable -- Allows JVM to convert object state into sequence of bytes.
    - java.lang.Cloneable -- Allows JVM to create copy of the class object.

Cloneable interface

- Enable creating copy/clone of the object.
- If a class is Cloneable, Object.clone() method creates a shallow copy of the object. If class is not Cloneable, Object.clone() throws CloneNotSupportedException.
- A class should implement Cloneable and override clone() to create a deep/shallow copy of the object.

```
class Date implements Cloneable {
    private int day, month, year;
    // ...
    // shallow copy
    public Object clone() throws CloneNotSupportedException {
        Date temp = (Date)super.clone();
        return temp;
    }
}
```

```
class Person implements Cloneable {
    private String name;
    private int weight;
    private Date birth;
    // ...
    // deep copy
    public Object clone() throws CloneNotSupportedException {
        Person temp = (Person)super.clone(); // shallow copy
        temp.birth = (Date)this.birth.clone(); // + copy reference types
explicitly
        return temp;
    }
}
```

```
class Program {
    public static void main(String[] args) throws CloneNotSupportedException
{
        Date d1 = new Date(28, 9, 1983);
        System.out.println("d1 = " + d1.toString());
        Date d2 = (Date)d1.clone();
        System.out.println("d2 = " + d2.toString());
        Person p1 = new Person("Nilesh", 70, d1);
        System.out.println("p1 = " + p1.toString());
        Person p2 = (Person)p1.clone();
        System.out.println("p2 = " + p2.toString());
    }
}
```

Java Strings

- java.lang.Character is wrapper class that represents char. In Java, each char is 2 bytes because it follows unicode encoding.
- String is sequence of characters.
    1. java.lang.String: "Immutable" character sequence
    2. java.lang.StringBuffer: Mutable character sequence (Thread-safe)
    3. java.lang.StringBuilder: Mutable character sequence (Not Thread-safe)
- String helpers
    1. java.util.StringTokenizer: Helper class to split strings

String objects

- java.lang.String is class and strings in java are objects.
- String constants/literals are stored in string pool.

```
String str1 = "Sunbeam";
```

- String objects created using "new" operator are allocated on heap.

```
String str2 = new String("Nilesh");
```

- In java, String is immutable. If try to modify, it creates a new String object on heap.

String literals

- Since strings are immutable, string constants are not allocated multiple times.
- String constants/literals are stored in string pool. Multiple references may refer the same object in the pool.
- String pool is also called as String literal pool or String constant pool.
- From Java 7, String pool is in the heap space (of JVM).
- The string literal objects are created during class loading.

String objects vs String literals

- Example 01:

```
String s1 = "Sunbeam";
String s2 = "Sunbeam";
System.out.println(s1 == s2);       // ???
System.out.println(s1.equals(s2));  // ???
```

- Example 02:

```
String s1 = new String("Sunbeam");
String s2 = new String("Sunbeam");
System.out.println(s1 == s2);       // ???
System.out.println(s1.equals(s2));  // ???
```

- Example 03:

```
String s1 = "Sunbeam";
String s2 = new String("Sunbeam");
System.out.println(s1 == s2);       // ???
System.out.println(s1.equals(s2));  // ???
```

- Example 04:

```
String s1 = "Sunbeam";
String s2 = "Sun" + "beam";
```

```
System.out.println(s1 == s2);       // ???
System.out.println(s1.equals(s2));  // ???
```

- Example 05:

```
String s1 = "Sunbeam";
String s2 = "Sun";
String s3 = s2 + "beam";
System.out.println(s1 == s3);       // ???
System.out.println(s1.equals(s3));  // ???
```

- Example 06:

```
String s1 = "Sunbeam";
String s2 = new String("Sunbeam").intern();
System.out.println(s1 == s2);       // ???
System.out.println(s1.equals(s2));  // ???
```

- Example 07:

```
String s1 = "Sunbeam";
String s2 = "SunBeam";
System.out.println(s1 == s2);       // ???
System.out.println(s1.equals(s2));  // ???
System.out.println(s1.equalsIgnoreCase(s2));  // ???
System.out.println(s1.compareTo(s2));  // ???
System.out.println(s1.compareToIgnoreCase(s2));  // ???
```

String operations

- int length()
- char charAt(int index)
- int compareTo(String anotherString)
- boolean equals(String anotherString)
- boolean equalsIgnoreCase(String anotherString)
- boolean matches(String regex)
- boolean isEmpty()
- boolean startsWith(String prefix)
- boolean endsWith(String suffix)
- int indexOf(int ch)
- int indexOf(String str)
- String concat(String str)
- String substring(int beginIndex)
- String substring(int beginIndex, int endIndex)

- String[] split(String regex)
- String toLowerCase()
- String toUpperCase()
- String trim()
- byte[] getBytes()
- char[] toCharArray()
- String intern()
- static String valueOf(Object obj)
- static String format(String format, Object... args)

StringBuffer vs StringBuilder

- StringBuffer and StringBuilder are final classes declared in java.lang package.
- It is used create to mutable string instance.
- equals() and hashCode() method is not overridden inside it.
- Can create instances of these classes using new operator only. Objects are created on heap.
- StringBuffer capacity grows if size of internal char array is less (than string to be stored).
    - The default capactiy is 16.

```
int max = (minimumCapacity > value.length? value.length * 2 + 2 :
value.length);
minimumCapacity = (minimumCapacity < max? max : minimumCapacity);
char[] nb = new char[minimumCapacity];
```

- StringBuffer implementation is thread safe while StringBuilder is not thread-safe.
- StringBuilder is introduced in Java 5.0 for better performance in single threaded applications.

Examples

- Example 01:

```
StringBuffer s1 = new StringBuffer("Sunbeam");
StringBuffer s2 = new StringBuffer("Sunbeam");
System.out.println(s1 == s2);       // false
System.out.println(s1.equals(s2));  // false
```

- Example 02:

```
StringBuffer s1 = new StringBuffer("Sunbeam");
String s2 = new String("Sunbeam");
System.out.println(s1 == s2);       // false
System.out.println(s1.equals(s2));  // false
```

- Example 03:

```
String s1 = new String("Sunbeam");
StringBuffer s2 = new StringBuffer("Sunbeam");
System.out.println(s1.equals(s2));  // false -- String compared with
StringBuffer
System.out.println(s1.equals(s2.toString()));  // true -- String compared
with String
```

- Example 04:

```
StringBuffer s1 = new StringBuffer("Sunbeam");
StringBuffer s2 = s1.reverse();
System.out.println(s1 == s2);       // true
System.out.println(s1.equals(s2));  // true
```

- Example 05:

```
StringBuilder s1 = new StringBuilder("Sunbeam");
StringBuilder s2 = new StringBuilder("Sunbeam");
System.out.println(s1 == s2);       // false
System.out.println(s1.equals(s2));  // false -- calls Object.equals()
```

- Example 06:

```
StringBuffer s = new StringBuffer();
System.out.println("Capacity: " + s.capacity() + ", Length: " + s.length());
// 16, 0
s.append("1234567890");
System.out.println("Capacity: " + s.capacity() + ", Length: " + s.length());
// 16, 10
s.append("ABCDEFGHIJKLMNOPQRSTUV");
System.out.println("Capacity: " + s.capacity() + ", Length: " + s.length());
// 34, 32
```

Resource Management

- System resources should be released immediately after the use.
- Few system resources are Memory, File, IO Devices, Socket/Connection, etc.
- The Garbage collector automatically releases memory if objects are no more used (unreferenced).
- The GC collector doesn't release memory/resources immediately; rather it is executed only memory is full upto a threshold.
- The standard way to release the resources immediately after their use is java.io.Closeable interface. It has only one method.
    - void close() throws IOException;
- Programmer should call close() explicitly on resource object after its use.

    - e.g. FileInputStream, FileOutputStream, etc.
- Java 7 introduced an interface java.lang.AutoCloseable as super interface of Closeable. It has only one method.
    - void close() throws Exception;
- Since it is super-interface of Closeable, all classes implementing Closeable now also inherit from AutoCloseable.
- If a class is inherited from AutoCloseable, then it can be closed using try-with-resource syntax.

```
class MyResource implements AutoCloseable {
    // ...
    public void close() {
        // cleanup code
    }
}

class Program {
    public static void main(String[] args) {
        try(MyResource res = new MyResource()) {
            // ...
        } // res.close() called automatically
    }
}
```

- The Scanner class is also AutoCloseable.

```
class Program {
    public static void main(String[] args) {
        try(Scanner sc = new Scanner(System.in)) {
            // ...
        } // sc.close() is auto-closed
    }
}
```

JVM Architecture

Java compilation process

- Hello.java --> Java Compiler --> Hello.class
    - javac Hello.java
- Java compiler converts Java code into the Byte code.

Byte code

- Byte code is machine level instructions that can be executed by Java Virtual Machine (JVM).
    - Instruction = Op code + Operands
        - e.g. iadd op1, op2
- Each Instruction in byte-code is of 1 byte.

    - .class --> JVM --> Windows (x86)
    - .class --> JVM --> Linux (ARM)
- JVM converts byte-code into target machine/native code (as per architecture).

.class format

- .class file contains header, byte-code, meta-data, constant-pool, etc.
- .class header contains
    - magic number -- 0xCAFEBABE (first 4 bytes of .class file)
    - information of other sections
- .class file can inspected using "javap" tool.
    - terminal> javap java.lang.Object
        - Shows public and protected members
    - terminal> javap -p java.lang.Object
        - Shows private members as well
    - terminal> javap -c java.lang.Object
        - Shows byte-code of all methods
    - terminal> javap -v java.lang.Object
        - Detailed (verbose) information in .class
            - Constant pool
            - Methods & their byte-code
            - ...
- "javap" tool is part of JDK.

Executing Java program (.class)

- terminal> java Hello
- "java" is a Java Application Launcher.
- java.exe (disk) --> Loader --> (Windows OS) Process
- When "java" process executes, JVM (jvm.dll) gets loaded in the process.
- JVM will now find (in CLASSPATH) and execute the .class.

JVM Architecture (Overview)

- JVM = Classloader + Memory Areas + Execution Engine

Classloader sub-system

- Load and initialize the class

Loading

- Three types of classloaders
    - Bootstrap classloader: Load Java builtin classes from jre/lib jars (e.g. rt.jar).
    - Extended classloader: Load extended classes from jre/lib/ext directory.
    - Application classloader: Load classes from the application classpath.
- Reads the class from the disk and loads into JVM method (memory) area.

Linking

- Three steps: Verification, Preparation, Resolution
- Verification: Byte code verifier does verification process. Ensure that class is compiled by a valid compiler and not tampered.
- Preparation: Memory is allocated for static members and initialized with their default values.
- Resolution: Symbolic references in constant pool are replaced by the direct references.

Initialization

- Static variables of the class are assigned with given values (field initializers).
- Execute static blocks if present.

JVM memory areas

- During execution, memory is required for byte code, objects, variables, etc.
- There are five areas: Method area, Heap area, Stack area, PC Registers, Native Method Stack area.

Method area

- Created during JVM startup.
- Shared by all threads (global).
- Class contents (for all classes) are loaded into Method area.
- Method area also holds constant pool for all loaded classes.

Heap area

- Created during JVM startup.
- Shared by all threads (global).
- All allocated objects (with new keyword) are stored in heap.
- The class Metadata is stored in a java.lang.Class object (in heap) once class is loaded.
- The string pool is part of Heap area.

Stack area

- Separate stack is created for each thread in JVM (when thread is created).
- When a method is called from the stack, a new FAR (stack frame) is created on its stack.
- Each stack frame conatins local variable array, operand stack, and other frame data.
- When method returns, the stack frame is destroyed.

PC Registers

- Separate PC register is created for each thread. It maintains address of the next instruction executed by the thread.
- After an instruction is completed, the address in PC is auto-incremented.

Native method stack area

- Separate native method stack is created for each thread in JVM (when thread is created).
- When a native method is called from the stack, a stack frame is created on its stack.

Execution engine

- The main comonent of JVM.
- Execution engine executes for executing Java classes.

Interpreter

- Convert byte code into machine code and execute it (instruction by instruction).
- Each method is interpreted by the interpreter at least once.
- If method is called frequently, interpreting it each time slow down the execution of the program.
- This limitation is overcomed by JIT (added in Java 1.1).

JIT compiler

- JIT stands for Just In Time compiler.
- Primary purpose of the JIT compiler to improve the performance.
- If a method is getting invoked multile times, the JIT compiler convert it into native code and cache it.
- If the method is called next time, its cached native code is used to speedup execution process.

Profiler

- Tracks resource (memory, threads, ...) utilization for execution.
- Part of JIT that identifies hotspots. It counts number of times any method is executing. If the number is more than a threshold value, it is considered as hotspot.

Garbage collector

- When any object is unreferenced, the GC release its memory.

