Agenda

- OOSD and OOP Revision
- Introduction
- Command line compilation
- JRE/JDK/JVM
- Hello World in STS
- Console input and output(Scanner & Console clas)
- Language Fundamentals

Java History

- In 1991, group of sun engineers led by James Gosling and Patrick Naughton decided to design a language that could run on small devices like remote controls, cable tv boxes.
- Since these devices have very small power and memory the language needs to be small.
- Also differnt manufactures can choose different CPU's the language cannot be bound to single architecture
- this project was named as green.
- these engineers came from UNIX background, so they used c++ as their base.
- James decided to call this language as OAK, however the language with this name was already existing, Hence it was later renamed by James to Java.
- In 1992 they delivered their first product called as "*7"(a smart remote control)
- Unfortunately Sun Miccrosystem was not intrested in producing this, also nor the consumer electronic companies were intrested in it.
- The team then deciced to market their technology in some other way where they worked for next 1 and half year on it.
- Meanwhile world wide web (www) was growing bigger.
- the key to it was browser transalating hyper text pages to the screen.
- the java developers developed a browser called as HotJava browser which was based on client server architeture and was working in real time.
- the developers made the browser capable of executing java code inside the web pages called as Applets.

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
- 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 it is implemented for wide variety of hardware and operating system
- 1. Java Card
    - used to run java based applications on small devices with small memory devices like smart cards
- 2. Java ME(Micro Edition)
    - used to develop applications for small devices with less memory, display and power capacities like mobiles, printers
- 3. Java SE(Standard Edition)
    - It is widely used for development of portable code for desktop environment
- 4. Java EE(Enterprise Edition)
    - It is widely used in development of enterprise applications/softwares. -also used for web application development

Java Installation

- Windows and Mac:
- Download .msi/.dmg file and follow installation steps.

```
https://adoptium.net/temurin/releases/?version=11
```

- Ubuntu:

```
sudo apt install openjdk-11-jdk
```

JDK vs JRE vs JVM

- SDK -> Software Development Kit

- SDK = Software Development Tools + Libraries + Runtime environment + Documentation + IDE
  - Software Development Tools = Compiler, Debugger, etc.
  - Libraries = Set of functions/classes.
- JDK -> Java Development kit
  - used for developing Java applications.
- JDK = Java Development tools + Java docs + JRE
- JRE = Java API(Java class libraries) (rt.jar replaced by jmods in java9) + Java Virtual Machine
  - till java 8
    - JRE = rt.jar + JVM
  - from java 9
    - JRE = jmods + libraries + JVM

Directory Structure of JDK

- Ubuntu: /usr/lib/jvm/java-11-openjdk-amd64

```
openjdk-11.0.22
  |- bin: Contains executable binaries like java, javac, etc.
  |- jmods: Contains JMOD (Java Modular Archive) files for Java modules similar to
JAR (Java Archive) (available from Java 9 onwards) .
  |- lib: Contains libraries and other resources.
  |- man: Contains manual pages (man pages) for Java commands.
  |- src.zip: Contains Java source code for the JDK (not always included in all
distributions).
```

Eclipse STS 4.x

- check your hardware architecture and download the latest STS and extract it.

```
  https://spring.io/tools
```

Documentation and tutorial Link

```
- Java SE 8 Document Link
https://docs.oracle.com/javase/8/docs/api/index.html

- Java SE 11 Document Link
https://docs.oracle.com/javase/11/docs/api/index.html

- Oracle Java Tutorial
https://docs.oracle.com/javase/tutorial/
```

Object Oriented

- basic principles of OOP are
- 1. class
- 2. object

class

- It is a logical entity
- It is a user defined datatype (same as struct in c)
- It consists of field(data members) and methods(member functions)
- Methods
  - static methods -> Accessed using classname directly
  - non static methods-> Accessed using object of the class
- It is also called as blueprint of object/instance

Object

- It is a physical entity
- It is an instance of a class
- one class can have multiple objects
- Object is created in java using new operator

HelloWorld

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

main()

- In java every variable/function should be class (Encapsulation)
- JVM calls main method without creating object of the class, so main method should be static
- main does not return anythinig to JVM so it is void.
- main takes command line arguments and hence String args[]
- main should be accessiable outside the class directly and hence public.
- JVM invokes main method.
- Can be overloaded.
- Can write one entry-point in each Java class.

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

Compilation & Execution

In same directory

```
javac Program.java
java Program
```

In src and bin directory

- create a directory called as RectangleArea
- add two subdirectories src and bin inside it.
- in src add Rectangle.java file
- from the src directory open the terminal.

```
javac -d ../bin Rectangle.java

//For Windows
set CLASSPATH=..\bin

// For Linux
export CLASSPATH=../bin

java Rectangle
```

CLASSPATH

- It is a JAVA environment variable which holds all directories seprated by ;(Windows) :(Linux)
- It informs java compiler, application launcher, JVM, and other java tools about the directories in which classes/packages are kept(location of the class files)
- To display CLASSPATH variable
  - Windows cmd> set CLASSPATH
  - Linux terminal> echo $CLASSPATH

C/C++ vs Java Compilation and Execution

- main.cpp -->compiler --> main.o --> linker --> main.out OR main.exe
- Main.java --> compiler --> Main.class --> JVM
  - Main.java -> Source code
  - Main.class -> Byte code

Bytecode

- Bytecode is an intermediate representation of a program that is generated by a compiler and typically executed by a virtual machine.
- In the context of Java programming, bytecode refers specifically to the binary format that Java source code is compiled into.

- It enables platform independence, portability, security, and potential performance optimizations in Java programming.
- It forms a crucial part of the Java platform's architecture, allowing Java programs to run on a wide range of devices and operating systems.

.class File

- A .class file in Java contains the bytecode instructions that are executed by the Java Virtual Machine (JVM).
- When you compile Java source code using a Java compiler like javac, it translates the source code into bytecode, which is then stored in a .class file.
- Here's what a .class file typically consists of:

1. Magic Number and Version Information:
    - The file starts with a magic number 0xCAFEBABE to identify it as a valid Java class file.
    - This is followed by version information indicating the version of the Java bytecode format used.
2. Constant Pool:
    - The constant pool is a table of structures that contains various types of constants used by the class file, such as strings, numeric literals, class and interface names, method and field references, and more.
    - It's a crucial part of the class file and provides the foundation for the bytecode instructions.
3. Access Flags:
    - Access flags specify the access level of the class or interface, such as whether it's public, private, final, abstract, etc.
4. This Class and Superclass Information:
    - The .class file contains information about the class itself (its name, superclass, and interfaces it implements).
5. Fields:
    - Information about the fields (variables) declared in the class, including their names, types, and access modifiers.
6. Methods:
    - Information about the methods (functions) declared in the class, including their names, return types, parameter types, and access modifiers.
7. Attributes:
    - Attributes provide additional metadata about various elements in the class file. For example, they may include debugging information, annotations, runtime-visible annotations, source file names, etc.
8. Code:

- For methods that contain executable code, such as constructors or regular methods, there's a Code attribute that contains the bytecode instructions to be executed by the JVM.

9. Exception Table:
    - If a method contains exception handling code (try-catch blocks), this section contains information about how exceptions are handled.

Public class

- As per Java Langauage Specification
    - 1. Name of public class and name of java file should be same.
    - 2. A single .java file can have only 1 public class.
    - 3. A single .java file can have multiple non public classes.

main() Variations

- In STS .class files are placed under bin directory after auto compilation
- one java project can have multiple .java files.
- each java file can have its own main method which can be executed seperately
- the main() must be public static void main otherwise we get an error.
- the entry point method must be be main(String args[]) otherwise error main 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.

Console input and output

- Java has several ways to take input and print output. Most popular ways in Java 8 are
- 1. using Scanner class (Program01)
    - it is present in java.util package

```
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);
```

- 2. using Console class
    - it is present in java.io package

```
Console console = System.console();
String email = console.readLine("Enter Email: ");
```

```
char[] passwd = console.readPassword("Enter Password: ");
console.printf("Email: %s\n", email);
```

Variable

- 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.

Java Method

- 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).
- Functions in C/CPP are called as Method in java.