JNI

- JNI acts as a bridge between Java method calls and native method implementations.

```
            double result = divide(num1, num2);
            System.out.println("Result: " + result);
        }
        catch(RuntimeException e) {
            e.printStackTrace();
        }
    }
```

- Java operators/APIs throw pre-defined exception if runtime problem occurs.
- For example, ArithmeticException is thrown when divide by zero is tried.
- Example 2:

```
static int divide(int numerator, int denominator) {
    return numerator / denominator;
}
public static void main(String[] args) {
    // ...
    try {
        int num1 = sc.nextInt();
        int num2 = sc.nextInt();
        int result = divide(num1, num2);
        System.out.println("Result: " + result);
    }
    catch(ArithmeticException e) {
        e.printStackTrace();
    }
}
```

- Java exception class hierarchy

```
Object
    |- Throwable
        |- Error
        |   |- AssertionError
        |   |- VirtualMachineError
        |           |- StackOverflowError
        |           |- OutOfMemoryError
        |- Exception
            |- CloneNotSupportedException
            |- IOException
            |       |- EOFException
            |       |- FileNotFoundException
            |- SQLException
            |- InterruptedException
            |- RuntimeException
                |- NullPointerException
                |- ArithmeticException
                |- NoSuchElementException
                |       |- InputMismatchException
```

```
                    |- IndexOutOfBoundsException
                            |- ArrayIndexOutOfBoundsException
                            |- StringIndexOutOfBoundsException
```

- One catch block cannot handle problems from multiple try blocks.
- One try block may have multiple catch blocks. Specialized catch block must be written before generic catch block.
- If certain code to be executed irrespective of exception occur or not, write it in finally block.
- Example:

```
try {
    // file read code -- possible problems
        // 1. file not found
        // 2. end of file is reached
        // 3. error while reading from file
        // 4. null reference (programmer's mistake)
}
catch(NullPointerException ex) {
    // ...
}
catch(FileNotFoundException ex) {
    // ...
}
catch(EOFException ex) {
    // ...
}
catch(IOException ex) {
    // ...
}
finally {
    // close the file
}
```

- When exception is raised, it will be caught by nearest matching catch block. If no matching catch block is found, the exception will be caught by JVM and it will abort the program.

java.lang.Throwable class

- Throwable is root class for all errors and exceptions in Java.
- Only objects of
- Methods
    - Throwable()
    - Throwable(String message)
    - Throwable(Throwable cause)
    - Throwable(String message, Throwable cause)
    - String getMessage()
    - void printStackTrace()
    - void printStackTrace(PrintStream s)

    - void printStackTrace(PrintWriter s)
    - String toString()
    - Throwable getCause()

java.lang.Error class

- Usually represents the runtime problems that are not recoverable.
- Generated due to environmental condition/Runtime environment (e.g. OS error, Memory error, etc.)
- Examples:
    - AssertionError
    - VirtualMachineError
    - StackOverflowError
    - OutOfMemoryError

java.lang.Exception class

- Represents the runtime problems that can be handled.
- The exception is handled using try-catch block.
- Examples:
    - CloneNotSupportedException
    - IOException
    - SQLException
    - NullPointerException
    - ArrayIndexOutOfBoundsException
    - ClassCastException

Exception types

- There are two types of exceptions
    - Checked exception -- Checked by compiler and forced to handle
    - Unchecked exception -- Not checked by compiler

Unchecked exception

- RuntimeException and all its sub classes are unchecked exceptions.
- Typically represents programmer's or user's mistake.
    - NullPointerException, NumberFormatException, ClassCastException, etc.
- Compiler doesn't provide any checks -- if exception is handled or not.
- Programmer may or may not handle (catch block) the exception. If exception is not handled, it will be caught by JVM and abort the application.

Checked exception

- Exception and all its sub classes (except RuntimeException) are checked exceptions.
- Typically represents problems arised out of JVM/Java i.e. at OS/System level.
    - IOException, SQLException, InterruptedException, etc.
- Compiler checks if the exception is handled in one of following ways.
    - Matching catch block to handle the exception.

```
void someMethod() {
    try {
        // file io code
    }
    catch(IOException ex) {
        // ...
    }
}
```

- throws clause indicating exception to be handled by calling method.

```
void someMethod() throws IOException {
    // file io code
}
void callingMethod() {
    try {
        someMethod();
    }
    catch(IOException ex) {
        // ...
    }
}
```

Exception handling keywords

- "try" block
  - Code where runtime problems may arise should be written in try block.
  - try block must have one of the following
    - catch block
    - finally block
    - try-with-resource
  - Can be nested in try, catch, or finally block.
- "throw" statement
  - Throws an exception/error i.e. any object that is inherited from Throwable class.
  - Can throw only one exception at time.
  - All next statements are skipped and control jumps to matching catch block.
- "catch" block
  - Code to handle error/exception should be written in catch block.
  - Argument of catch block must be Throwable or its sub-class.
  - Generic catch block -- Performs upcasting -- Should be last (if multiple catch blocks)

```
try {
    // ...
}
catch(Throwable e) {
```

```
    // can handle exception of any type
}
```

- Multi-catch block -- Same logic to execute different exceptions

```
try {
    // ...
}
catch(ArithmeticException|InputMismatchException e) {
    // can handle exception of any type
}
```

- Exception sub-class should be caught before super-class.

```
try {
    // ...
}
catch(EOFException ex) {
    // ...
}
catch(IOException ex) {
    // ...
}
```

- "finally" block
  - Resources are closed in finally block.
  - Executed irrespective of exception occurred or not.
  - finally block not executed if JVM exits (System.exit()).

```
Scanner sc = new Scanner(System.in);
try {
    // ...
}
catch(Exception ex) {
    // ...
}
finally {
    sc.close();
}
```

- "throws" clause
  - Written after method declaration to specify list of exception not handled by called method and to be handled by calling method.
  - Writing unhandled checked exceptions in throws clause is compulsory. The unchecked exceptions written in throws clause are ignored by the compiler.

```
void someMethod() throws IOException, SQLException {
    // ...
}
```

- Sub-class overridden method can throw same or subset of exception from super-class method.

```
class SuperClass {
    // ...
    void method() throws IOException, SQLException,
InterruptedException {

    }
}
class SubClass extends SuperClass {
    // ...
    void method() throws IOException, SQLException {

    }
}
```

```
class SuperClass {
    // ...
    void method() throws IOException {

    }
}
class SubClass extends SuperClass {
    // ...
    void method() throws FileNotFoundException, EOFException {

    }
}
```

Exception Handling

Exception chaining

- Sometimes an exception is generated due to another exception.
- For example, database SQLException may be caused due to network problem SocketException.
- To represent this an exception can be chained/nested into another exception.
- If method's throws clause doesn't allow throwing exception of certain type, it can be nested into another (allowed) type and thrown.

User defined exception class

- If pre-defined exception class are not suitable to represent application specific problem, then user-defined exception class should be created.
- User defined exception class may contain fields to store additional information about problem and methods to operate on them.
- Typically exception class's constructor call super class constructor to set fields like message and cause.
- If class is inherited from RuntimeException, it is used as unchecked exception. If it is inherited from Exception, it is used as checked exception.

Generic Programming

- Code is said to be generic if same code can be used for various (practically all) types.
- Best example:
  - Data structure e.g. Stack, Queue, Linked List, ...
  - Algorithms e.g. Sorting, Searching, ...
- Two ways to do Generic Programming in Java
  - using java.lang.Object class -- Non typesafe
  - using Generics -- Typesafe

Generic Programming Using java.lang.Object

```Java
class Box {
    private Object obj;
    public void set(Object obj) {
        this.obj = obj;
    }
    public Object get() {
        return this.obj;
    }
}
```
```Java
Box b1 = new Box();
b1.set("Nilesh");
String obj1 = (String)b1.get();
System.out.println("obj1 : " + obj1);

Box b2 = new Box();
b2.set(new Date());
Date obj2 = (Date)b2.get();
System.out.println("obj2 : " + obj2);

Box b3 = new Box();
b3.set(new Integer(11));
String obj3 = (String)b3.get();  // ClassCastException
System.out.println("obj3 : " + obj3);
```

Generic Programming Using Generics

- Added in Java 5.0.
- Similar to templates in C++.
- We can implement
    - Generic classes
    - Generic methods
    - Generic interfaces

Advantages of Generics

- Stronger type checking at compile time i.e. type-safe coding.
- Explicit type casting is not required.
- Generic data structure and algorithm implementation.

Generic Classes

- Implementing a generic class

```
class Box<TYPE> {
    private TYPE obj;
    public void set(TYPE obj) {
        this.obj = obj;
    }
    public TYPE get() {
        return this.obj;
    }
}
```

```
Box<String> b1 = new Box<String>();
b1.set("Nilesh");
String obj1 = b1.get();
System.out.println("obj1 : " + obj1);

Box<Date> b2 = new Box<Date>();
b2.set(new Date());
Date obj2 = b2.get();
System.out.println("obj2 : " + obj2);

Box<Integer> b3 = new Box<Integer>();
b3.set(new Integer(11));
String obj3 = b3.get();  // Compiler Error
System.out.println("obj3 : " + obj3);
```

- Instantiating generic class

```
Box<String> b1 = new Box<String>(); // okay

Box<String> b2 = new Box<>(); // okay -- type inference -- type of object is
inferred/guessed looking at reference declaration

Box<> b3 = new Box<>(); // compiler error -- type must be given while
declaring reference

Box<Object> b4 = new Box<String>(); // compiler error

Box b5 = new Box(); // okay -- compiler warning "raw types" -- internally
considered as Object type (for T).
    // not recommended -- doesn't do compiler time type-checking

Box<Object> b6 = new Box<Object>(); // okay -- Not usually required/used
```

Generic types naming convention

1. T : Type
2. N : Number
3. E : Element
4. K : Key
5. V : Value
6. S,U,R : Additional type param

Bounded generic types

- Bounded generic param restricts data type that can be used as type argument.
- Decided by the developer of the generic class.

```
// T can be any type so that T is Number or its sub-class.
class Box<T extends Number> {
    private T obj;
    public T get() {
        return this.obj;
    }
    public void set(T obj) {
        this.obj = obj;
    }
}
```

- The Box<> can now be used only for the classes inherited from the Number class.

```
Box<Number> b1 = new Box<>(); // okay
Box<Boolean> b2 = new Box<>(); // error
Box<Character> b3 = new Box<>(); // error
Box<String> b4 = new Box<>(); // error
```

```
Box<Integer> b5 = new Box<>(); // okay
Box<Double> b6 = new Box<>(); // okay
Box<Date> b7 = new Box<>(); // error
Box<Object> b8 = new Box<>(); // error
```

Unbounded generic types

- Unbounded generic type is indicated with wild-card "?".
- Can be given while declaring generic class reference.

```
class Box<T> {
    private T obj;
    public Box(T obj) {
        this.obj = obj;
    }
    public T get() {
        return this.obj;
    }
    public void set(T obj) {
        this.obj = obj;
    }
}
```

```
public static void printBox(Box<?> b) {
    Object obj = b.get();
    System.out.println("Box contains: " + obj);
}
```

```
Box<String> sb = new Box<String>("DAC");
printBox(sb); // okay
Box<Integer> ib = new Box<Integer>(100);
printBox(ib); // okay
Box<Date> db = new Box<Date>(new Date());
printBox(db); // okay
Box<Float> fb = new Box<Float>(200.5f);
printBox(fb); // okay
```

Upper bounded generic types

- Generic param type can be the given class or its sub-class.

```
public static void printBox(Box<? extends Number> b) {
    Object obj = b.get();
```

```
    System.out.println("Box contains: " + obj);
}
```

```
Box<String> sb = new Box<String>("DAC");
printBox(sb); // error
Box<Integer> ib = new Box<Integer>(100);
printBox(ib); // okay
Box<Date> db = new Box<Date>(new Date());
printBox(db); // error
Box<Object> ob = new Box<Object>(new Object());
printBox(ob); // error
```

Lower bounded generic types

- Generic param type can be the given class or its super-class.

```
public static void printBox(Box<? super Number> b) {
    Object obj = b.get();
    System.out.println("Box contains: " + obj);
}
```

```
Box<String> sb = new Box<String>("DAC");
printBox(sb); // error
Box<Integer> ib = new Box<Integer>(100);
printBox(ib); // error
Box<Object> fb = new Box<Object>(new Object());
printBox(fb); // okay
Box<Number> nb = new Box<Number>(null);
printBox(nb); // okay
```

Core Java

Generic Programming

Generic Methods

- Generic methods are used to implement generic algorithms.
- Example:

```
// non type-safe
void printArray(Object[] arr) {
    for(Object ele : arr)
        System.out.println(ele);
    System.out.println("Number of elements printed: " + arr.length);
}
```

```
// type-safe
<T> void printArray(T[] arr) {
    for(T ele : arr)
        System.out.println(ele);
    System.out.println("Number of elements printed: " + arr.length);
}
```

```
String[] arr1 = { "John", "Dagny", "Alex" };
printArray(arr1); // printArray<String> -- String type is inferred

Integer[] arr2 = { 10, 20, 30 };
printArray(arr2); // printArray<Integer> -- Integer type is inferred
```

Generics Limitations

1. Cannot instantiate generic types with primitive Types. Only reference types are allowed.

```
ArrayList<Integer> list = new ArrayList<Integer>(); // okay
ArrayList<int> list = new ArrayList<int>(); // compiler error
```

2. Cannot create instances of Type parameters.

```
Integer i = new Integer(11); // okay
T obj = new T(); // error
```

3. Cannot declare static fields with generic type parameters.

```
class Box<T> {
    private T obj; // okay
    private static T object; // compiler error
    // ...
}
```

4. Cannot Use casts or instanceof with generic Type params.

```
if(obj instanceof T) {  // compiler error
    newobj = (T)obj;    // compiler error
}
```

5. Cannot Create arrays of generic parameterized Types

```
T[] arr = new T[5]; // compiler error
```

6. Cannot create, catch, or throw Objects of Parameterized Types

```
throw new T(); // compiler error

try {
    // ...
} catch(T ex) { // compiler error
    // ...
}
```

7. Cannot overload a method just by changing generic type. Because after erasing/removing the type param, if params of two methods are same, then it is not allowed.

```
public void printBox(Box<Integer> b) {
    // ...
}
public void printBox(Box<String> b) { // compiler error
    // ...
}
```

Type erasure

- The generic type information is erased (not maintained) at runtime (in JVM). `Box<Integer>` and `Box<Double>` both are internally (JVM level) treated as Box objects. The field "T obj" in Box class, is

treated as "Object obj".