Agenda

- Language Fundamentals
- Packages
- Access Modifiers
- ~~this reference~~
- ~~Types of Methods~~
- ~~Constructor Chaning~~

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 EmployeeManagement{
}
```

- Constat Fields
    - must be in capital letters only
    - for eg -

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

- package names

- package names should be lower case only
- for eg -> java.lang

comments

```
// Single line comment.
/* Multi line comments */
/** Documentation comments */
```

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

DataTypes

- It defines 3 things
    - 1. Nature (type of data stored)
    - 2. Memory (Memory required to store the data)
    - 3. Operations (what operations we can perform)
- Java is Strictly type checked language
- In java, data types are classified as:

1. Primitive types or Value types
2. 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
```

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	-9223372036854775807 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

Literals

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

```
int num1 = 10; // Integral

float num2 = 123.456f; // floating point

char ch = 'c'; // character literal

String name = "sunbeam";// string literal

boolean status = true or false; // boolean literal

String s = null;// 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.
  - representation of floating-point numbers using 32 bits.
  - single precision is known as "binary32".
  - typically provide about 7 decimal digits of precision.
- Double precision (8 bytes) floating-point number. Suffix d or D.
  - representation of floating-point numbers using 64 bits.
  - double precision is known as "binary64".
  - typically provide about 15-16 decimal digits of precision.
- 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;
```

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	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

widening & Narrowing

- converting state of primitive value of narrower type into wider type is called as widening

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

- converting state of primitive value of wider type into narrow type is called as 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.

Wrapper classes

- 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
```

Applications of wrapper classes

1. Use primitive values like objects

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

2. 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);
```

3. 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);
```

4. 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));
```

- 5. Java collections only store object types and not primitive types

Boxing & UnBoxing

- Converting value from primitive type to reference type is called as boxing
- Converting value from refrence type to primitive type is called as unboxing

```
int num1 = 10;
Integer i1 = new Integer(num1); // boxing
Integer i2 = num1; // auto-boxing

Integer i3 = new Integer(20);
int num2 = i3.intValue(); // unboxing
int num3 = i3; // auto-unboxing
```

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, String , System
    - java.util -> Scanner
    - java.io -> PrintStream, Console
    - java.sql -> Connection, Statement
- 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.
- When compiled, packages are created in form of directories (and sub-directories).

```
package com.sunbeaminfo.kdac
```

creating package using command line execution

- cretae a folder demo01
- create 2 sub directories src and bin
- write a .java file with the package p1.
- use below steps for compilation and execution

```
javac -d ../bin Program.java

export CLASSPATH=../bin

java p1.Program

// if without setting classpath we want to execute the java code use below command
java -cp ../bin p1.Program
```

- for multiple files in multiple packages (Demo02)

```
javac -d ../bin Time.java
export CLASSPATH=../bin

//add import statement inside the Program.java file
javac -d ../bin Program.java
```

```
java p1.Program
```

- If the class is not kept public, the class won't be able to be accessed in other packages

Access Modifiers

- For class
    - 1. default
    - 2. public
- For class members
    - 1. private
        - only within the class directly
    - 2. default (package level private)
        - in same class directly
        - in all the classes in the same package on class object
    - 3. protected
        - in same class directly
        - in all the classes in the same package on class object
        - in subclasses directly
    - 4. public
        - are visible every where.

		In same Package		In Different Package	
Access Modifier	**Same class**	**Other Class**	**Sub Class**	**Other class**	**Sub Class**
private	A	NA	NA	NA	NA
default	A	A	A	NA	NA
protected	A	A	A	NA	A
public	A	A	A	A	A

Difference between protected and default

- Default access restricts visibility to only classes within the same package. This allows you to encapsulate implementation details that are not intended to be accessed by classes outside the package.
- Protected access, on the other hand, allows access by subclasses (regardless of package) and by other classes within the same package.
- If you want to hide implementation details from all classes, including subclasses, default access provides stricter encapsulation.

Agenda

- Class,Reference and Object
- this reference
- types of methods
- Ctor chaning
- Initializers
- Array

class

- Logical entity
- blueprint of an object
- consists of fields and methods
- it is a reference type in java

Object

- physical enity
- instance of the class
- It defines three things
    - 1. state
    - 2. Behaviour
    - 3. identity
- the objects are created using new operator
- objects of the class are always created on heap.
- process of creating object of the class is also called as instantiation

Reference

- class is a reference type in java.
- variable created of a class is called as reference.
- reference variables get space on the stack.
- reference variables stores the address

Points to remember

- in java all local variables need to be initialized before using
- class fileds are assigned with the default values
- primitive types are assigned 0 while references are assigned null

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"
- "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.

Types of Methods

1. constructor
2. setters
    - Used to set value of the field from outside the class.
    - It modifies state of the object.
3. getters
    - Used to get value of the field outside the class.
4. facilitators
    - Provides additional functionalities
    - Business logic methods

Constructor

- It is a special method of the class
- In Java fields have default values if unitialized
- Primitive types default value is usually zero
- Reference type default value is null
- Constructor should initialize fields to the desired values.
- Types of Constructor
    - 1. Default/Parameterless Ctor
    - 2. Parameterized Ctor

Constructor Chaning

- Constructor chaining is executing a constructor of the class from another constructor (of the same class).
- Constructor chaining (if done) must be on the very first line of the constructor.

Object/Field Initializer

- In C++/Java Fields of the class are initialized using constructor
- In java, field can also be initialized using
    - 1. field initializer
    - 2. object initializer

    - 3. Constructor

Array

- Array is collection of similar data elements. Each element is accessible using indexes
- It is a reference type in java
- its object is created using new operator (on heap).
- The array of primitive type holds values (0 if uninitialized) and array of non-primitive type holds references (null if uninitialized).
- In Java, checking array bounds is responsibility of JVM. When invalid index is accessed, ArrayIndexOutOfBoundsException is thrown.
- Array types are
    - 1. 1-D array
    - 2. 2-D/Multi-dimensional array
    - 3. Ragged array
        - In 2D array if the second dimension of array is having differnt length then such array is called as Ragged Array

Agenda

- Array
- Variable Arity/Argument Method
- Method Arguments
    - pass by value and reference
- Method Overloading
- final Keyword
- static Keyword
- ~~Singleton Design Pattern~~

Method Overloading

- Defining methods with same name but differnt arguments(signature) is called as method overloading
- Arguments can differ in one of the following ways

1. No of parameters should be different
2. If no of parameters are same then their type of parameters should be different
3. If no and type are same then the order of paramters should be different

- Count (no of parameters)

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

- type of parameter

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

- Order or parameters

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

- Note that return type is NOT considered in method overloading.

Variable Arity/Argument Method

- It is a method which can take variable no of arguments.
- We can also pass array to this method.
- If we want to pass different types of variables to this arity method then we can use the object as the type.

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.
- Pass by value -- Creates copy of the variable. Changes done in called method are not available in calling method.
- Pass by reference for value/primitive types can be simulated using array.

final

- In Java, const is reserved word, but not used.
- Java has final keyword instead.
- It can be used for
    - variables
    - fields
    - methods
    - class
- if variables and fields are made final, they cannot be modified after initialization.
- final fields of the class must be initialized using any of the following below
    - field initializer
    - object initializer
    - constructor
- final methods cannot be overriden, final class cannot be extended(we will see at the time of inheritance)

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
    - 1. static fields

    - 2. static methods
    - 3. static block
    - 4. static import
- Note that, static local variables cannot be created in Java.

1. 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).
- eg :
    - Integer.SIZE
- Similar to field initializer, static fields can be initialized at declaration.

2. Static methods

- These 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 we need to call a method without creating object, then make such methods as static.
- Since static methods are designed to be called on class name, they do not have "this" reference. Hence, they cannot access non-static members in the static method (directly), However, we can access them on an object reference if created inside them.
- eg:
    - Integer.valueOf(10);
    - Factory Methods -> to cretae object of the class

static Field Initializer

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

```
static double roi = 5000.0;
// static final field -- constant
static final double PI = 3.142;
```

static Initializer 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.

Agenda

- Singleton Design pattern
- Hirerachy
    - Association
    - Inheritance
    - Types of Inheritance
- Method Overiding
- Upcasting

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.).

Singleton Design Pattern

- Singleton is a design pattern.
- 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.

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
    - 1. Composition
    - 2. Aggregation

Composition

- Represents part-of relation i.e. tight coupling between the objects.
- The inner object is essential part of outer object.
- Heart is part of Human.
- Engine is part of Car.
- Wall is part of Room.
- joining date is a part of employee

Aggegration

- 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.
- Room has a window
- Employee has a vehicle

Inheritance

- If "is-a"/"kind-of" relationship exist between the types, then use inheritance.
- Inheritance is process of generalization to specialization.
- All members of parent class are inherited to the child class.
- Parent class is also called as super class and child class is also called as sub-class.
- Example:
    - Manager is a Employee
    - Mango is a Fruit
    - Rectangle is a Shape
- In Java, inheritance is done using extends keyword.
- 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.
- used for 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.
- 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.

Types of Inheritance

- 1. Single

```
class A {

}
class B extends A{

}
```

- 2. Multiple

```
class A {

}
class B {

}
class C extends A,B{ // Not Allowed

}

interface I1{

}
interface I2{

}

interface I3 extends I1,I2{ // Allowed

}

class D implements I1,I2{ // Allowed

}
```

- 3. Hirerachical

```
class A {

}
class B extends A{

}
class C extends A{

}
```

- 4. Multilevel

```
class A {

}
class B extends A{

}
class C extends B{
```

```
}
```

- Hybrid inheritance: Any combination of above types

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
    1. Super-class has not provided method implementation at all (abstract method).
    2. 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).
    3. Sub-class needs different implementation than that of super-class method implementation.

Rules of method overriding in Java

1. Each method in Java can be overridden unless it is private, static or final.
2. Sub-class method must have same or wider access modifier than super-class method.
3. Arguments of sub-class method must be same as of super-class method.
4. 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.
5. Checked exception list in sub-class method should be same or subset of exception list in super-class method.

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

Upcasting

- 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, only super-class methods inherited into sub-class can be called. This is "Object slicing".
- Using such super-class reference, super-class methods overridden into sub-class can also be called.

Agenda

- Upcasting & Downcasting
- Final Method & Class
- Object class
    - Methods of object class
        - toString()
        - equals()
- Abstract class/method
- ~~Interfaces~~
- ~~Marker interfaces~~

Downcasting

- 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
- It has two types
    1. compile time
        - implemented using 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".
    2. Runtime - implemented using 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".

instanceof operator

- Java's instanceof operator checks if given reference points to the object of given type (or its sub-class) or not. Its result is boolean.
- Typically "instanceof" operator is used for type-checking before down-casting.

```
Person p = new SomeClass();
if(p instanceof Employee) {
    Employee e = (Employee)p;
```

```
        System.out.println("Salary: " + e.getSalary());
    }
```

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 hierarchy.
- 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)
    - public Object();
    - public native int hashCode();
    - public boolean equals(Object);
    - protected native Object clone() throws CloneNotSupportedException;
    - public String toString();
    - protected void finalize() throws Throwable;
    - public final native Class<?> getClass();
    - 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

- it is a non final method of object class
- 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.

equals() method

- It is non final method of object class
- 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.
- It is recommended to override hashcode method along when equals method is overriden.

Abstract Methods

- 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.
- 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 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.
- Example:
- java.lang.Number
- java.lang.Enum

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.

Agenda

- Interface
- Garbage Collector
- JVM Architecture
- Java Buzzwords
- ~~Date/LocalDate/Calendar~~

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.
- 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, static methods, and constructors.
- Examples:
  - java.io.Closeable / java.io.AutoCloseable
  - java.lang.Runnable
- Multiple interface inheritance is allowed in Java

```
interface Displayable {
void display();
}
interface Acceptable {
void accept();
}
class Employee implements Displayable,Acceptable{}
```

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

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

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.

Garbage Collector

- 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:

```
// 1. Nullify the reference.
MyClass obj = new MyClass();
obj = null;

//2. Reassign the reference.
MyClass obj = new MyClass();
obj = new MyClass();

//3. Object created locally in method.
void method() {
MyClass obj = new MyClass();
// ...
}

//4. Island of isolation i.e. objects are referencing each other, but not
```

```
referenced externally.

class Test {
    Test tref;
}

public class Program {

    public static void main(String[] args) {
        Test t1 = new Test();
        Test t2 = new Test();

        t1.tref = t2;
        t2.tref = t1;

        t1 = null;
        t2 = null;
    }
}
```

- 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 Test {
    Scanner sc = new Scanner(System.in);

    @Override
    protected void finalize() throws Throwable {
        sc.close();
    }
}

public class Program{

    public static void main(String[] args) {
        Test t1 = new Test();
        t1 = null;
        System.gc();// request GC
    }
}
```

- GC can be requested (not forced) by one of the following.

    1. System.gc();
    2. Runtime.getRuntime().gc();

- GC is of two types i.e. Minor and Major.

    1. Minor GC: Unreferenced objects from young generation are reclaimed. Objects not reclaimed here are moved to old/permanent generation.
    2. Major GC: Unreferenced objects from all generations are reclaimed. This is unefficient (slower process).

- JVM GC internally use Mark and Compact algorithm.
- GC Internals: https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

JVM Archicecture

- 1. Compilation
  - .class file is cretaed which consists of byte code
- 2. Byte Code
  - It is a machine level instructions that gets executed by the JVM
  - JVM converts byte code into target machine/native code
- 3. Execution
  - java is a tool used to execute the .class file.
  - It loads the .class file and invokes jvm for executing the file from the classpath
- JVM Archiceture Overview
  - ClassLoader + Memory Area + Execution Engine

ClassLoader SubSystem

- It loads and initialize the class

1. Loading

- Three types of classLoaders
  - 1. BootStrap classloader that loads built in java classes from jre/lib jars (rt.jar)
  - 2. Extension classloader that loads the extended classes from jre/lib/ext directory
  - 3. Application classloader that loads the classes from the application classpath
- It reads the classes from the disk and loads into JVM method(memory) area

2. Linking

- Three steps
  - 1. Verifiaction : Bytecode verifier ensures that class is compiled by valid compiler and not tampered
  - 2. Preparation : Memory is allocated for static members and initialized with default values
  - 3. Resolution : Symbolic references in constant pool are replaced by the direct references

3. Initialization

- All static variables of class are assigned with their assigned values(field initializers)
- all static blocks are executed if present

Memory Areas

- Their are 5 memory areas

    - 1. Method Area
    - 2. heap Area
    - 3. Stack Area
    - 4. PC Registers
    - 5. Native Method Stack Area

1. Method Area

- Create during JVM startup
- shared by all the threads
- class contents (for all classes) loaded into this area
- Method area also holds constant pool for all loaded classes.

2. Heap Area

- Create during JVM startup
- shared by all the threads
- All allocated objects (with new) are stored in heap
- The string pool is part of heap Area.
- The class Metadata is stored in a java.lang.Class object (in heap) once class is loaded.

3. Stack Area

- Separate stack is created for each thread in JVM (when thread is created).
- When a method is called 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.

4. 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.

5. Native Method Stack

- 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 component of JVM
- Convert byte code into machine code and execute it (instruction by instruction).
- It consists of
    - 1. Interpreter
    - 2. JIT Compiler
    - 3. Garbage Collector

1. Interpreter

- 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).

2. 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.

3. 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.

4. 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.

Java BuzzWords

- 1. Simple
  - Simple for Professional Programmers if aware about OOP.
  - It removed the complicated fetaures like pointers and rarely used features like operator overloading from c++
  - It was simple till java 1.4
  - the new features added made it powerful (but also complex)
- 2. Object Oriented
  - Java is a object-oriented programming language.
  - It supports all the pillars of OOP
- 3. Distributed
  - Java is designed to create distributed applications on networks.
  - Java applications can access remote objects on the Internet as easily as they can do in the local system.
  - Java enables multiple programmers at multiple remote locations to collaborate and work together on a single project.
- 4. Compiled and Interpreted
  - Usually, a computer language is either compiled or Interpreted.
  - Java combines both this approach and makes it a two-stage system.
  - Compiled: Java enables the creation of cross-platform programs by compiling them into an intermediate representation called Java Bytecode.
  - Interpreted: Bytecode is then interpreted, which generates machine code that can be directly executed by the machine/CPU.

- 5. Robust
  - It provides many features that make the program execute reliably in a variety of environments.
  - Java is a strictly typed language. It checks code both at compile time and runtime.
  - Java takes care of all memory management problems with garbage collection.
- 6. Secure
  - Java achieves this protection by confining a Java program to the Java execution environment and not allowing it to access other parts of the computer
- 7. Architecture Neutral
  - Java language and Java Virtual Machine helped in achieving the goal of WORA - Write Once Run Anywhere.
  - Java byte code is interpreted by JIT and convert into CPU machine code/native code.
  - So Java byte code can execute on any CPU architecture (on which JVM is available)
- 8. Portable
  - As java is Architecture Neutral it is portable.
  - Java is portable because of the Java Virtual Machine (JVM).
- 9. High Performance
  - Java performance is high because of the use of bytecode.
  - The bytecode was used so that it can be efficiently translated into native machine code by JIT compiler (in JVM).
- 10. Multithreaded
  - Multithreaded Programs handled multiple tasks simultaneously (within a process)
  - Java supports multi-process/thread communication and synchronization.
  - When Java application executes 2 threads are started
    - 1. main thread
    - 2. garbage collector thread.
- 11. Dynamic
  - Java is capable of linking in new class libraries, methods, and objects.
  - Java classes has run-time type information that is used to verify and resolve accesses to objects/members at runtime.

Agenda

- Exception Handling
  - Exceptions
  - Errors
  - Exception Chaning
  - Custom Exceptions
- ~~Date/LocalDate/Calender~~

Exception Handling

- Exceptions represents runtime problems
- If not handled in the current method it is sent back to the calling method.
- To perform exception handling java have provided below keywords
  - try
  - catch
  - throw
  - throws
  - finally
- Java operators/APIs throw pre-defined exception if runtime problem occurs.
- Exceptions need to handled otherwise the it terminates the program by throwing that exception.
- we use try catch block to handle the exception. Inside try block keep all the method calls that generate exception and handle the exception inside catch block
- 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.
- We can handle multiple exceptions in a single catch block.
- when exceptions are generated if we dont to handle it program terminates,however the resources that are in used should be closed.
- the resources can be closed in finally block that can be used with a try block.
- if the classes have implemented AutoCloseable interface we can also use try with resource with the try block.
- when we use try block then,it should atleast have
  - 1. a catch block
  - 2. a finally block
  - 3. try with resource
- Java have divided the exceptions into two categories
  - 1. Error

    - 2. Exception

- java.lang.Throwable is the super class of all the Errors and Exceptions