- Because of this method overloading with genric type difference is not allowed.

```
void printBox(Box<Integer> b) { ... }
    // void printBox(Box b) { ... } <-- In JVM
void printBox(Box<Double> b) { ... } //compiler error
    // void printBox(Box b) { ... } <-- In JVM
```

Generic Interfaces

- Interface is standard/specification.

```
// Comparable is pre-defined interface -- non-generic till Java 1.4
interface Comparable {
    int compareTo(Object obj);
}
class Person implements Comparable {
    // ...
    public int compareTo(Object obj) {
        Person other = (Person)obj; // down-casting
        // compare "this" with "other" and return difference
    }
}
class Program {
    public static void main(String[] args) {
        Person p1 = new Person("James Bond", 50);
        Person p2 = new Person("Ironman", 45);
        int diff = p1.compareTo(p2);
        if(diff == 0)
            System.out.println("Both are same");
        else if(diff > 0)
            System.out.println("p1 is greater than p2");
        else //if(diff < 0)
            System.out.println("p1 is less than p2");

        diff = p2.compareTo("Superman"); // will fail at runtime with
ClassCastException (in down-casting)
    }
}
```

- Generic interface has type-safe methods (arguments and/or return-type).

```
// Comparable is pre-defined interface -- generic since Java 5.0
interface Comparable<T> {
    int compareTo(T obj);
}
class Person implements Comparable<Person> {
    // ...
```

```
    public int compareTo(Person other) {
        // compare "this" with "other" and return difference


    }
}
class Program {
    public static void main(String[] args) {
        Person p1 = new Person("James Bond", 50);
        Person p2 = new Person("Ironman", 45);
        int diff = p1.compareTo(p2);
        if(diff == 0)
            System.out.println("Both are same");
        else if(diff > 0)
            System.out.println("p1 is greater than p2");
        else //if(diff < 0)
            System.out.println("p1 is less than p2");

        diff = p2.compareTo("Superman"); // compiler error
    }
}
```

Comparable<>

- Standard for comparing the current object to the other object.
- Also referred as "Natural Ordering" for the class.
- Has single abstract method `int compareTo(T other);`
- In java.lang package.
- Used by various methods like Arrays.sort(Object[]), ...

```
// pre-defined interface
interface Comparable<T> {
    int compareTo(T other);
}
```

```
class Employee implements Comparable<Employee> {
    private int empno;
    private String name;
    private int salary;
    // ...
    public int compareTo(Employee other) {
        int diff = this.empno - other.empno;
        return diff;
    }
}
```

```
Employee e1 = new Employee(2, "Sarang", 50000);
Employee e2 = new Employee(1, "Nitin", 40000);
int diff = e1.compareTo(e2);
```

```
Employee[] arr = { ... };
Arrays.sort(arr);
for(Employee e:arr)
    System.out.println(e);
```

Comparator<>

- Standard for comparing two (other) objects.
- Has single abstract method `int compare(T obj1, T obj2);`
- In java.util package.
- Used by various methods like Arrays.sort(T[], comparator), ...

```
// pre-defined interface
interface Comparator<T> {
    int compare(T obj1, T obj2);
}
```

```
class EmployeeSalaryComparator implements Comparator<Employee> {
    @Override
    public int compare(Employee e1, Employee e2) {
        if(e1.getSalary() == e2.getSalary())
            return 0;
        if(e1.getSalary() > e2.getSalary())
            return +1;
        return -1;
    }
}
```

Multi-level sorting

```
class Employee implements Comparable<Employee> {
    private int empno;
    private String name;
    private String designation;
    private int department;
    private int salary;
```

```
    // ...
}
```

```
// Multi-level sorting -- 1st level: department, 2nd level: designation, 3rd
level: salary(int)
class CustomComparator implements Comparator<Employee> {
    public int compare(Employee e1, Employee e2) {
        int diff = e1.getDepartment().compareTo(e2.getDepartment());
        if(diff == 0)
            diff = e1.getDesignation().compareTo(e2.getDesignation());
        if(diff == 0)
            diff = e1.getSalary() - e2.getSalary();
        return diff;
    }
}
```

```
Employee[] arr = { ... };
Arrays.sort(arr, new CustomComparator());
// ...
```

Java Collection Framework

- Collection framework is Library of reusable data structure classes that is used to develop application.
- Main purpose of collection framework is to manage data/objects in RAM efficiently.
- Collection framework was introduced in Java 1.2 and type-safe implementation is provided in 5.0 (using generics).
- java.util package.
- Java collection framework provides
    - Interfaces -- defines standard methods for the collections.
    - Implementations -- classes that implements various data stuctures.
    - Algorithms -- helper methods like searching, sorting, ...

Collection Hierarchy

- Interfaces: Iterable, Collection, List, Queue, Set, Map, Deque, SortedSet, SortedMap, ...
- Implementations: ArrayList, LinkedList, HashSet, HashMap, ...
- Algorithms: sort(), reverse(), max(), min(), ... -> in Collections class static methods

Iterable interface

- To traverse any collection it provides an Iterator.
- Enable use of for-each loop.
- In java.lang package
- Methods
    - Iterator iterator() // SAM

    - default Spliterator spliterator()
    - default void forEach(Consumer<? super T> action)

Collection interface

- Root interface in collection framework interface hierarchy.
- Most of collection classes are inherited from this interface (indirectly).
- Provides most basic/general functionality for any collection
- Abstract methods
    - boolean add(E e)
    - int size()
    - boolean isEmpty()
    - void clear()
    - boolean contains(Object o)
    - boolean remove(Object o)
    - boolean addAll(Collection<? extends E> c)
    - boolean containsAll(Collection<?> c)
    - boolean removeAll(Collection<?> c)
    - boolean retainAll(Collection<?> c)
    - Object[] toArray()
    - Iterator iterator() -- inherited from Iterable
- Default methods
    - default Stream stream()
    - default Stream parallelStream()
    - default boolean removeIf(Predicate<? super E> filter)

Assignment

1. Write a generic static method to find minimum from an array of Number.
2. Use Arrays.sort() to sort array of Students using Comparator. The 1st level sorting should be on city (desc), 2nd level sorting should be on marks (desc), 3rd level sorting should be on name (asc).

```
class Student {
    private int roll;
    private String name;
    private String city;
    private double marks;
    // ...
}
```

Core Java

Java Collection Framework

List interface

- Ordered/sequential collection.
- List can contain duplicate elements.
- List can contain multiple null elements.
- Elements can be accessed sequentially (bi-directional using Iterator) or randomly (index based).
- List enable searching (in the list)
- Implementations: ArrayList, Vector, Stack, LinkedList, etc.
- Abstract methods
    - void add(int index, E element)
    - E get(int index)
    - E set(int index, E element)
    - E remove(int index)
    - boolean addAll(int index, Collection<? extends E> c)
    - int indexOf(Object o)
    - int lastIndexOf(Object o)
    - String toString()
    - ListIterator listIterator()
    - ListIterator listIterator(int index)
    - List subList(int fromIndex, int toIndex)
- To store objects of user-defined types in the list, you must override equals() method for the objects. It is mandetory while searching operations like conatains(), indexOf(), lastIndexOf().

Iterator vs Enumeration

- Enumeration
    - Since Java 1.0
    - Methods
        - boolean hasMoreElements()
        - E nextElement()
    - Example

```
Enumeration<E> e = v.elements();
while(e.hasMoreElements()) {
    E ele = e.nextElement();
    System.out.println(ele);
}
```

    - Enumeration behaves similar to fail-safe iterator.
- Iterator
    - Part of collection framework (1.2)

    - Methods
        - boolean hasNext()
        - E next()
        - void remove()
    - Example

```
Iterator<E> e = v.iterator();
while(e.hasNext()) {
    E ele = e.next();
    System.out.println(ele);
}
```

- ListIterator
    - Part of collection framework (1.2)
    - Inherited from Iterator
    - Bi-directional access
    - Methods
        - boolean hasNext()
        - E next()
        - int nextIndex()
        - boolean hasPrevious()
        - E previous()
        - int previousIndex()
        - void remove()
        - void set(E e)
        - void add(E e)

Traversal

- Using Iterator

```
Iterator<Integer> itr = list.iterator();
while(itr.hasNext()) {
    Integer i = itr.next();
    System.out.println(i);
}
```

- Using for-each loop

```
for(Integer i:list)
    System.out.println(i);
```

    - Gets converted into Iterator traversal

```
for(Iterator<Integer> itr = list.iterator(); itr.hasNext();)  {
    Integer i = itr.next();
    System.out.println(i);
}
```

- Enumeration -- Traversing Vector (Java 1.0)

```
// v is Vector<Integer>
Enumeration<Integer> e = v.elements();
while(e.hasMoreElements()) {
    Integer i = e.nextElement();
    System.out.println(i);
}
```

LinkedList class

- Internally LinkedList is doubly linked list.
- Elements can be traversed using Iterator, ListIterator, or using index.
- Primary use
    - Add/remove elements (anywhere)
    - Less contiguous memory available
- Limitations:
    - Slower random access
- Inherited from List<>, Deque<>.

ArrayList class

- Internally ArrayList is dynamic array (can grow or shrink dynamically).
- When ArrayList capacity is full, it grows by half of its size.
- Elements can be traversed using Iterator, ListIterator, or using index.
- Primary use
    - Random access
    - Add/remove elements (at the end)
- Limitations
    - Slower add/remove in between the collection
    - Uses more contiguous memory
- Inherited from List<>.

Vector class

- Internally Vector is dynamic array (can grow or shrink dynamically).
- Vector is a legacy collection (since Java 1.0) that is modified to fit List interface.
- Vector is synchronized (thread-safe) and hence slower.
- When Vector capacity is full, it doubles its size.
- Elements can be traversed using Enumeration, Iterator, ListIterator, or using index.
- Primary use

    - Random access
    - Add/remove elements (at the end)
- Limitations
    - Slower add/remove in between the collection
    - Uses more contiguous memory
    - Synchronization slow down performance in single threaded environment
- Inherited from List<>.

Fail-fast vs Fail-safe Iterator

- If state of collection is modified (add/remove operation other than iterator methods) while traversing a collection using iterator and iterator methods fails (with ConcurrentModificationException), then iterator is said to be Fail-fast.
    - e.g. Iterators from ArrayList, LinkedList, Vector, ...
- If iterator allows to modify the underlying collection (add/remove operation other than iterator methods) while traversing a collection (NO ConcurrentModificationException), then iterator is said to be Fail-safe.
    - e.g. Iterators from CopyOnWriteArrayList, ...

Synchronized vs Unsynchronized collections

- Synchronized collections are thread-safe and sync checks cause slower execution.
- Legacy collections were synchronized.
    - Vector
    - Stack
    - Hashtable
    - Properties
- Collection classes in collection framework (since 1.2) are non-synchronized (for better performance).
- Collection classes can be converted to synchronized collection using Collections class methods.
    - syncList = Collections.synchronizedList(list)
    - syncSet = Collections.synchronizedSet(set)
    - syncMap = Collections.synchronizedMap(map)

Collections class

- Helper/utility class that provides several static helper methods
- Methods
    - List reverse(List list);
    - List shuffle(List list);
    - void sort(List list, Comparator cmp)
    - E max(Collection list, Comparator cmp);
    - E min(Collection list, Comparator cmp);
    - List synchronizedList(List list);

Collection vs Collections

- Collection interface
    - All methods are public and abstract. They implemented in sub-classes.

- Since all methods are non-static, must be called on object.

```
Collection<Integer> list = new ArrayList<>();
//List<Integer> list = new ArrayList<>();
//ArrayList<Integer> list = new ArrayList<>();
list.remove(new Integer(12));
```

- Collections class
  - Helper class that contains all static methods.
  - We never create object of "Collections" class.

```
Collections.methodName(...);
```

Assignments

1. Store book details in a library in a list -- ArrayList.
   - Book details: isbn(string), price(double), authorName(string), quantity(int)
   - Write a menu driven program to
     1. Add new book in list
     2. Display all books in forward order
     3. Display all books in reverse order
     4. Delete a book at given index.
     5. Sort all books by price in desc order -- list.sort();

Core Java

Collection framework

Queue interface

- Represents utility data structures (like Stack, Queue, ...) data structure.
- Implementations: LinkedList, ArrayDeque, PriorityQueue.
- Can be accessed using iterator, but no random access.
- Methods
    - boolean add(E e) - throw IllegalStateException if full.
    - E remove() - throw NoSuchElementException if empty
    - E element() - throw NoSuchElementException if empty
    - boolean offer(E e) - return false if full.
    - E poll() - returns null if empty
    - E peek() - returns null if empty
- In queue, addition and deletion is done from the different ends (rear and front).