```
- Throwable (Super class of all Errors and Exceptions)
    - Error
        - VirtualMachineError
            - OutOfMemoryError
            - StackOverflowError
        - IOError
    - Exception (Checked Exception - Checked at compile time,forced to handle)
        - CloneNotSupportedException
        - IOException
        - InterruptedException
        - RuntimeException (Unchecked Exception - Not Checked By Compiler)
            - ArithmeticException
            - ClassCastException
            - IndexOutOfBoundsException
                - ArrayIndexOutOfBoundsException
                - StringIndexOutOfBoundsException
            - NegativeArraySizeException
            - NoSuchElementException
                - InputMismatchException
            - NullPointerException
```

Errors

- Errors are generated due to runtime environment.
- It can be due to problems in RAM/JVM for memory management or like crashing of harddisk, etc.
- We cannot recover from such errors in our program and hence such errors should not be handled.
- we can write a try catch block to handle such errors but it is recommended not to handle such errors.

Exceptions

- Exception class and all its sub classes except Runtime exception class are all Checked Exception
- Runtime Exception and all its sub classes all unchecked exceptions
- Checked exceptions are mandatory to handle.

Exception Handling Keywords

- 1. try
    - Code where runtime problems may arise should be written in try block.
- 2. catch
    - 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)
- 3. finally
    - Resources are closed in finally block.
    - Executed irrespective of exception occurred or not.

    - finally block not executed if JVM exits (System.exit()).
- 4. "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.
- 5. throws
    - 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.
    - Sub-class overridden method can throw same or subset of exception from super-class method.

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.

```
public static void getEmployees() throws SQLException {
        // logic to get all the employees from database
        boolean connection = false;
        if (connection) {
            // fetch data
            boolean data = false;
            if (data)
                System.out.println(data);
            else
                throw new SQLException("Data not found");
        } else
            throw new SQLException("failed", new SocketException("Connection
rejected"));
    }
```

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.

Agenda

- String
- String Pool
- StringBuffer
- StringBuilder
- String Tokenizer
- Enum
- ~~clone()~~
- ~~Date/LocalDate/Calendar~~

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 Class Object

- java.lang.String is class and strings in java are objects.
- String constants/literals are stored in string pool.
- String objects created using "new" operator are allocated on heap.
- In java, String is immutable. If try to modify, it creates a new String object on heap.

```
String name = "sunbeam"; // goes in string pool

String name2 = new String("Sunbeam"); // goes on heap
```

- 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.

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

String Tokenizer

- Used to break a string into multiple tokens - like split() method.
- Methods of java.util.StringTokenizer
    - boolean hasMoreTokens()
    - String nextToken()
    - String nextToken(String delim)

Enum

- In C enums were internally integers
- In java, It is a keyword added in java 5 and enums are object in java.
- used to make constants for code readability
- mostly used for switch cases
- In java, enums cannot be declared locally (within a method).
- The declared enum is converted into enum class.
- 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 enums constants can be used in switch-case and can also be compared using == operator.
- The enum may have fields and methods.

```
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);
    // ...
}
```

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

// generated enum code
final class ArithmeticOperations extends Enum {
```

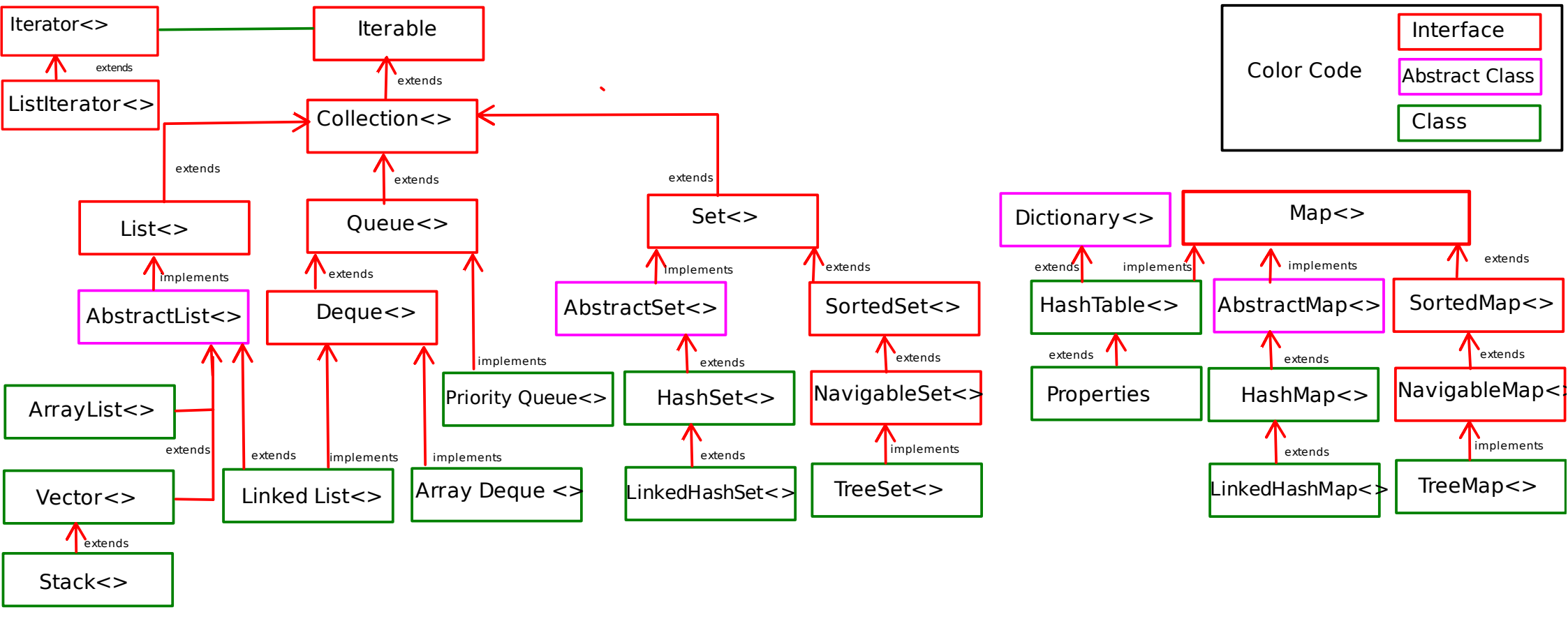
```
private ArithmeticOperations(String name, int ordinal) {
    super(name, ordinal); // invoke sole constructor Enum(String,int);
}

public static ArithmeticOperations[] values() {
    return (ArithmeticOperations[])$VALUES.clone();
}

public static ArithmeticOperations valueOf(String s) {
    return (ArithmeticOperations)Enum.valueOf(ArithmeticOperations,s);
}


  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
      });
  }
}
```



Rohan Paramane

Agenda

- Generics
- Generic class
- Generic method
- Generic Limitations
- Generic Interfaces
    - Comparable
    - ~~Comparator~~

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
    1. using java.lang.Object class -- Non typesafe
    2. using Generics -- Typesafe

1.Generics using Object class

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

```
Box b1 = new Box();
b1.set("Sunbeam");
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);
```

2.Generics using Generics

- Added in Java 5.0.
- Similar to templates in C++.
- We can implement
    1. Generic classes
    2. Generic methods
    3. 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;
  s
  public void set(TYPE obj) {
    this.obj = obj;
  }
  public TYPE get() {
    return this.obj;
  }
}
```

```
Box<String> b1 = new Box<String>();
b1.set("Sunbeam");
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

Box<> b3 = new Box<>(); // error -- type must be given while creating generic
class reference, as reference cannot be auto-detected

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

Box b5 = new Box(); // okay -- internally considered Object type -- compiler
warning "raw types"

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

```
class Box<T extends Number>{
    private T obj;

    public T getObj() {
        return obj;
    }

    public void setObj(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`.
- Remember unbounded work for class references and not for class types.

```
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<Float> fb = new Box<Float>(200.5);
printBox(fb); // okay
```

- Here the upper bound is set (to Number) that means all the classes that inherits Number are allowed

Lower bounded generic types

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

```
public static void printBox(Box<? super Integer> 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<Float> fb = new Box<Float>(200.5f);
printBox(fb); // error
Box<Number> nb = new Box<Number>(null);
printBox(nb); // okay
```

- Here the lower bound is set (to Integer) that means all the classes that are super classes of that lower bound class are allowed.

Generic Methods

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

```
    // Not Type-safe
//  public static void printArray(Object[] arr) {
//      for (Object element : arr) {
//          System.out.println(element);
//      }
//  }

    // Type-safe
    public static <Type> void printArray(Type[] arr) {
        for (Type element : arr) {
```

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

    public static void main(String[] args) {
        String[] arr = { "Rohan", "Nilesh", "Amit" };
        printArray(arr);

        Integer[] arr2 = { 10, 20, 30, 40 };
        Program01.<Integer>printArray(arr2);

        Double[] arr3 = { 10.11, 20.12, 30.13 };
        // printArray(arr3); // type is inferred
        // Program01.<Integer>printArray(arr3); // compiler error
        Program01.<Double>printArray(arr3);// OK
}
```

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 instanceof with generic Type params.

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

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 and Box 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.

Generic Interfaces

- Interface is standard/specification.
- comparable is a predefined interface in java

```
// Comparable is pre-defined interface which was 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.

- Has single abstract method int compareTo(T other);
- In java.lang package.
- Used by various methods like Arrays.sort(Object[]), ...
- It does the comparision for the natural ordering

Assignment

- Cretae a class Student with fields rollno,name and marks. Provide natural ordering on the basis of marks in desc order. Cretae an array of 5 stdents and sort them on the basis of marks in descending order.

- Cretae a class Product with fields id,name,category and price. Provide natural ordering on the basis of category. Cretae an array of 10 products and sort them on the basis of category.

Agenda

- Comparator
- Collection FrameWork
- Traversal
- FailSafe and FailFast Iterator
- ~~List~~

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).
- Collection is available in java.util package.
- Java collection framework provides

1. Interfaces -- defines standard methods for the collections.
2. Implementations -- classes that implements various data stuctures.
3. 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

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)

Iterable interface

- To traverse any collection it provides an Iterator.
- Enable use of for-each loop.
- In java.lang package
- Iterable yeilds an iterator
- Methods
    - Iterator iterator()
    - default Spliterator spliterator()
    - default void forEach(Consumer<? super T> action)

Iterator (Demo01)

- Part of collection framework (1.2)
- Methods
    - boolean hasNext()
    - E next()
    - void remove()

Collections class- Since Java 1.0

- 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

1. 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));
```

2. Collections class

- Helper class that contains all static methods.

- We never create object of "Collections" class.

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

Fail-fast vs Fail-safe Iterator

- If state of collection is modified (add/remove operation other than iterator methods) i.e structural change while traversing a collection using iterator and iterator methods fails (with ConcurrentModificationException), then iterator is said to be Fail-fast.
- The collections from java.util package have fail-fast iterators
- 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.
- The collections from java.util.concurrent package have fail-safe iterators.
- e.g. Iterators from CopyOnWriteArrayList, ...
- If any changes are done in the collection using these iterators then the changes may not be reflected using the same iterator however by creating the new iterator we can get the changes displayed.

Traversal

1. Using Iterator

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

2. Using for-each loop

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

// gets converted into Iterator traversal internally

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

3. using for loop

```
for(int index=0; index<list.size(); index++) {
    Integer i = list.get(index);
        System.out.println(i);
}
```

Assignment

- Create a student class with fields rollno,name,marks and course. provide natural ordering of elements on rollno. Create an array of 10 students and Write a menu driven code that has below menus.

1. Add Student
2. display all students sorted on rollno
3. display all students sorted on name
4. display all students sorted on marks in desc order
5. display all students sorted on course

Agenda

- List
- Queue
- Set
- Map
- hashcode()
- Date/Calenendr/LocalDate

List Interface

- Ordered/sequential collection.
- Implementations: ArrayList, Vector, Stack, LinkedList, etc.
- 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 enables searching in the list
- Abstract methods
    - void add(int index, E element)
    - String toString()
    - E get(int index)
    - E set(int index, E element)
    - int indexOf(Object o)
    - int lastIndexOf(Object o)
    - E remove(int index)
    - boolean addAll(int index, Collection<? extends E> c)
    - 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 contains(), indexOf(), lastIndexOf().

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<>.

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);
}
```

- Enumeration behaves similar to fail-safe iterator.
- Since Java 1.0
- Methods
  - boolean hasMoreElements()
  - E nextElement()
- only useful when traversing with vector

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<>.

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<>.

Stack

- It is inherited from vector class.
- Generally used to have only the stack operations like push, pop and peek opertaions.
- It is recommended to use the Deque from the queue collection.
- It is synchronized and hence gives low performanance.

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).
- Difference between these methods is first 3 methods throws exception however next 3 methods do not throw exception if operation fails.

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.
- Time Complexity to add and remove is O(1)

LinkedList class

- Internally LinkedList is doubly linked list.
- Time Complexity to add and remove is O(1)

PriorityQueue class

- Internally PriorityQueue is a "binary heap" (Array implementation of binary Tree) 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
- Elements are duplicated in Hashset even if equals() is overriden.
- Its because the hashset dosent compare elements only on the basis of equals().
- Hashset considers elements equal if and only if their hashcode() is same and calling equals() to compare them return true.

LinkedHashSet class

- Ordered set (preserves order of insertion)
- Elements must implement equals() and hashCode()
- Slower than HashSet
- Elements are duplicated in LinkedHashset even if equals() is overriden.
- Its because the LinkedHashset dosent compare elements only on the basis of equals().
- LinkedHashset considers elements equal if and only if their hashcode() is same and calling equals() to compare them return true.

SortedSet interface

- Use natural ordering or Comparator to keep elements 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).
- Load factor = Number of entries / Number of slots.
- Multiple keys can compete for the same slot which can cause the collision

- To avoid the collision two techniques are used

1. Open Adderessing
2. Seperate Chaining

- In Seperate Chaning mechanism to avoid the collision Key-value entries are stored in the same bucket depending on hash code of the "key".
- In java we have readymade/ built-in hashtables

1. HashMap
2. LinkedHashMap
3. TreeMap
4. HashTable (Legacy)
5. Properties (Legacy)

- Here we neeed to calculate the hash value of the key using hash function(Override hashcode method).
- The slot in the table is calculated internaly by slot = key.hashcode()%size
- 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).

Similarity between Set and Map

- Set in internally using map implementation where it have all the values as null.

- In set the the elements are stored as keys and the corresponsing values are null.
- HashSet = HashMap<K,null>
- LinkedHashSet = LinkedHashMap<K,null>
- TreeSet = TreeMap<K,null>
- in set duplicate elements are not allowed, in map duplicate keys are not allowed
- For HashSet,Hashmap, LinkedHashSet, LinkedHashMap duplication is based on equals() and hashcode() of key
- For TreeSet and TreeMap the duplication is based on comparable of K or Comparator of K given in constructor

Date/ LocalDate/ Calender

- Date and Calender class are in java.util package
- The class Date represents a specific instant in time, with millisecond precision.
- the formatting and parsing of date strings were not standardized it is not recommended to use
- As of JDK 1.1, the Calendar class should be used to convert between dates and time fields and the DateFormat class should be used to format and parse date strings.
- LocalDate is in java.time Package
- It is immutable and threadsafe class.

Java DateTime APIs

- DateTime APIs till Java 7