Deque interface

- Represents double ended queue data structure i.e. add/delete can be done from both the ends.
- Two sets of methods
    - Throwing exception on failure: addFirst(), addLast(), removeFirst(), removeLast(), getFirst(), getLast().
    - Returning special value on failure: offerFirst(), offerLast(), pollFirst(), pollLast(), peekFirst(), peekLast().
- Can used as Queue as well as Stack.
- Methods
    - boolean offerFirst(E e)
    - E pollFirst()
    - E peekFirst()
    - boolean offerLast(E e)
    - E pollLast()
    - E peekLast()

ArrayDeque class

- Internally ArrayDeque is dynamically growable array.
- Elements are allocated contiguously in memory.

LinkedList class

- Internally LinkedList is doubly linked list.

PriorityQueue class

- Internally PriorityQueue is a "binary heap" data structure.
- Elements with highest priority is deleted first (NOT FIFO).
- Elements should have natural ordering or need to provide comparator.

Set interface

- Collection of unique elements (NO duplicates allowed).
- Implementations: HashSet, LinkedHashSet, TreeSet.
- Elements can be accessed using an Iterator.
- Abstract methods (same as Collection interface)
    - add() returns false if element is duplicate

HashSet class

- Non-ordered set (elements stored in any order)
- Elements must implement equals() and hashCode()
- Fast execution

LinkedHashSet class

- Ordered set (preserves order of insertion)
- Elements must implement equals() and hashCode()
- Slower than HashSet

SortedSet interface

- Use natural ordering or Comparator to keep elemenrs in sorted order
- Methods
    - E first()
    - E last()
    - SortedSet headSet(E toElement)
    - SortedSet subSet(E fromElement, E toElement)
    - SortedSet tailSet(E fromElement)

NavigableSet interface

- Sorted set with additional methods for navigation
- Methods
    - E higher(E e)
    - E lower(E e)
    - E pollFirst()
    - E pollLast()
    - NavigableSet descendingSet()
    - Iterator descendingIterator()

TreeSet class

- Sorted navigable set (stores elements in sorted order)

- Elements must implement Comparable or provide Comparator
- Slower than HashSet and LinkedHashSet
- It is recommended to have consistent implementation for Comparable (Natural ordering) and equals() method i.e. equality and comparison should done on same fields.
- If need to sort on other fields, use Comparator.

```
class Book implememts Comparable<Book> {
    private String isbn;
    private String name;
    // ...
    public int hashCode() {
        return isbn.hashCode();
    }
    public boolean equals(Object obj) {
        if(!(obj instanceof Book))
            return false;
        Book other=(Book)obj;
        if(this.isbn.equals(other.isbn))
            return true;
        return false;
    }
    public int compareTo(Book other) {
        return this.isbn.compareTo(other.isbn);
    }
}
```

```
// Store in sorted order by name
set = new TreeSet<Book>((b1,b2) -> b1.getName().compareTo(b2.getName()));
```

```
// Store in sorted order by isbn (Natural ordering)
set = new TreeSet<Book>();
```

HashTable Data structure

- Hashtable stores data in key-value pairs so that for the given key, value can be searched in fastest possible time.
- Internally hash-table is a table(array), in which each slot(index) has a bucket(collection). Key-value entries are stored in the buckets depending on hash code of the "key".
- Load factor = Number of entries / Number of buckets.
- Examples
  - Key=pincode, Value=city/area
  - Key=Employee, Value=Manager
  - Key=Department, Value=list of Employees

hashCode() method

- Object class has hashCode() method, that returns a unique number for each object (by converting its address into a number).
- To use any hash-based data structure hashCode() and equals() method must be implemented.
- If two distinct objects yield same hashCode(), it is referred as collision. More collisions reduce performance.
- Most common technique is to multiply field values with prime numbers to get uniform distribution and lesser collsions.
- hashCode() overriding rules
    - hash code should be calculated on the fields that decides equality of the object.
    - hashCode() should return same hash code each time unless object state is modified.
    - If two objects are equal (by equals()), then their hash code must be same.
    - If two objects are not equal (by equals()), then their hash code may be same (but reduce performance).

Map interface

- Collection of key-value entries (Duplicate "keys" not allowed).
- Implementations: HashMap, LinkedHashMap, TreeMap, Hashtable, ...
- The data can be accessed as set of keys, collection of values, and/or set of key-value entries.
- Map.Entry<K,V> is nested interface of Map<K,V>.
    - K getKey()
    - V getValue()
    - V setValue(V value)
- Abstract methods

```
* boolean isEmpty()
* int size()
* V put(K key, V value)
* V get(Object key)
* Set<K> keySet()
* Collection<V> values()
* Set<Map.Entry<K,V>> entrySet()
* boolean containsValue(Object value)
* boolean containsKey(Object key)
* V remove(Object key)
* void clear()
* void putAll(Map<? extends K,? extends V> map)
```

- Maps not considered as true collection, because it is not inherited from Collection interface.

HashMap class

- Non-ordered map (entries stored in any order -- as per hash code of key)
- Keys must implement equals() and hashCode()
- Fast execution
- Mostly used Map implementation

LinkedHashMap class

- Ordered map (preserves order of insertion)
- Keys must implement equals() and hashCode()
- Slower than HashSet
- Since Java 1.4

TreeMap class

- Sorted navigable map (stores entries in sorted order of key)
- Keys must implement Comparable or provide Comparator
- Slower than HashMap and LinkedHashMap
- Internally based on Red-Black tree.
- Doesn't allow null key (allows null value though).

Hashtable class

- Similar to HashMap class.
- Legacy collection class (since Java 1.0), modified for collection framework (Map interface).
- Synchronized collection -- Thread safe but slower performance
- Inherited from java.util.Dictionary abstract class (it is Obsolete).

Annoymous Inner class

- Creates a new class inherited from the given class/interface and its object is created.
- If in static context, behaves like static member class. If in non-static context, behaves like non-static member class.
- Along with Outer class members, it can also access (effectively) final local variables of the enclosing method.

```
// (named) local class
class EmpnoComparator implements Comparator<Employee> {
    public int compare(Employee e1, Employee e2) {
        return e1.getEmpno() - e2.getEmpno();
    }
}
Arrays.sort(arr, new EmpnoComparator());    // anonymous obj of local class
```

```
// Anonymous inner class
Comparator<Employee> cmp = new Comparator<Employee>() {
    public int compare(Employee e1, Employee e2) {
        return e1.getEmpno() - e2.getEmpno();
    }
};
Arrays.sort(arr, cmp);
```

```
// Anonymous object of Anonymous inner class.
Arrays.sort(arr, new Comparator<Employee>() {
    public int compare(Employee e1, Employee e2) {
        return e1.getEmpno() - e2.getEmpno();
    }
});
```

Java 8 Interfaces

- Before Java 8
  - Interfaces are used to design specification/standards. It contains only declarations – public abstract.

```
interface Geometry {
    /*public static final*/ double PI = 3.14;
    /*public abstract*/ int calcRectArea(int length, int breadth);
    /*public abstract*/ int calcRectPeri(int length, int breadth);
}
```

  - As interfaces doesn't contain method implementations, multiple interface inheritance is supported (no ambiguity error).
  - Interfaces are immutable. One should not modify interface once published.
- Java 8 added many new features in interfaces in order to support functional programming in Java. Many of these features also contradicts earlier Java/OOP concepts.

Default methods

- Java 8 allows default methods in interfaces. If method is not overridden, its default implementation in interface is considered.
- This allows adding new functionalities into existing interfaces without breaking old implementations e.g. Collection, Comparator, ...

```
interface Emp {
    double getSal();
    default double calcIncentives() {
        return 0.0;
    }
}
class Manager implements Emp {
    // ...
    // calcIncentives() is overridden
    double calcIncentives() {
        return getSal() * 0.2;
    }
}
```

```
class Clerk implements Emp {
    // ...
    // calcIncentives() is not overridden -- so method of interface is
considered
}
```

```
new Manager().calcIncentives(); // return sal * 0.2
new Clerk().calcIncentives(); // return 0.0
```

- However default methods will lead to ambiguity errors as well, if same default method is available from multiple interfaces. Error: Duplicate method while declaring class.
- Superclass same method get higher priority. But super-interfaces same method will lead to error.
    - Super-class wins! Super-interfaces clash!!

```
interface Displayable {
    default void show() {
        System.out.println("Displayable.show() called");
    }
}
interface Printable {
    default void show() {
        System.out.println("Printable.show() called");
    }
}
class FirstClass implements Displayable, Printable { // compiler error:
duplicate method
    // ...
}
class Main {
    public static void main(String[] args) {
        FirstClass obj = new FirstClass();
        obj.show();
    }
}
```

```
interface Displayable {
    default void show() {
        System.out.println("Displayable.show() called");
    }
}
interface Printable {
    default void show() {
        System.out.println("Printable.show() called");
    }
}
class Superclass {
    public void show() {
```

```
        System.out.println("Superclass.show() called");
    }
}
class SecondClass extends Superclass implements Displayable, Printable {
    // ...
}
class Main {
    public static void main(String[] args) {
        SecondClass obj = new SecondClass();
        obj.show(); // Superclass.show() called
    }
}
```

- A class can invoke methods of super interfaces using InterfaceName.super.

```
interface Displayable {
    default void show() {
        System.out.println("Displayable.show() called");
    }
}
interface Printable {
    default void show() {
        System.out.println("Printable.show() called");
    }
}
class FourthClass implements Displayable, Printable {
    @Override
    public  void show() {
        System.out.println("FourthClass.show() called");
        Displayable.super.show();
        Printable.super.show();
    }
}
class Main {
    public static void main(String[] args) {
        FourthClass obj = new FourthClass();
        obj.show(); // calls FourthClass method
    }
}
```

Static methods

- Before Java 8, interfaces allowed public static final fields.
- Java 8 also allows the static methods in interfaces.
- They act as helper methods and thus eliminates need of helper classes like Collections, ...

```
interface Emp {
    double getSal();
    public static double calcTotalSalary(Emp[] a) {
```

```
        double total = 0.0;
        for(int i=0; i<a.length; i++)
            total += a[i].getSal();
        return total;
    }
}
```

Functional Interface

- If interface contains exactly one abstract method (SAM), it is said to be functional interface.
- It may contain additional default & static methods. E.g. Comparator, Runnable, ...
- @FunctionalInterface annotation does compile time check, whether interface contains single abstract method. If not, raise compile time error.

```
@FunctionalInterface // okay
interface Foo {
    void foo(); // SAM
}
```

```
@FunctionalInterface // okay
interface FooBar1 {
    void foo(); // SAM
    default void bar() {
        /*... */
    }
}
```

```
@FunctionalInterface // NO -- error
interface FooBar2 {
    void foo(); // AM
    void bar(); // AM
}
```

```
@FunctionalInterface // NO -- error
interface FooBar3 {
    default void foo() {
        /*... */
    }
    default void bar() {
        /*... */
    }
}
```

```
@FunctionalInterface    // okay
interface FooBar4 {
    void foo(); // SAM
    public static void bar() {
        /*... */
    }
}
```

- Functional interfaces forms foundation for Java lambda expressions and method references.

Built-in functional interfaces

- New set of functional interfaces given in java.util.function package.
  - `Predicate<T>`: test: T -> boolean
  - `Function<T,R>`: apply: T -> R
  - `BiFunction<T,U,R>`: apply: (T,U) -> R
  - `UnaryOperator<T>`: apply: T -> T
  - `BinaryOperator<T>`: apply: (T,T) -> T
  - `Consumer<T>`: accept: T -> void
  - `Supplier<T>`: get: () -> T
- For efficiency primitive type functional interfaces are also supported e.g. IntPredicate, IntConsumer, IntSupplier, IntToDoubleFunction, ToIntFunction, ToIntBiFunction, IntUnaryOperator, IntBinaryOperator.

Lambda expressions

- Traditionally Java uses anonymous inner classes to compact the code. For each inner class separate .class file is created.
- However code is complex to read and un-efficient to execute.
- Lambda expression is short-hand way of implementing functional interface.
- Its argument types may or may not be given. The types will be inferred.
- Lambda expression can be single liner (expression not statement) or multi-liner block { ... }.

```
// Anonymous inner class
Arrays.sort(arr, new Comparator<Emp>() {
    public int compare(Emp e1, Emp e2) {
        int diff = e1.getEmpno() - e2.getEmpno();
        return diff;
    }
});
```

```
// Lambda expression -- multi-liner
Arrays.sort(arr, (Emp e1, Emp e2) -> {
    int diff = e1.getEmpno() - e2.getEmpno();
    return diff;
});
```

```
// Lambda expression -- multi-liner -- Argument types inferred
Arrays.sort(arr, (e1, e2) -> {
    int diff = e1.getEmpno() - e2.getEmpno();
    return diff;
});
```

```
// Lambda expression -- single-liner -- with block { ... }
Arrays.sort(arr, (e1, e2) -> {
    return e1.getEmpno() - e2.getEmpno();
});
```

```
// Lambda expression -- single-liner
Arrays.sort(arr, (e1,e2) -> e1.getEmpno() - e2.getEmpno());
```

- Practically lambda expressions are used to pass as argument to various functions.
- Lambda expression enable developers to write concise code (single liners recommended).