```
// java.util.Date
Date d = new Date();
System.out.println("Timestamp: " + d.getTime());
// number of milliseconds since 1-1-1970 00:00.
SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy");
System.out.println("Date: " + sdf.format(d));

// java.util.Date
String str = "28-09-1983";
SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy");
Date d = sdf.parse(str);
System.out.println(d.toString());

// java.util.Calendar
Calendar c = Calendar.getInstance();
System.out.println(c.toString());
System.out.println("Current Year: " + calendar.get(Calendar.YEAR));
System.out.println("Current Month: " + calendar.get(Calendar.MONTH));
System.out.println("Current Date: " + calendar.get(Calendar.DATE));
```

- Limitations of existing DateTime APIs
    - Thread safety
    - API design and ease of understanding
    - ZonedDate and Time

- Most commonly used java 8 onwards new classes are LocalDate, LocalTime and LocalDateTime.
- LocalDate

```
LocalDate localDate = LocalDate.now();
LocalDate tomorrow = localDate.plusDays(1);
DayOfWeek day = tomorrow.getDayOfWeek();
int date = tomorrow.getDayOfMonth();
System.out.println("Date: " +
tomorrow.format(DateTimeFormatter.ofPattern("dd-MMM-yyyy")));

//LocalDate date = LocalDate.of(1983, 09, 28);
LocalDate date = LocalDate.parse("1983-09-28");
System.out.println("Is Leap Year: " + date.isLeapYear());
```

- LocalTime

```
LocalTime now = LocalTime.now();
LocalTime nextHour = now.plus(1, ChronoUnit.HOURS);
System.out.println("Hour: " + nextHour.getHour());
System.out.println("Time: " +
nextHour.format(DateTimeFormatter.ofPattern("HH:mm")));
```

- LocalDateTime

```
LocalDateTime now = LocalDateTime.now();
LocalDateTime dt = LocalDateTime.parse("2000-01-30T06:30:00");
dt.minusHours(2);
System.out.println(dt.toString());
```

Clone method

- The clone() method is used to create a copy of an object in Java. - It's defined in the java.lang.Object class and is inherited by all classes in Java.
- It returns a shallow copy of the object on which it's called.

```
protected Object clone() throws CloneNotSupportedException
```

- This means that it creates a new object with the same field values as the original object, but the fields themselves are not cloned.
- If the fields are reference types, the new object will refer to the same objects as the original object.
- In order to use the clone() method, the class of the object being cloned must implement the Cloneable interface.
- This interface acts as a marker interface, indicating to the JVM that the class supports cloning.

- It's recommended to override the clone() method in the class being cloned to provide proper cloning behavior.
- The overridden method should call super.clone() to create the initial shallow copy, and then perform any necessary deep copying if required.
- The clone() method throws a CloneNotSupportedException if the class being cloned does not implement Cloneable, or if it's overridden to throw the exception explicitly.

Agenda

- JDBC

JDBC

- RDBMS understand SQL language only.
- JDBC driver converts Java requests in database understandable form and database response in Java understandable form.
- JDBC drivers are of 4 types

1. Type I - Jdbc Odbc Bridge driver
    - ODBC is standard of connecting to RDBMS (by Microsoft).
    - Needs to create a DSN (data source name) from the control panel.
    - From Java application JDBC Type I driver can communicate with that ODBC driver (DSN).
    - The driver class: sun.jdbc.odbc.JdbcOdbcDriver -- built-in in Java.
    - database url: jdbc:odbc:dsn
    - Advantages:
    - Can be easily connected to any database.
    - Disadvantages:
    - Slower execution (Multiple layers).
    - The ODBC driver needs to be installed on the client machine.
2. Type II - Partial Java/Native driver
    - Partially implemented in Java and partially in C/C++. Java code calls C/C++ methods via JNI.
    - Different driver for different RDBMS. Example: Oracle OCI driver.
    - Advantages:
    - Faster execution
    - Disadvantages:
        - Partially in Java (not truely portable)
        - Different driver for Different RDBMS
3. Type III - Middleware/Network driver
    - Driver communicate with a middleware that in turn talks to RDBMS.
    - Example: WebLogic RMI Driver
    - Advantages:
        - Client coding is easier (most task done by middleware)
    - Disadvantages:
        - Maintaining middleware is costlier
        - Middleware specific to database
4. Type IV
    - Database specific driver written completely in Java.
    - Fully portable.
    - Most commonly used.

- Example: Oracle thin driver, MySQL Connector/J, ...

MySQL Programming Steps

- step 0: Add JDBC driver into project/classpath. In Eclipse, project -> right click -> properties -> java build path -> libraries -> Add external jars -> select mysql driver jar.
- step 1: Load and register JDBC driver class. These drivers are auto-registered when loaded first time in JVM. This step is optional in Java SE applications from JDBC 4 spec.

```
Class.forName("com.mysql.cj.jdbc.Driver");
// for Oracle: Use driver class oracle.jdbc.driver.OracleDriver
```

- step 2: Create JDBC connection using helper class DriverManager.

```
// db url = jdbc:dbname://db-server:port/database
Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/classwork", "root",
"manager");
// for Oracle: jdbc:oracle:thin:@localhost:1521:sid
```

- step 3: Create the statement.

```
Statement stmt = con.createStatement();
```

- step 4: Execute the SQL query using the statement and process the result.

```
String sql = "non-select query";
int count = stmt.executeUpdate(sql); // returns number of rows affected
OR
String sql = "select query";
ResultSet rs = stmt.executeQuery(sql);
while(rs.next()) // fetch next row from db(return false when all rows completed)
{
  x = rs.getInt("col1");
  // get first column from the current row
  y = rs.getString("col2");
  // get second column from the current row
  z = rs.getDouble("col3");
  // get third column from the current row
  // process/print the result
}
rs.close();
```

- step 5: Close statement and connection.

```
con.close();
stmt.close();
```

MySQL Driver Download

https://mvnrepository.com/artifact/com.mysql/mysql-connector-j/8.1.0

SQL Injection

- Building queries by string concatenation is inefficient as well as insecure.
- Example:

```
dno = sc.nextLine();
sql = "SELECT * FROM emp WHERE deptno="+dno;
```

- If user input "10", then effective SQL will be "SELECT _ FROM emp WHERE deptno=10". This will select all emps of deptno 10 from the RDBMS.
- If user input "10 OR 1", then effective SQL will be "SELECT _ FROM emp WHERE deptno=10 OR 1". Here "1" represent true condition and it will select all rows from the RDBMS.
- In Java, it is recommeded NOT to use "Statement" and building SQL by string concatenation. Instead use PreparedStatement.

PreparedStatement

- PreparedStatement represents parameterized queries.

```
String sql = "SELECT * FROM students WHERE name=?";
PreparedStatement stmt = con.prepareStatement(sql);

System.out.print("Enter name to find: ");
String name = sc.next();

stmt.setString(1, name);
ResultSet rs = stmt.executeQuery();

while(rs.next()) {
  int roll = rs.getInt("roll");
  String name = rs.getString("name");
  double marks = rs.getDouble("marks");
  System.out.printf("%d, %s, %.2f\n", roll, name, marks);
}
```

- The same PreparedStatement can be used for executing multiple queries. There is no syntax checking repeated. This improves the performance.

JDBC concepts

java.sql.Driver

- Implemented in JDBC drivers.
- MySQL: com.mysql.cj.jdbc.Driver
- Oracle: oracle.jdbc.OracleDriver
- Postgres: org.postgresql.Driver
- Driver needs to be registered with DriverManager before use.
- When driver class is loaded, it is auto-registered (Class.forName()).
- Driver object is responsible for establishing database "Connection" with its connect() method.
- This method is called from DriverManager.getConnection().

java.sql.Connection

- Connection object represents database socket connection.
- All communication with db is carried out via this connection.
- Connection functionalities:
  - Connection object creates a Statement.
  - Transaction management.

java.sql.Statement

- Represents SQL statement/query.
- To execute the query and collect the result.

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(selectQuery);
int count = stmt.executeUpdate(nonSelectQuery);
```

- Since query built using string concatenation, it may cause SQL injection.

java.sql.PreparedStatement

- Inherited from java.sql.Statement.
- Represents parameterized SQL statement/query.
- The query parameters (?) should be set before executing the query.
- Same query can be executed multiple times, with different parameter values.
- This speed up execution, because query syntax checking is done only once.

```
PreparedStatement stmt = con.prepareStatement(query);
stmt.setInt(1, intValue);
stmt.setString(2, stringValue);
stmt.setDouble(3, doubleValue);
stmt.setDate(4, dateObject); // java.sql.Date
stmt.setTimestamp(5, timestampObject); // java.sql.Timestamp

ResultSet rs = stmt.executeQuery();
```

```
// OR
int count = stmt.executeUpdate();
```

java.sql.ResultSet

ResultSet represents result of SELECT query. The result may have one/more rows and one/more columns. Can access only the columns fetched from database in SELECT query (projection).

```
// SELECT id, quote, created_at FROM quotes
ResultSet rs = stmt.executeQuery();
while(rs.next()) {
  int id = rs.getInt("id");
  String quote = rs.getString("quote");
  Timestamp createdAt = rs.getTimestamp("created_at"); // java.sql.Timestamp
// ...
}
// SELECT id, quote, created_at FROM quotes
ResultSet rs = stmt.executeQuery();
while(rs.next()) {
  int id = rs.getInt(1);
  String quote = rs.getString(2);
  Timestamp createdAt = rs.getTimestamp(3); // java.sql.Timestamp
// ...
}
```

DAO class

- In enterprise applications, there are multiple tables and frequent data transfer from database is needed.
- Instead of writing a JDBC code in multiple Java files of the application (as and when needed), it is good practice to keep all the JDBC code in a centralized place -- in a single application layer.
- DAO (Data Access Object) class is standard way to implement all CRUD operations specific to a table. It is advised to create different DAO for different table.
- DAO classes makes application more readable/maintainable.
- Example 1:

```
class StudentDao implements AutoClosable {
private Connection con;
public StudentDao() throws Exception {
  con = DriverManager.getConnection(DbUtil.DB_URL, DbUtil.DB_USER,
DbUtil.DB_PASSWORD);
}
public void close() {
  try{
    if(con != null)
    con.close();
  } catch(Exception ex) {
  }
```

```
}
public int update(Student s) throws Exception {
  int count = 0;
  String sql = "UPDATE students SET name=?, marks=? WHERE roll=?"
  try(PreparedStatement stmt = con.prepareStatement(sql)) {
  // optionally you may create PreparedStatement in constructor (as implemented)
  stmt.setString(1, s.getName());
  stmt.setDouble(2, s.getMarks());
  stmt.setInt(3, s.getRoll());
  count = stmt.executeUpdate();
}
return count;
}
}

// in main()
try(StudentDao dao = new StudentDao()) {
  System.out.print("Enter roll to be updated: ");
  int roll = sc.nextInt();
  System.out.print("Enter new name: ");
  String name = sc.next();
  System.out.print("Enter new marks: ");
  double marks = sc.next();
  Student s = new Student(roll, name, marks);
  int cnt = dao.update(s);
  System.out.println("Rows updated: " + cnt);
} // dao.close()
catch(Exception ex) {
  ex.printStackTrace();
}
```

- Example 2:

```
// POJO (Entity)
class Emp {
private int empno;
private String ename;
private Date hire;
// ...
}

class DbUtil {
public static final String DB_DRIVER = "com.mysql.cj.jdbc.Driver";
public static final String DB_URL = "jdbc:mysql://localhost:3306/test";
public static final String DB_USER = "root";
public static final String DB_PASSSWD = "root";
static {
  try {
    Class.forName(DB_DRIVER);
} catch (ClassNotFoundException e) {
    e.printStackTrace();
    System.exit(0);
```

```
}
}
public static Connection getConnection() throws Exception {
  return DriverManager.getConnection(DB_URL, DB_USER, DB_PASSSWD);
}
}

class EmpDao implements AutoClosable {
private Connection con;
public EmpDao() throws Exception {
  con = DbUtil.getConnection();
}
public void close() {
  try {
    if(con != null)
    con.close();
} catch(Exception ex) {
  ex.printStackTrace();
}
}
public int update(Emp e) throws Exception {
  String sql = "UPDATE emp SET ename=?, hire=? WHERE id=?";
  try(PreparedStatement stmt = con.prepareStatement(sql)) {
    stmt.setString(1, e.getEname());
    java.util.Date uDate = e.getHire();
    java.sql.Date sDate = new java.sql.Date(uDate.getTime());
    stmt.setDate(2, sDate);
    stmt.setInt(3, e.getEmpno());
    int cnt = stmt.executeUpdate();
    return cnt;
} // stmt.close();
}
// ...
}

// in main()
try(EmpDao dao = new EmpDao()) {
  Emp e = new Emp();
// input emp data from end user (Scanner)
/*
String dateStr = sc.next(); // dd-MM-yyyy
SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy");
java.util.Date uDate = sdf.parse(dateStr);
e.setHire(uDate);
*/
  int cnt = dao.update(e);
  System.out.println("Emps updated: " + cnt);
} // dao.close();
catch(Exception ex) {
ex.printStackTrace();
}
```

- Example 3 (using the POJO and DBUtil same as Example 2)

```
class EmpDao implements AutoClosable {
private Connection con;
private PreparedStatement stmtFindById;
// ...
public EmpDao() throws Exception {
  con = DbUtil.getConnection();
  String sql = "SELECT * FROM emp WHERE empno=?";
  stmtFindById = con.prepareStatement(sql);
// ...
}
public void close() {
  try {
  // ...
  if(stmtFindById != null)
  stmtFindById.close();
  if(con != null)
  con.close();
} catch(Exception ex) {
  ex.printStackTrace();
}
}
public Emp findById(int empno) throws Exception {
  stmtFindById.setInt(1, empno);
  try(ResultSet rs = stmtFindById.executeQuery()) {
  if(rs.next()) {
  int empno = rs.getInt("empno");
  String ename = rs.getString("ename");
  java.sql.Date sDate = rs.getDate("hire");
  // ...
  java.util.Date uDate = new java.util.Date( sDate.getTime() );
  Emp e = new Emp(empno, ename, uDate);
  return e;
}
} // rs.close();
return null;
}
}
```
```java
// in main()
try(EmpDao dao = new EmpDao()) {
  System.out.print("Enter empno to find: ");
  id = sc.nextInt();
  e = dao.findById(id);
  System.out.println("Found: " + e);
  System.out.print("Enter empno to find: ");
  id = sc.nextInt();
  e = dao.findById(id);
  System.out.println("Found: " + e);
  System.out.print("Enter empno to find: ");
  id = sc.nextInt();
  e = dao.findById(id);
  System.out.println("Found: " + e);
```

```
}
catch(Exception ex) {
  ex.printStackTrace();
}
```

Agenda

- JDBC
- Annotations
- Reflection

Call Stored Procedure using JDBC (without OUT parameters)

- Stored Procedure - Increment votes of candidate with given id

```
DELIMITER //
CREATE PROCEDURE sp_incrementvotes(IN p_id INT)
BEGIN
UPDATE candidates SET votes=votes+1 WHERE id=p_id;
END;
//
DELIMITER ;

CALL sp_incrementvotes(10);
```

- JDBC use CallableStatement interface to invoke the stored procedures.
- CallableStatement interface is extended from PreparedStatement interface.
- Steps to call Stored procedure are same as PreparedStatement.
    - Create connection.
    - Create CallableStatement using con.prepareCall("CALL ...").
    - Set IN parameters using stmt.setXYZ(...);
    - Execute the procedure using stmt.executeQuery() or stmt.executeUpdate().
    - Close statement & connection.
- To invoke stored procedure, in general stmt.execute() is called. This method returns true, if it is returning ResultSet (i.e.multi-row result). Otherwise it returns false, if it is returning update/affected rows count.

```
boolean isResultSet = stmt.execute();
if(isResultSet) {
ResultSet rs = stmt.getResultSet();
// process the ResultSet
}
else {
int count = stmt.getUpdateCount();
// process the count
}
```

Call Stored Procedure using JDBC (with OUT parameters)

- Stored Procedure - Get votes for given party -- using OUT parameters.

```
DELIMITER //
CREATE PROCEDURE sp_getpartyvotes(IN p_party CHAR(40), OUT p_votes INT)
BEGIN
SELECT SUM(votes) INTO p_votes FROM candidates WHERE party=p_party;
END;
//
DELIMITER ;

CALL sp_getpartyvotes('BJP', @votes);
SELECT @votes;
```

- Steps to call Stored procedure with out params.
    - Create connection.
    - Create CallableStatement using con.prepareCall("CALL ...").
    - Set IN parameters using stmt.setXYZ(...) and register out parameters using stmt.registerOutParam(...).
    - Execute the procedure using stmt.execute().
    - Get values of out params using stmt.getXYZ(paramNumber).
    - Close statement & connection.