Core Java

Built-in functional interfaces

- New set of functional interfaces given in java.util.function package.
  - `Predicate<T>`: test: T -> boolean
  - `Function<T,R>`: apply: T -> R
  - `BiFunction<T,U,R>`: apply: (T,U) -> R
  - `UnaryOperator<T>`: apply: T -> T
  - `BinaryOperator<T>`: apply: (T,T) -> T
  - `Consumer<T>`: accept: T -> void
  - `Supplier<T>`: get: () -> T
- For efficiency primitive type functional interfaces are also supported e.g. IntPredicate, IntConsumer, IntSupplier, IntToDoubleFunction, ToIntFunction, ToIntBiFunction, IntUnaryOperator, IntBinaryOperator.

Lambda expressions

- Traditionally Java uses anonymous inner classes to compact the code. For each inner class separate .class file is created.
- However code is complex to read and un-efficient to execute.
- Lambda expression is short-hand way of implementing functional interface.
- Its argument types may or may not be given. The types will be inferred.
- Lambda expression can be single liner (expression not statement) or multi-liner block { ... }.

```
// Anonymous inner class
Arrays.sort(arr, new Comparator<Emp>() {
    public int compare(Emp e1, Emp e2) {
        int diff = e1.getEmpno() - e2.getEmpno();
        return diff;
    }
});
```

```
// Lambda expression -- multi-liner
Arrays.sort(arr, (Emp e1, Emp e2) -> {
    int diff = e1.getEmpno() - e2.getEmpno();
    return diff;
});
```

```
// Lambda expression -- multi-liner -- Argument types inferred
Arrays.sort(arr, (e1, e2) -> {
    int diff = e1.getEmpno() - e2.getEmpno();
    return diff;
});
```

```
// Lambda expression -- single-liner -- with block { ... }
Arrays.sort(arr, (e1, e2) -> {
    return e1.getEmpno() - e2.getEmpno();
});
```

```
// Lambda expression -- single-liner
Arrays.sort(arr, (e1,e2) -> e1.getEmpno() - e2.getEmpno());
```

- Practically lambda expressions are used to pass as argument to various functions.
- Lambda expression enable developers to write concise code (single liners recommended).

Non-capturing lambda expression

- If lambda expression result entirely depends on the arguments passed to it, then it is non-capturing (self-contained).

```
BinaryOperator<Integer> op1 = (a,b) -> a + b;
testMethod(op);
```

```
static void testMethod(BinaryOperator<Integer> op) {
    int x=12, y=5, res;
    res = op.apply(x, y); // res = x + y;
    System.out.println("Result: " + res)
}
```

- In functional programming, such functions/lambdas are referred as pure functions.

Capturing lambda expression

- If lambda expression result also depends on additional variables in the context of the lambda expression passed to it, then it is capturing.

```
int c = 2; // must be effectively final
BinaryOperator<Integer> op = (a,b) -> a + b + c;
testMethod(op);
```

```
static void testMethod(BinaryOperator<Integer> op) {
    int x=12, y=5, res;
    res = op.apply(x, y); // res = x + y + c;
```

```
        System.out.println("Result: " + res);
    }
```

- Here variable c is bound (captured) into lambda expression. So it can be accessed even out of scope (effectively). Internally it is associated with the method/expression.
- In some functional languages, this is known as Closures.

Java 8 Streams

- Java 8 Stream is NOT IO streams.
- java.util.stream package.
- Streams follow functional programming model in Java 8.
- The functional programming is based on functional interface (SAM).
- Number of predefined functional interfaces added in Java 8. e.g. Consumer, Supplier, Function, Predicate, ...
- Lambda expression is short-hand way of implementing SAM -- arg types & return type are inferred.
- Java streams represents pipeline of operations through which data is processed.
- Stream operations are of two types
    - Intermediate operations: Yields another stream.
        - filter()
        - map(), flatMap()
        - limit(), skip()
        - sorted(), distinct()
    - Terminal operations: Yields some result.
        - reduce()
        - forEach()
        - collect(), toArray()
        - count(), max(), min()
    - Stream operations are higher order functions (take functional interfaces as arg).

Java stream characteristics

- No storage: Stream is an abstraction. Stream doesn't store the data elements. They are stored in source collection or produced at runtime.
- Immutable: Any operation doesn't change the stream itself. The operations produce new stream of results.
- Lazy evaluation: Stream is evaluated only if they have terminal operation. If terminal operation is not given, stream is not processed.
- Not reusable: Streams processed once (terminal operation) cannot be processed again.

Stream creation

- Collection interface: stream() or parallelStream()
- Arrays class: Arrays.stream()
- Stream interface: static of() method
- Stream interface: static generate() method
- Stream interface: static iterate() method

- Stream interface: static empty() method
- nio Files class: `static Stream<String> lines(filePath)` method

Stream creation

- Collection interface: stream() or parallelStream()

```
List<String> list = new ArrayList<>();
// ...
Stream<String> strm = list.stream();
```

- Arrays class: Arrays.stream()
- Stream interface: static of() method

```
Stream<Integer> strm = Stream.of(arr);
```

- Stream interface: static generate() method
  - generate() internally calls given Supplier in an infinite loop to produce infinite stream of elements.

```
Stream<Double> strm = Stream.generate(() -> Math.random()).limit(25);
```

```
Random r = new Random();
Stream<Integer> strm = Stream.generate(() -> r.nextInt(1000)).limit(10);
```

- Stream interface: static iterate() method
  - iterate() start the stream from given (arg1) "seed" and calls the given UnaryOperator in infinite loop to produce infinite stream of elements.

```
Stream<Integer> strm = Stream.iterate(1, i -> i + 1).limit(10);
```

- Stream interface: static empty() method
- nio Files class: static Stream lines(filePath) method

Stream operations

- Source of elements

```
String[] names = {"Smita", "Rahul", "Rachana", "Amit", "Shraddha", "Nilesh",
"Rohan", "Pradnya", "Rohan", "Pooja", "Lalita"};
```

- Create Stream and display all names

```
Stream.of(names)
    .forEach(s -> System.out.println(s));
```

- filter() -- Get all names ending with "a"
  - `Predicate<T>`: (T) -> boolean

```
Stream.of(names)
    .filter(s -> s.endsWith("a"))
    .forEach(s -> System.out.println(s));
```

- map() -- Convert all names into upper case
  - `Function<T,R>`: (T) -> R

```
Stream.of(names)
    .map(s -> s.toUpperCase())
    .forEach(s -> System.out.println(s));
```

- sorted() -- sort all names in ascending order
  - String class natural ordering is ascending order.
  - sorted() is a stateful operation (i.e. needs all element to sort).

```
Stream.of(names)
    .sorted()
    .forEach(s -> System.out.println(s));
```

- sorted() -- sort all names in descending order
  - `Comparator<T>`: (T,T) -> int

```
Stream.of(names)
    .sorted((x,y) -> y.compareTo(x))
    .forEach(s -> System.out.println(s));
```

- skip() & limit() -- leave first 2 names and print next 4 names

```
Stream.of(names)
    .skip(2)
```

```
    .limit(4)
    .forEach(s -> System.out.println(s));
```

- distinct() -- remove duplicate names
  - duplicates are removed according to equals().

```
Stream.of(names)
    .distinct()
    .forEach(s -> System.out.println(s));
```

- count() -- count number of names
  - terminal operation: returns long.

```
long cnt = Stream.of(names)
    .count();
System.out.println(cnt);
```

- collect() -- collects all stream elements into an collection (list, set, or map)

```
List<String> list = Stream.of(names)
        .collect(Collectors.toList());
// Collectors.toList() returns a Collector that can collect all stream
elements into a list
```

```
Set<String> set = Stream.of(names)
    .collect(Collectors.toSet());
// Collectors.toSet() returns a Collector that can collect all stream
elements into a set
```

- reduce() -- addition of 1 to 5 numbers

```
int result = Stream
    .iterate(1, i -> i+1)
    .limit(5)
    .reduce(0, (x,y) -> x + y);
```

- max() -- find the max string
  - terminal operation
  - See examples.

Collect Stream result

- Collecting stream result is terminal operation.
- Object[] toArrray()
- R collect(Collector)
    - Collectors.toList(), Collectors.toSet(), Collectors.toCollection(), Collectors.joining()
    - Collectors.toMap(key, value)

Stream of primitive types

- Efficient in terms of storage and processing. No auto-boxing and unboxing is done.
- IntStream class
    - IntStream.of() or IntStream.range() or IntStream.rangeClosed() or Random.ints()
    - sum(), min(), max(), average(), summaryStatistics(),

Method references

- If lambda expression involves single method call, it can be shortened by using method reference.
- Method references are converted into instances of functional interfaces.
- Method reference can be used for class static method, class non-static method, object non-static method or constructor.

Examples

- Class static method: Integer::sum [(a,b) -> Integer.sum(a,b)]
    - Both lambda param passed to static function explicitly
- Class non-static method: String::compareTo [(a,b) -> a.compareTo(b)]
    - First lambda param become implicit param (this) of the function and second is passed explicitly (as arguments).
- Object non-static method: System.out::println [x -> System.out.println(x)]
    - Lambda param is passed to function explicitly.
- Constructor: Date::new [() -> new Date()]
    - Lambda param is passed to constructor explicitly.

enum

- "enum" keyword is added in Java 5.0.
- Used to make constants to make code more readable.
- Typical switch case

```
int choice;
// ...
switch(choice) {
    case 1: // addition
        c = a + b;
        break;
    case 2: // subtraction
        c = a - b;
        break;
    // ...
}
```

- The switch constants can be made more readable using Java enums.

```
enum ArithmeticOperations {
    ADDITION, SUBTRACTION, MULIPLICATION, DIVISION;
}

ArithmeticOperations choice = ArithmeticOperations.ADDITION;
// ...
switch(choice) {
    case ADDITION:
        c = a + b;
        break;
    case SUBTRACTION:
        c = a - b;
        break;
    // ...
}
```

- In java, enums cannot be declared locally (within a method).
- The declared enum is converted into enum class.

```
// user-defined enum
enum ArithmeticOperations {
    ADDITION, SUBTRACTION, MULIPLICATION, DIVISION;
}
```

```
// generated enum code
final class ArithmeticOperations extends Enum {
    public static ArithmeticOperations[] values() {
        return (ArithmeticOperations[])$VALUES.clone();
    }
    public static ArithmeticOperations valueOf(String s) {
        return (ArithmeticOperations)Enum.valueOf(ArithmeticOperations, s);
    }
    private ArithmeticOperations(String name, int ordinal) {
        super(name, ordinal); // invoke sole constructor Enum(String,int);
    }
    public static final ArithmeticOperations ADDITION;
    public static final ArithmeticOperations SUBTRACTION;
    public static final ArithmeticOperations MULIPLICATION;
    public static final ArithmeticOperations DIVISION;
    private static final ArithmeticOperations $VALUES[];
    static {
        ADDITION = new ArithmeticOperations("ADDITION", 0);
        SUBTRACTION = new ArithmeticOperations("SUBTRACTION", 1);
        MULIPLICATION = new ArithmeticOperations("MULIPLICATION", 2);
```

```
        DIVISION = new ArithmeticOperations("DIVISION", 3);
        $VALUES = (new ArithmeticOperations[] {
            ADDITION, SUBTRACTION, MULIPLICATION, DIVISION
        });
    }
}
```

- The enum type declared is implicitly inherited from java.lang.Enum class. So it cannot be extended from another class, but enum may implement interfaces.
- The enum constants declared in enum are public static final fields of generated class. Enum objects cannot be created explicitly (as generated constructor is private).
- The generated class will have a values() method that returns array of all constants and valueOf() method to convert String to enum constant.
- The enums constants can be used in switch-case and can also be compared using == operator.
- The java.lang.Enum class has following members:

```
public abstract class Enum<E> implements java.lang.Comparable<E>,
java.io.Serializable {
    private final String name;
    private final int ordinal;

    protected Enum(String,int); // sole constructor - can be called from
user-defined enum class only
    public final String name(); // name of enum const
    public final int ordinal(); // position of enum const (0-based)

    public String toString(); // returns name of const
    public final int compareTo(E);// compares with another enum of same type
on basis of ordinal number
    public static <T> T valueOf(Class<T>, String);
    // ...
}
```

- The enum may have fields and methods.

```
enum Element {
    H(1, "Hydrogen"),
    HE(2, "Helium"),
    LI(3, "Lithium");

    public final int num;
    public final String label;

    private Element(int num, String label) {
        this.num = num;
        this.label = label;
    }
}
```

Reflection

- .class = Byte-code + Meta-data + Constant pool + ...
- When class is loaded into JVM all the metadata is stored in the object of java.lang.Class (heap area).
- This metadata includes class name, super class, super interfaces, fields (field name, field type, access modifier, flags), methods (method name, method return type, access modifier, flags, method arguments, ...), constructors (access modifier, flags, ctor arguments, ...), annotations (on class, fields, methods, ...).