Transaction Management

- RDBMS Transactions
- Transaction is set of DML operations to be executed as a single unit. Either all queries in tx should be successful or all should be discarded.
- The transactions must be atomic. They should never be partial.

```
CREATE TABLE accounts(id INT, type CHAR(30), balance DOUBLE);
INSERT INTO accounts VALUES (1, 'Saving', 30000.00);
INSERT INTO accounts VALUES (2, 'Saving', 2000.00);
INSERT INTO accounts VALUES (3, 'Saving', 10000.00);
SELECT * FROM accounts;
START TRANSACTION;
--SET @@autocommit=0;
UPDATE accounts SET balance=balance-3000 WHERE id=1;
UPDATE accounts SET balance=balance+3000 WHERE id=2;
SELECT * FROM accounts;
COMMIT;
-- OR
ROLLBACK;
```

- JDBC transactions (Logical code)

```
try(Connection con = DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD)) {
con.setAutoCommit(false); // start transaction

String sql = "UPDATE accounts SET balance=balance+? WHERE id=?";
```

```
try(PreparedStatement stmt = con.prepareStatement(sql)) {
  stmt.setDouble(1, -3000.0); // amount=3000.0
  stmt.setInt(2, 1); // accid = 1
  cnt1 = stmt.executeUpdate();
  stmt.setDouble(1, +3000.0); // amount=3000.0
  stmt.setInt(2, 2); // accid = 2
  cnt2 = stmt.executeUpdate();
  if(cnt1 == 0 || cnt2 == 0)
    throw new RuntimeException("Account Not Found");
}
con.commit(); // commit transaction
}
catch(Exception e) {
  e.printStackTrace();
  con.rollback(); // rollback transaction
}
```

ResultSet

- ResultSet types

1. TYPE_FORWARD_ONLY -- default type
    - next() -- fetch the next row from the db and return true. If no row is available, return false.
2. TYPE_SCROLL_INSENSITIVE
    - next() -- fetch the next row from the db and return true. If no row is available, return false.
    - previous() -- fetch the previous row from the db and return true. If no row is available, return false.
    - absolute(rownum) -- fetch the row with given row number and return true. If no row is available (of that number), return false.
    - relative(rownum) -- fetch the row of next rownum from current position and return true. If no row is available (of that number), return false.
    - first(), last() -- fetch the first/last row from db. beforeFirst(), afterLast() -- set ResultSet to respective positions.
    - INSENSITIVE -- After taking ResultSet if any changes are done in database, those will NOT be available/accessible using ResultSet object.
    - Such ResultSet is INSENSITIVE to the changes (done externally).
3. TYPE_SCROLL_SENSITIVE
    - SCROLL -- same as above.
    - SENSITIVE -- After taking ResultSet if any changes are done in database, those will be available/accessible using ResultSet object. Such ResultSet is SENSITIVE to the changes (done externally).

ResultSet concurrency

- CONCUR_READ_ONLY -- Using this ResultSet one can only read from db (not DML operations). This is default concurrency.
- CONCUR_UPDATABLE -- Using this ResultSet one can read from db as well as perform INSERT, UPDATE and DELETE operations on database.

```
String sql = "SELECT roll, name, marks FROM students";
stmt = con.prepareStatement(sql, ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
rs = stmt.executeQuery();
rs.absolute(2); // moves the cursor to the 2nd row of rs
rs.updateString("name", "Bill"); // updates the 'name' column of row 2 to be Bill

rs.updateDouble("marks", 76.32); // updates the 'marks' column of row 2 to be
76.32
rs.updateRow(); // updates the row in the database

rs.moveToInsertRow(); // moves cursor to the insert row -- is a blank row
rs.updateInt(1, 9); // updates the 1st column (roll) to be 9
rs.updateString(2, "AINSWORTH"); // updates the 2nd column (name) of to be
AINSWORTH
rs.updateDouble(3, 76.23); // updates the 3rd column (marks) to true 76.23
rs.insertRow(); // inserts the row in the database
rs.moveToCurrentRow();

rs.absolute(2); // moves the cursor to the 2nd row of rs
rs.deleteRow(); // deletes the current row from the db
```

Reflection

- It is a technique to read the metadata and work with that data.
- .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
```

Invoking method dynamically

```
```Java
public class Middleware {
    public static Object invoke(String className, String methodName, Class[]
methodParamTypes, Object[] methodArgs) throws Exception {
        // load the given class
        Class c = Class.forName(className);
        // create object of that class
        Object obj = c.newInstance(); // also invokes param-less constructor
        // find the desired method
        Method method = c.getDeclaredMethod(methodName, methodParamTypes);
        // allow to access the method (irrespective of its access specifier)
        method.setAccessible(true);
        // invoke the method on the created object with given args & collect the
result
        Object result = method.invoke(obj, methodArgs);
        // return the results
        return result;
    }
}
```
```Java
// invoking method statically
Date d = new Date();
String result = d.toString();
```
```Java
// invoking method dynamically
String result = Middleware.invoke("java.util.Date", "toString", null, null);
```
```

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 )
@Taget({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)
@Taget({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="Borle", 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 {
    // ...
}
```

Annotation processing (using Reflection)

```
Annotation[] anns = MyClass.class.getDeclaredAnnotations();
for (Annotation ann : anns) {
    System.out.println(ann.toString());
    if(ann instanceof Developer) {
        Developer devAnn = (Developer) ann;
        System.out.println(" - Name: " + devAnn.firstName() + " " + devAnn.
lastName());
        System.out.println(" - Company: " + devAnn.company());
        System.out.println(" - Role: " + devAnn.value());
    }
}
System.out.println();

Field field = MyClass.class.getDeclaredField("myField");
anns = field.getAnnotations() ;
for (Annotation ann : anns)
```

```
    System.out.println(ann.toString());
System.out.println();

//anns = YourClass.class.getDeclaredAnnotations();
anns = YourClass.class.getAnnotations();
for (Annotation ann : anns)
    System.out.println(ann.toString());
System.out.println();
```

Agenda

- File IO

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.
- Two types of APIs are available file handling
    - FileSystem API -- Accessing/Manipulating Metadata
    - File IO API -- Accessing/Manipulating Contents/Data

File

- File is a collection of data and information on a storage device.
- File = Data + Metadata
- collection of data/info on storage disk
- data = contents
- metadata = Information

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.
- Java IO Streams are completly different from java.util.Stream. No relation between them
- Stream generally determines flow of data
- 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
- Stream is abstraction of data source/sink.
    - Data source -- InputStream(Byte Stream) or Reader(Char Stream)
    - Data sink -- OutputStream(Byte Stream) or Writer(Char Stream)
- All these streams are AutoCloseable (so can be used with try-with-resource construct)

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.
- It is used only to write the formatted data in to the file.

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.

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()
    - ...

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

Java NIO vs Java IO

- IO: Stream-oriented
- NIO: Buffer-oriented
- IO: Blocking IO
- NIO: Non-blocking IO

Agenda

- Java 8 Interfaces
  - default Methods
  - Static Methods
  - Functional Interfaces
- Annoymous Inner Classes
- Lambda Expressions
- Stream Programming
- Method references

Java 8 Interface

- 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.

1. 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");
    }
}
```

2. Functional Interfaces

- 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.

Anonymous 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();
    }
});
```

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

1. Intermediate operations: Yields another stream.

- intermediatte operations are again classified as

    1. stateless operation
    - filter(), map(), flatMap(), limit(), skip()
    2. stateful operation
    - sorted(), distinct()

2. Terminal operations: Yields some result.

- reduce()
- forEach()for (Employee e : arr) System.out.println(e);
- collect(), toArray()
- count(), max(), min()
- Stream operations are higher order functions (take functional interfaces as arg).

Java stream characteristics

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

Stream creation

- Collection interface: stream() or parallelStream()

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

- Arrays class: Arrays.stream()

```
  Double arr[] = {1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9};
    Stream<Double> strm =  Arrays.stream(arr);
```

- 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)

Method references

- lambda expression is an short-hand implementation of Single Abstract Method (Functional Interface)
- Method reference is short-hand of lambda-expression
- 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.

Agenda

- Multi-Threading
- Nested Classes
- Local Class

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)

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.

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 --
    - Non-static member class --
    - Local class --

    - Annoymous Inner class --
- 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);
```