Reflection applications

- Inspect the metadata (like javap)
- Build IDE/tools (Intellisense)
- Dynamically creating objects and invoking methods
- Access the private members of the class

Get the java.lang.Class object

- way 1: When you have class-name as a String (taken from user or in properties file)

```
Class<?> c = Class.forName(className);
```

- way 2: When the class is in project/classpath.

```
Class<?> c = ClassName.class;
```

- way 3: When you have object of the class.

```
Class<?> c = obj.getClass();
```

Access metadata in java.lang.Class

- Name of the class

```
String name = c.getName();
```

- Super class of the class

```
Class<?> supcls = c.getSuperclass();
```

- Super interfaces of the class

```
Class<?> supintf[] = c.getInterfaces();
```

- Fields of the class

```
Field[] fields = c.getFields(); // all fields accessible (of class & its
super class)
```

```
Field[] fields = c.getDeclaredFields(); // all fields in the class
```

- Methods of the class

```
Method[] methods = c.getMethods(); // all methods accessible (of class & its
super class)
```

```
Method[] methods = c.getDeclaredMethods(); // all methods in the class
```

- Constructors of the class

```
Constructor[] ctors = c.getConstructors(); // all ctors accessible (of class
& its super class)
```

```
Constructor[] ctors = c.getDeclaredConstructor(); // all ctors in the class
```

Reflection Tutorial

- https://youtu.be/lAoNJ_7LD44
- https://youtu.be/UVWdtk5ibK8

Annotations

- Added in Java 5.0.
- Annotation is a way to associate metadata with the class and/or its members.
- Annotation applications
    - Information to the compiler
    - Compile-time/Deploy-time processing

    - Runtime processing
- Annotation Types
    - Marker Annotation: Annotation is not having any attributes.
        - @Override, @Deprecated, @FunctionalInterface ...
    - Single value Annotation: Annotation is having single attribute -- usually it is "value".
        - @SuppressWarnings("deprecation"), ...
    - Multi value Annotation: Annotation is having multiple attribute
        - @RequestMapping(method = "GET", value = "/books"), ...

Pre-defined Annotations

- @Override
    - Ask compiler to check if corresponding method (with same signature) is present in super class.
    - If not present, raise compiler error.
- @FunctionalInterface
    - Ask compiler to check if interface contains single abstract method.
    - If zero or multiple abstract methods, raise compiler error.
- @Deprecated
    - Inform compiler to give a warning when the deprecated type/member is used.
- @SuppressWarnings
    - Inform compiler not to give certain warnings: e.g. deprecation, rawtypes, unchecked, serial, unused
    - @SuppressWarnings("deprecation")
    - @SuppressWarnings({"rawtypes", "unchecked"})
    - @SuppressWarnings("serial")
    - @SuppressWarnings("unused")

Meta-Annotations

- Annotations that apply to other annotations are called meta-annotations.
- Meta-annotation types defined in java.lang.annotation package.

@Retention

- RetentionPolicy.SOURCE
    - Annotation is available only in source code and discarded by the compiler (like comments).
    - Not added into .class file.
    - Used to give information to the compiler.
    - e.g. @Override, ...
- RetentionPolicy.CLASS
    - Annotation is compiled and added into .class file.
    - Discared while class loading and not loaded into JVM memory.
    - Used for utilities that process .class files.
    - e.g. Obfuscation utilities can be informed not to change the name of certain class/member using @SerializedName, ...
- RetentionPolicy.RUNTIME
    - Annotation is compiled and added into .class file. Also loaded into JVM at runtime and available for reflective access.

- Used by many Java frameworks.
- e.g. @RequestMapping, @Id, @Table, @Controller, ...

@Target

- Where this annotation can be used.
- ANNOTATION_TYPE, CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE, TYPE_PARAMETER, TYPE_USE
- If annotation is used on the other places than mentioned in @Target, then compiler raise error.

@Documented

- This annotation should be documented by javadoc or similar utilities.

@Repeatable

- The annotation can be repeated multiple times on the same class/target.

@Inherited

- The annotation gets inherited to the sub-class and accessible using c.getAnnotation() method.

Custom Annotation

- Annotation to associate developer information with the class and its members.

```
@Inherited
@Retention(RetentionPolicy.RUNTIME) // the def attribute is considered as
"value" = @Retention(value = RetentionPolicy.RUNTIME )
@Target({TYPE, CONSTRUCTOR, FIELD, METHOD}) // { } represents array
@interface Developer {
    String firstName();
    String lastName();
    String company() default "Sunbeam";
    String value() default "Software Engg";
}

@Repeatable
@Retention(RetentionPolicy.RUNTIME)
@Target({TYPE})
@interface CodeType {
    String[] value();
}
```

```
//@Developer(firstName="Nilesh", lastName="Ghule", value="Technical
Director") // compiler error -- @Developer is not @Repeatable
@CodeType({"businessLogic", "algorithm"})
@Developer(firstName="Nilesh", lastName="Ghule", value="Technical Director")
```

```
class MyClass {
    // ...
    @Developer(firstName="Shubham", lastName="Patil", company="Sunbeam Karad
")
    private int myField;
    @Developer(firstName="Rahul", lastName="Sansuddi")
    public MyClass() {

    }
    @Developer(firstName="Shubham", lastName="Borle", company="Sunbeam Karad
")
    public void myMethod() {
        @Developer(firstName="James", lastName="Bond") // compiler error
        int localVar = 1;
    }
}
```

```
// @Developer is inherited
@CodeType("frontEnd")
@CodeType("businessLogic") // allowed because @CodeType is @Repeatable
class YourClass extends MyClass {
    // ...
}
```

Java IO framework

- Input/Output functionality in Java is provided under package java.io and java.nio package.
- IO framework is used for File IO, Network IO, Memory IO, and more.
- File is a collection of data and information on a storage device.
- File = Data + Metadata
- Two types of APIs are available file handling
    - FileSystem API -- Accessing/Manipulating Metadata
    - File IO API -- Accessing/Manipulating Contents/Data

java.io.File class

- A path (of file or directory) in file system is represented by "File" object.
- Used to access/manipulate metadata of the file/directory.
- Provides FileSystem APIs
    - String[] list() -- return contents of the directory
    - File[] listFiles() -- return contents of the directory
    - boolean exists() -- check if given path exists
    - boolean mkdir() -- create directory
    - boolean mkdirs() -- create directories (child + parents)
    - boolean createNewFile() -- create empty file
    - boolean delete() -- delete file/directory
    - boolean renameTo(File dest) -- rename file/directory

- String getAbsolutePath() -- returns full path (drive:/folder/folder/...)
- String getPath() -- return path
- File getParentFile() -- returns parent directory of the file
- String getParent() -- returns parent directory path of the file
- String getName() -- return name of the file/directory
- static File[] listRoots() -- returns all drives in the systems.
- long getTotalSpace() -- returns total space of current drive
- long getFreeSpace() -- returns free space of current drive
- long getUsableSpace() -- returns usable space of current drive
- boolean isDirectory() -- return true if it is a directory
- boolean isFile() -- return true if it is a file
- boolean isHidden() -- return true if the file is hidden
- boolean canExecute()
- boolean canRead()
- boolean canWrite()
- boolean setExecutable(boolean executable) -- make the file executable
- boolean setReadable(boolean readable) -- make the file readable
- boolean setWritable(boolean writable) -- make the file writable
- long length() -- return size of the file in bytes
- long lastModified() -- last modified time
- boolean setLastModified(long time) -- change last modified time

Core Java

Java IO framework

- Input/Output functionality in Java is provided under package java.io and java.nio package.
- IO framework is used for File IO, Network IO, Memory IO, and more.
- File is a collection of data and information on a storage device.
- File = Data + Metadata
- Two types of APIs are available file handling
    - FileSystem API -- Accessing/Manipulating Metadata
    - File IO API -- Accessing/Manipulating Contents/Data

java.io.File class

- A path (of file or directory) in file system is represented by "File" object.
- Used to access/manipulate metadata of the file/directory.
- Provides FileSystem APIs
    - String[] list() -- return contents of the directory
    - File[] listFiles() -- return contents of the directory
    - boolean exists() -- check if given path exists
    - boolean mkdir() -- create directory
    - boolean mkdirs() -- create directories (child + parents)
    - boolean createNewFile() -- create empty file
    - boolean delete() -- delete file/directory
    - boolean renameTo(File dest) -- rename file/directory
    - String getAbsolutePath() -- returns full path (drive:/folder/folder/...)
    - String getPath() -- return path
    - File getParentFile() -- returns parent directory of the file
    - String getParent() -- returns parent directory path of the file
    - String getName() -- return name of the file/directory
    - static File[] listRoots() -- returns all drives in the systems.
    - long getTotalSpace() -- returns total space of current drive
    - long getFreeSpace() -- returns free space of current drive
    - long getUsableSpace() -- returns usable space of current drive
    - boolean isDirectory() -- return true if it is a directory
    - boolean isFile() -- return true if it is a file
    - boolean isHidden() -- return true if the file is hidden
    - boolean canExecute()
    - boolean canRead()
    - boolean canWrite()
    - boolean setExecutable(boolean executable) -- make the file executable
    - boolean setReadable(boolean readable) -- make the file readable
    - boolean setWritable(boolean writable) -- make the file writable
    - long length() -- return size of the file in bytes
    - long lastModified() -- last modified time

    - boolean setLastModified(long time) -- change last modified time

Java IO

- Java File IO is done with Java IO streams.
- Stream is abstraction of data source/sink.
    - Data source -- InputStream or Reader
    - Data sink -- OutputStream or Writer
- Java supports two types of IO streams.
    - Byte streams (binary files) -- byte by byte read/write
    - Character streams (text files) -- char by char read/write
- All these streams are AutoCloseable (so can be used with try-with-resource construct)

IO Framework

Chaining IO Streams

- Each IO stream object performs a specific task.
    - FileOutputStream -- Write the given bytes into the file (on disk).
    - BufferedOutputStream -- Hold multiple elements in a temporary buffer before flushing it to underlying stream/device. Improves performance.
    - DataOutputStream -- Convert primitive types into sequence of bytes. Inherited from DataOutput interface.
    - ObjectOutputStream -- Convert object into sequence of bytes. Inherited from ObjectOutput interface.
    - PrintStream -- Convert given input into formatted output.
    - Note that input streams does the counterpart of OutputStream class hierarchy.
- Streams can be chained to fulfil application requirements.

Primitive types IO

- DataInputStream & DataOutputStream -- convert primitive types from/to bytes
    - primitive type --> DataOutputStream --> bytes --> FileOutputStream --> file.
        - DataOutput interface provides methods for conversion - writeInt(), writeUTF(), writeDouble(), ...
    - primitive type <-- DataInputStream <-- bytes <-- FileInputStream <-- file.
        - DataInput interface provides methods for conversion - readInt(), readUTF(), readDouble(), ...

DataOutput/DataInput interface

- interface DataOutput
    - writeUTF(String s)
    - writeInt(int i)
    - writeDouble(double d)
    - writeShort(short s)
    - ...
- interface DataInput

    - String readUTF()
    - int readInt()
    - double readDouble()
    - short readShort()
    - ...

Serialization

- ObjectInputStream & ObjectOutputStream -- convert java object from/to bytes
    - Java object --> ObjectOutputStream --> bytes --> FileOutputStream --> file.
        - ObjectOutput interface provides method for conversion - writeObject().
    - Java object <-- ObjectInputStream <-- bytes <-- FileInputStream <-- file.
        - ObjectInput interface provides methods for conversion - readObject().
- Converting state of object into a sequence of bytes is referred as Serialization. The sequence of bytes includes object data as well as metadata.
- Serialized data can be further saved into a file (using FileOutputStream) or sent over the network (Marshalling process).
- Converting (serialized) bytes back to the Java object is referred as Deserialization.
- These bytes may be received from the file (using FileInputStream) or from the network (Unmarshalling process).

ObjectOutput/ObjectInput interface

- interface ObjectOutput extends DataOutput
    - writeObject(obj)
- interface ObjectInput extends DataInput
    - obj = readObject()

Serializable interface

- Object can be serialized only if class is inherited from Serializable interface; otherwise writeObject() throws NotSerializableException.
- Serializable is a marker interface.

transient fields

- writeObject() serialize all non-static fields of the class. If fields are objects, then they are also serialized.
- If any field is intended not to serialize, then it should be marked as "transient".
- The transient and static fields (except serialVersionUID) are not serialized.

serialVersionUID field

- Each serializable class is associated with a version number, called a serialVersionUID.
- It is recommended that programmer should define it as a static final long field (with any access specifier). Any change in class fields expected to modify this serialVersionUID.

```
private static final long serialVersionUID = 1001L;
```

- During deserialization, this number is verified by the runtime to check if right version of the class is loaded in the JVM. If this number mismatched, then InvalidClassException will be thrown.
- If a serializable class does not explicitly declare a serialVersionUID, then the runtime will calculate a default serialVersionUID value for that class (based on various aspects of the class described in the Java(TM) Object Serialization specification).

Buffered streams

- Each write() operation on FileOutputStream will cause data to be written on disk (by OS). Accessing disk frequently will reduce overall application performance. Similar performance problems may occur during network data transfer.
- BufferedOutputStream classes hold data into a in-memory buffer before transferring it to the underlying stream. This will result in better performance.
    - Java object --> ObjectOutputStream --> BufferedOutputStream --> FileOutputStream --> file on disk.
- Data is sent to underlying stream when buffer is full or flush() called explicitly.
- BufferedInputStream provides a buffering while reading the file.
- The buffer size can be provided while creating the respective objects.

Chaining IO Streams

- Each IO stream object performs a specific task.
    - FileOutputStream -- Write the given bytes into the file (on disk).
    - BufferedOutputStream -- Hold multiple elements in a temporary buffer before flushing it to underlying stream/device. Improves performance.
    - DataOutputStream -- Convert primitive types into sequence of bytes. Inherited from DataOutput interface.
    - ObjectOutputStream -- Convert object into sequence of bytes. Inherited from ObjectOutput interface.
    - PrintStream -- Convert given input into formatted output.
    - Note that input streams does the counterpart of OutputStream class hierarchy.
- Streams can be chained to fulfil application requirements.

Primitive types IO

- DataInputStream & DataOutputStream -- convert primitive types from/to bytes
    - primitive type --> DataOutputStream --> bytes --> FileOutputStream --> file.
        - DataOutput interface provides methods for conversion - writeInt(), writeUTF(), writeDouble(), ...
    - primitive type <-- DataInputStream <-- bytes <-- FileInputStream <-- file.
        - DataInput interface provides methods for conversion - readInt(), readUTF(), readDouble(), ...

DataOutput/DataInput interface

- interface DataOutput
  - writeUTF(String s)
  - writeInt(int i)
  - writeDouble(double d)
  - writeShort(short s)
  - ...
- interface DataInput
  - String readUTF()
  - int readInt()
  - double readDouble()
  - short readShort()
  - ...

Serialization

- ObjectInputStream & ObjectOutputStream -- convert java object from/to bytes
  - Java object --> ObjectOutputStream --> bytes --> FileOutputStream --> file.
    - ObjectOutput interface provides method for conversion - writeObject().
  - Java object <-- ObjectInputStream <-- bytes <-- FileInputStream <-- file.
    - ObjectInput interface provides methods for conversion - readObject().
- Converting state of object into a sequence of bytes is referred as Serialization. The sequence of bytes includes object data as well as metadata.
- Serialized data can be further saved into a file (using FileOutputStream) or sent over the network (Marshalling process).
- Converting (serialized) bytes back to the Java object is referred as Deserialization.
- These bytes may be received from the file (using FileInputStream) or from the network (Unmarshalling process).

ObjectOutput/ObjectInput interface

- interface ObjectOutput extends DataOutput
  - writeObject(obj)
- interface ObjectInput extends DataInput
  - obj = readObject()

Serializable interface

- Object can be serialized only if class is inherited from Serializable interface; otherwise writeObject() throws NotSerializableException.
- Serializable is a marker interface.

transient fields

- writeObject() serialize all non-static fields of the class. If fields are objects, then they are also serialized.
- If any field is intended not to serialize, then it should be marked as "transient".

- The transient and static fields (except serialVersionUID) are not serialized.

serialVersionUID field

- Each serializable class is associated with a version number, called a serialVersionUID.
- It is recommended that programmer should define it as a static final long field (with any access specifier). Any change in class fields expected to modify this serialVersionUID.

```
private static final long serialVersionUID = 1001L;
```

- During deserialization, this number is verified by the runtime to check if right version of the class is loaded in the JVM. If this number mismatched, then InvalidClassException will be thrown.
- If a serializable class does not explicitly declare a serialVersionUID, then the runtime will calculate a default serialVersionUID value for that class (based on various aspects of the class described in the Java(TM) Object Serialization specification).

Buffered streams

- Each write() operation on FileOutputStream will cause data to be written on disk (by OS). Accessing disk frequently will reduce overall application performance. Similar performance problems may occur during network data transfer.
- BufferedOutputStream classes hold data into a in-memory buffer before transferring it to the underlying stream. This will result in better performance.
    - Java object --> ObjectOutputStream --> BufferedOutputStream --> FileOutputStream --> file on disk.
- Data is sent to underlying stream when buffer is full or flush() called explicitly.
- BufferedInputStream provides a buffering while reading the file.
- The buffer size can be provided while creating the respective objects.

PrintStream class

- Produce formatted output (in bytes) and send to underlying stream.
- Formatted output is done using methods print(), println(), and printf().
- System.out and System.err are objects of PrintStream class.

Scanner class

- Added in Java 5 to get the formatted input.
- It is java.util package (not part of java io framework).

```
Scanner sc = new Scanner(inputStream);
// OR
Scanner sc = new Scanner(inputFile);
```

- Helpful to read text files line by line.

Character streams

- Character streams are used to interact with text file.
- Java char takes 2 bytes (unicode), however char stored in disk file may take 1 or more bytes depending on char encoding.
    - https://www.w3.org/International/questions/qa-what-is-encoding
- The character stream does conversion from java char to byte representation and vice-versa (as per char encoding).
- The abstract base classes for the character streams are the Reader and Writer class.
- Writer class -- write operation
    - void close() -- close the stream
    - void flush() -- writes data (in memory) to underlying stream/device.
    - void write(char[] b) -- writes char array to underlying stream/device.
    - void write(int b) -- writes a char to underlying stream/device.
- Writer Sub-classes
    - FileWriter, OutputStreamWriter, PrintWriter, BufferedWriter, etc.
- Reader class -- read operation
    - void close() -- close the stream
    - int read(char[] b) -- reads char array from underlying stream/device
    - int read() -- reads a char from the underlying device/stream. Returns -1
- Reader Sub-classes
    - FileReader, InputStreamReader, BufferedReader, etc.

Process vs Threads

Program

- Program is set of instructions given to the computer.
- Executable file is a program.
- Executable file contains text, data, rodata, symbol table, exe header.

Process

- Process is program in execution.
- Program (executable file) is loaded in RAM (from disk) for execution. Also OS keep information required for execution of the program in a struct called PCB (Process Control Block).
- Process contains text, data, rodata, stack, and heap section.

Thread

- Threads are used to do multiple tasks concurrently within a single process.
- Thread is a lightweight process.
- When a new thread is created, a new TCB is created along with a new stack. Remaining sections are shared with parent process.

Process vs Thread

- Process is a container that holds resources required for execution and thread is unit of execution/scheduling.

- Each process have one thread created by default -- called as main thread.

Process creation (Java)

- In Java, process can be created using Runtime object.
- Runtime object holds information of current runtime environment that includes number of processors, JVM memory usage, etc.
- Current runtime can be accessed using static getRuntime() method.

```
Runtime rt = Runtime.getRuntime();
```

- The process is created using exec() method, which returns the Process object. This object represents the OS process and its waitFor() method wait for the process termination (and returns exit status).

```
String[] args = { "/path/of/executable", "cmd-line arg1", ... };
Process p = rt.exec(args);
int exitStatus = p.waitFor();
```

Multi-threading (Java)

- Java applications are always multi-threaded.
- When any java application is executed, JVM creates (at least) two threads.
  - main thread -- executes the application main()
  - GC thread -- does garbage collection (release unreferenced objects)
- Programmer may create additional threads, if required.

Thread creation

- To create a thread
  - step 1: Implement a thread function (task to be done by the thread)
  - step 2: Create a thread (with above function)
- Method 1: extends Thread

```
class MyThread extends Thread {
    @Override
    public void run() {
        // task to be done by the thread
    }
}
```

```
MyThread th = new MyThread();
th.start();
```

- Method 2: implements Runnable

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        // task to be done by the thread
    }
}
```

```
MyRunnable runnable = new MyRunnable();
Thread th = new Thread(runnable);
th.start();
```

- Java doesn't support multiple inheritance. If your class is already inherited from a super class, you cannot extend it from Thread class. Prefer Runnable in this case; otherwise you may choose any method.

```
// In Java GUI application is inherited from Frame class.
// to create run() in the same class, you must use Runnable
class MyGuiApplication extends Frame implements Runnable {
    // ...
    public void run() {
        // ...
    }
    // ...
}
```

start() vs run()

- run():
  - Programmer implemented code to be executed by the thread.
- start():
  - Pre-defined method in Thread class.
  - When called, the thread object is submitted to the (JVM/OS) scheduler. Then scheduler select the thread for execution and thread executes its run() method.

Thread methods

- static Thread currentThread()
  - Returns a reference to the currently executing thread object.
- static void sleep(long millis)

    - Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.

- static void yield()
    - A hint to the scheduler that the current thread is willing to yield its current use of a processor.
- Thread.State getState()
    - Returns the state of this thread.
    - State can be NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED
- void run()
    - If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called. If thread class extends from Thread class, this method should be overridden. The default implementation is empty.
- void start()
    - Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
- void join()
    - Waits for this thread to die/complete.
- boolean isAlive()
    - Tests if this thread is alive.
- void setDaemon(boolean daemon);
    - Marks this thread as either a daemon thread (true) or a user thread (false).
- boolean isDaemon()
    - Tests if this thread is a daemon thread.
- long getId()
    - Returns the identifier of this Thread.
- void setName(String name)
    - Changes the name of this thread to be equal to the argument name.
- String getName()
    - Returns this thread's name.
- void setPriority(int newPriority)
    - Changes the priority of this thread.
    - In Java thread priority can be 1 to 10.

    - May use predefined constants MIN_PRIORITY(1), NORM_PRIORITY(5), MAX_PRIORITY(10).

- int getPriority()
    - Returns this thread's priority.
- ThreadGroup getThreadGroup()
    - Returns the thread group to which this thread belongs.
- void interrupt()
    - Interrupts this thread -- will raise InterruptedException in the thread.
- boolean isInterrupted()
    - Tests whether this thread has been interrupted.

Daemon threads

- By default all threads are non-daemon threads (including main thread).
- We can make a thread as daemon by calling its setDaemon(true) method -- before starting the thread.
- Daemon threads are also called as background threads and they support/help the non-daemon threads.
- When all non-daemon threads are terminated, the Daemon threads get automatically terminated.

Core Java

Process vs Threads

Program

- Program is set of instructions given to the computer.
- Executable file is a program.
- Executable file contains text, data, rodata, symbol table, exe header.

Process

- Process is program in execution.
- Program (executable file) is loaded in RAM (from disk) for execution. Also OS keep information required for execution of the program in a struct called PCB (Process Control Block).
- Process contains text, data, rodata, stack, and heap section.

Thread

- Threads are used to do multiple tasks concurrently within a single process.
- Thread is a lightweight process.
- When a new thread is created, a new TCB is created along with a new stack. Remaining sections are shared with parent process.

Process vs Thread

- Process is a container that holds resources required for execution and thread is unit of execution/scheduling.
- Each process have one thread created by default -- called as main thread.

Process creation (Java)

- In Java, process can be created using Runtime object.
- Runtime object holds information of current runtime environment that includes number of processors, JVM memory usage, etc.
- Current runtime can be accessed using static getRuntime() method.

```
Runtime rt = Runtime.getRuntime();
```

- The process is created using exec() method, which returns the Process object. This object represents the OS process and its waitFor() method wait for the process termination (and returns exit status).

```
String[] args = { "/path/of/executable", "cmd-line arg1", ... };
Process p = rt.exec(args);
int exitStatus = p.waitFor();
```

Multi-threading (Java)

- Java applications are always multi-threaded.
- When any java application is executed, JVM creates (at least) two threads.
  - main thread -- executes the application main()
  - GC thread -- does garbage collection (release unreferenced objects)
- Programmer may create additional threads, if required.

Thread creation

- To create a thread
  - step 1: Implement a thread function (task to be done by the thread)
  - step 2: Create a thread (with above function)
- Method 1: extends Thread

```
class MyThread extends Thread {
    @Override
    public void run() {
        // task to be done by the thread
    }
}
```

```
MyThread th = new MyThread();
th.start();
```

- Method 2: implements Runnable

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        // task to be done by the thread
    }
}
```

```
MyRunnable runnable = new MyRunnable();
Thread th = new Thread(runnable);
th.start();
```

- Java doesn't support multiple inheritance. If your class is already inherited from a super class, you cannot extend it from Thread class. Prefer Runnable in this case; otherwise you may choose any method.

```
// In Java GUI application is inherited from Frame class.
// to create run() in the same class, you must use Runnable
class MyGuiApplication extends Frame implements Runnable {
    // ...
    public void run() {
        // ...
    }
    // ...
}
```

start() vs run()

- run():
  - Programmer implemented code to be executed by the thread.
- start():
  - Pre-defined method in Thread class.
  - When called, the thread object is submitted to the (JVM/OS) scheduler. Then scheduler select the thread for execution and thread executes its run() method.

Thread methods

- static Thread currentThread()
  - Returns a reference to the currently executing thread object.
- static void sleep(long millis)
  - Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
- static void yield()
  - A hint to the scheduler that the current thread is willing to yield its current use of a processor.
- Thread.State getState()
  - Returns the state of this thread.
  - State can be NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED
- void run()
  - If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called. If thread class extends from Thread class, this method should be overridden. The default implementation is empty.
- void start()
  - Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.

- void join()
    - Waits for this thread to die/complete.
- boolean isAlive()
    - Tests if this thread is alive.
- void setDaemon(boolean daemon);
    - Marks this thread as either a daemon thread (true) or a user thread (false).
- boolean isDaemon()
    - Tests if this thread is a daemon thread.
- long getId()
    - Returns the identifier of this Thread.
- void setName(String name)
    - Changes the name of this thread to be equal to the argument name.
- String getName()
    - Returns this thread's name.
- void setPriority(int newPriority)
    - Changes the priority of this thread.
    - In Java thread priority can be 1 to 10.
    - May use predefined constants MIN_PRIORITY(1), NORM_PRIORITY(5), MAX_PRIORITY(10).
- int getPriority()
    - Returns this thread's priority.
- ThreadGroup getThreadGroup()
    - Returns the thread group to which this thread belongs.
- void interrupt()
    - Interrupts this thread -- will raise InterruptedException in the thread.
- boolean isInterrupted()
    - Tests whether this thread has been interrupted.

Daemon threads

- By default all threads are non-daemon threads (including main thread).
- We can make a thread as daemon by calling its setDaemon(true) method -- before starting the thread.

- Daemon threads are also called as background threads and they support/help the non-daemon threads.
- When all non-daemon threads are terminated, the Daemon threads get automatically terminated.

Thread life cycle

- Thread.State state = th.getState();
- NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED
    - NEW: New thread object created (not yet started its execution).
    - RUNNABLE: Thread is running on CPU or ready for execution. Scheduler picks ready thread and dispatch it on CPU.
    - BLOCKED: Thread is waiting for lock to be released. Thread blocks due to synchronized block/method.
    - WAITING: Thread is waiting for the notification. Waiting thread release the acquired lock.
    - TIMED_WAITING: Thread is waiting for the notification or timeout duration. Waiting thread release the acquired lock.
    - TERMINATED: Thread terminates when run() method is completed, stopped explicitly using stop(), or an exception is raised while executing run().

Synchronization

- When multiple threads try to access same resource at the same time, it is called as Race condition.
- Example: Same bank account undergo deposit() and withdraw() operations simultaneously.
- It may yield in unexpected/undesired results.
- This problem can be solved by Synchronization.
- The synchronized keyword in Java provides thread-safe access.
- Java synchronization internally use the Monitor object associated with any object. It provides lock/unlock mechanism.
- "synchronized" can be used for block or method.
- It acquires lock on associated object at the start of block/method and release at the end. If lock is already acquired by other thread, the current thread is blocked (until lock is released by the locking thread).
- "synchronized" non-static method acquires lock on the current object i.e. "this". Example:

```
class Account {
    // ...
    public synchronized void deposit(double amount) {
        double newBalance = this.balance + amount;
        this.balance = newBalance;
    }
    public synchronized void withdraw(double amount) {
        double newBalance = this.balance - amount;
        this.balance = newBalance;
    }
}
```

- "synchronized" static method acquires lock on metadata object of the class i.e. MyClass.class. Example:

```
class MyClass {
    private static int field = 0;
    // called by incThread
    public synchronized static void incMethod() {
        field++;
    }
    // called by decThread
    public synchronized static void decMethod() {
        field--;
    }
}
```

- "synchronized" block acquires lock on the given object.

```
// assuming that no method in Account class is synchronized.

// thread1
synchronized(acc) {
    acc.deposit(1000.0);
}

// thread2
synchronized(acc) {
    acc.withdraw(1000.0);
}
```

- Alternatively lock can be acquired using RentrantLock since Java 5.0. Example code:

```
class Example {
    private final ReentrantLock rl = new ReentrantLock();
    public void method() {
        rl.lock();
        try {
            // ...
        }
        finally {
            rl.unlock();
        }
    }
}
```

- Synchronized collections
    - Synchronized collections (e.g. Vector, Hashtable, ...) use synchronized keyword (block/method) to handle race conditions.

Core Java

Multi-Threading

Multi-Threading

Inter-thread communication

- wait()
  - Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
  - The current thread must own this object's monitor i.e. wait() must be called within synchronized block/method.
  - The thread releases ownership of this monitor and waits until another thread notifies.
  - The thread then waits until it can re-obtain ownership of the monitor and resumes execution.
- notify()
  - Wakes up a single thread that is waiting on this object's monitor.
  - If multiple threads are waiting on this object, one of them is chosen to be awakened arbitrarily.
  - The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object.
  - This method should only be called by a thread that is the owner of this object's monitor.
- notifyAll()
  - Wakes up all threads that are waiting on this object's monitor.
  - The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object.
  - This method should only be called by a thread that is the owner of this object's monitor.

Member/Nested classes

- By default all Java classes are top-level.
- In Java, classes can be written inside another class/method. They are Member classes.
- Four types of member/nested classes
  - Static member classes -- demo11_01
  - Non-static member class -- demo11_02
  - Local class -- demo11_03
  - Annoymous Inner class -- demo11_04
- When .java file is compiled, separate .class file created for outer class as well as inner class.

Static member classes

- Like other static members of the class (belong to the class, not the object).
- Accessed using outer class (Doesn't need the object of outer class).
- Can access static (private/public) members of the outer class directly.
- Static member class cannot access non-static members of outer class directly.
- The outer class can access all members (including private) of inner class directly (no need of getter/setter).

- The static member classes can be private, public, protected, or default.

```
class Outer {
    private int nonStaticField = 10;
    private static int staticField = 20;

    public static class Inner {
        public void display() {
            System.out.println("Outer.nonStaticField = " + nonStaticField);
// error
            System.out.println("Outer.staticField = " + staticField); // ok
- 20
        }
    }
}
public class Main {
    public static void main(String[] args) {
        Outer.Inner obj = new Outer.Inner();
        obj.display();
    }
}
```

Non-static member classes/Inner classes

- Like other non-static members of the class (belong to the object/instance of Outer class).
- Accessed using outer class object (Object of outer class is MUST).
- Can access static & non-static (private) members of the outer class directly.
- The outer class can access all members (including private) of inner class directly (no need of getter/setter).
- The non-static member classes can be private, public, protected, or default.

```
class Outer {
    private int nonStaticField = 10;
    private static int staticField = 20;
    public class Inner {
        public void display() {
            System.out.println("Outer.nonStaticField = " + nonStaticField);
// ok-10
            System.out.println("Outer.staticField = " + staticField); // ok-
20
        }
    }
}
public class Main {
    public static void main(String[] args) {
        //Outer.Inner obj = new Outer.Inner(); // compiler error
```

```
            // create object of inner class
                //Outer outObj = new Outer();
                //Outer.Inner obj = outObj.new Inner();
            Outer.Inner obj = new Outer().new Inner();
            obj.display();
        }
    }
```

- If Inner class member has same name as of outer class member, it shadows (hides) the outer class member. Such Outer class members can be accessed explicitly using `Outer.this`.

Static member class and Non-static member class -- Application

```
// top-level class
class LinkedList {
    // static member class
    static class Node {
        private int data;
        private Node next;
        // ...
    }
    private Node head;
    // non-static member class
    class Iterator {
        private Node trav;
        // ...
    }
    // ...
    public void display() {
        Node trav = head;
        while(trav != null) {
            System.out.println(trav.data);
            trav = trav.next;
        }
    }
}
```

Local class

- Like local variables of a method.
- The class scope is limited to the enclosing method.
- If enclosed in static method, behaves like static member class. If enclosed in non-static method, behaves like non-static member class.
- Along with Outer class members, it can also access (effectively) final local variables of the enclosing method.
- We can create any number of objects of local classes within the enclosing method.

```
public class Main {
    private int nonStaticField = 10;
    private static int staticField = 20;
    public static void main(String[] args) {
        final int localVar1 = 1;
        int localVar2 = 2;
        int localVar3 = 3;
        localVar3++;
        // local class (in static method) -- behave like static member class
        class Inner {
            public void display() {
                System.out.println("Outer.nonStaticField = " +
nonStaticField); // error
                System.out.println("Outer.staticField = " + staticField); //
ok 20
                System.out.println("Main.localVar1 = " + localVar1); // ok 1
                System.out.println("Main.localVar2 = " + localVar2); // ok 2
                System.out.println("Main.localVar3 = " + localVar3); //
error
            }
        }
        Inner obj = new Inner();
        obj.display();
        //new Inner().display();
    }
}
```

Annoymous Inner class

- Creates a new class inherited from the given class/interface and its object is created.
- If in static context, behaves like static member class. If in non-static context, behaves like non-static member class.
- Along with Outer class members, it can also access (effectively) final local variables of the enclosing method.

```
// (named) local class
class EmpnoComparator implements Comparator<Employee> {
    public int compare(Employee e1, Employee e2) {
        return e1.getEmpno() - e2.getEmpno();
    }
}
Arrays.sort(arr, new EmpnoComparator());    // anonymous obj of local class
```

```
// Anonymous inner class
Comparator<Employee> cmp = new Comparator<Employee>() {
    public int compare(Employee e1, Employee e2) {
```

```
            return e1.getEmpno() - e2.getEmpno();
        }
    };
    Arrays.sort(arr, cmp);
```

Date and Time API

- Legacy Java classes
  - Date
  - Calendar
- Java 8 Date Time
  - https://www.baeldung.com/java-8-date-time-intro

Java NIO

- Java NIO (New IO) is an alternative IO API for Java.
- Java NIO offers a different IO programming model than the traditional IO APIs.
- Since Java 7.
- Java NIO enables you to do non-blocking (not fully) IO.
- Java NIO consist of the following core components:
  - Channels e.g. FileChannel, ...
  - Buffers e.g. ByteBuffer, ...
  - Selectors
- Java NIO also provides "helper" classes Paths & Files.
  - exists()
  - ...

Paths and Files

- A Java Path instance represents a path in the file system. A path can point to either a file or a directory. A path can be absolute or relative.

```
Path path = Paths.get("c:\\data\\myfile.txt");
```

- Files class (Files) provides several static methods for manipulating files in the file system.

```
static InputStream newInputStream(Path, OpenOption...) throws IOException;
static OutputStream newOutputStream(Path, OpenOption...) throws IOException;
static DirectoryStream<Path> newDirectoryStream(Path) throws IOException;
static Path createFile(Path, attribute.FileAttribute<?>...) throws
IOException;
static Path createDirectory(Path, attribute.FileAttribute<?>...) throws
IOException;
static void delete(Path) throws IOException;
static boolean deleteIfExists(Path) throws IOException;
static Path copy(Path, Path, CopyOption...) throws IOException;
static Path move(Path, Path, CopyOption...) throws IOException;
```

```
static boolean isSameFile(Path, Path) throws IOException;
static boolean isHidden(Path) throws IOException;
static boolean isDirectory(Path, LinkOption...);
static boolean isRegularFile(Path, LinkOption...);
static long size(Path) throws IOException;
static boolean exists(Path, LinkOption...);
static boolean isReadable(Path);
static boolean isWritable(Path);
static boolean isExecutable(Path);
static List<String> readAllLines(Path) throws IOException;
static Stream<String> lines(Path) throws IOException;
```

Channels and Buffers

- All IO in NIO starts with a Channel. A Channel is similar to IO stream. From the Channel data can be read into a Buffer. Data can also be written from a Buffer into a Channel.

NIO Channels

- Java NIO Channels are similar to IO streams with a few differences:
  - You can both read and write to a Channels. Streams are typically one-way (read or write).
  - Channels can be read and written asynchronously (non-blocking).
  - Channels always read to, or write from, a Buffer.
- Channel Examples
  - FileChannel
  - DatagramChannel // UDP protocol
  - SocketChannel, ServerSocketChannel // TCP protocol

NIO Buffers

- A buffer is essentially a block of memory into which you can write data, which you can then later read again. This memory block is wrapped in a NIO Buffer object, which provides a set of methods that makes it easier to work with the memory block.
- Using a Buffer to read and write data typically follows this 4-step process:
  - Write data into the Buffer
  - Call buffer.flip()
  - Read data out of the Buffer
  - Call buffer.clear() or buffer.compact()
- Buffer Examples
  - ByteBuffer
  - CharBuffer
  - DoubleBuffer
  - FloatBuffer
  - IntBuffer
  - LongBuffer
  - ShortBuffer

Channel and Buffer Example

```
RandomAccessFile aFile = new RandomAccessFile("somefile.txt", "rw");
FileChannel inChannel = aFile.getChannel();

ByteBuffer buf = ByteBuffer.allocate(32);

int bytesRead = inChannel.read(buf); // write data into buffer (from channel)
while (bytesRead != -1) {
    System.out.println("Read " + bytesRead);
    buf.flip(); // switch buffer from write mode to read mode

    while(buf.hasRemaining()){
        System.out.print((char) buf.get()); // read data from the buffer
    }

    buf.clear(); // clear the buffer
    bytesRead = inChannel.read(buf);
}
aFile.close();
```

RandomAccessFile

- RandomAccessFile class from java.io package.
- Capable of reading and writing into a file (on a storage device).
- Internally maintains file read/write position/cursor.
- Homework: Read docs.

Java NIO vs Java IO

- IO: Stream-oriented
- NIO: Buffer-oriented
- IO: Blocking IO
- NIO: Non-blocking IO

Platform Independence

- Java is architecture neutral i.e. can work on various CPU architectures like x86, ARM, SPARC, PPC, etc (if JVM is available on those architectures).
- Java is NOT fully platform independent. It can work on various platforms like Windows, Linux, Mac, UNIX, etc (if JVM is available on those platforms).
- Few features of Java remains platform dependent.
    - Multi-threading (Scheduling, Priority)
    - File IO (Performance, File types, Paths)
    - AWT GUI (Look & Feel)
    - Networking (Socket connection)