Day1

Core Java:

Introduction to Java

Java is an Object Oriented Programming language meant for developing real time business applications

Types of Business Applications:

1. Desktop / Stand Alone - Single User System
2. Network Based - Multi User System (User count is fixed)
3. Web Based - Multi User System
   (User count is not fixed)
   (Accessed using Web Browser)
4. Mobile Based - Accessed using Mobiles, Tablets etc

Java language provides several platforms:

1. Java SE -> Java Standard Edition
2. Java EE -> Java Enterprise Edition
3. Android

OOP: Object Oriented Programming is a set of principles used to develop loosely coupled softwares.

It deals with domain specific entities known as Objects.
Object is an entity having well defined structure

OOP Principles:

Abstraction
- It deals with hiding the complexities
- It identifies key properties and ignores rest

Encapsulation
- It is a separation between abstraction and its implementation

Modularity
- It is a process of breaking the system into small units of work known as modules

Inheritance
- It deals with creating new structure based upon existing structure.
- Therefore it handles 2 things:
- Generalization and Specialization

Polymorphism
- It deals with multiple forms.
- Objects responding to same message in different ways is

known as Polymorphism

A First Java Program

Environment Setup

1. Install JDK (Java Development Kit)
   It is service pack that includes several utilities for:
   Compilation
   Execution
   Documentation
   Archiving
   etc.
2. Any Text Editor or IDE of your choice (Recommended)

All Java source code written inside a file is known as a source file.
It must have .java extension.

Once a source file is created, it needs to be compiled. This is done using a Java compiler "javac.exe"

Once a source code is compiled, it results into creation of one more file, known as CLASS file. This file has .class extension
This file contains a byte code which is a highly optimized set of instructions and it is platform neutral. This makes the language Platform Independent.

While executing this byte code, it is converted into platform specific code.
This is done by Java Interpreter which is JVM (Java Virtual Machine).
To launch the interpreter, "java.exe" is used.

JRE: Java Runtime Environment is a basic environment required to execute a Java Application.

JRE Constituents:

Class Loader:
It loads .class files into memory

Byte Code Verifier:
Checks the internal consistency and validity of the code. Once approved, it is given to JVM

JVM:
It acts as an interpreter. It is highly platform dependent.

Source File Rules / Concepts:

1. If a class is 'public', file name must match the class name.
2. The name of the .class file depends upon the name of the class rather than the name of the source file.
3. It is possible to define multiple classes in a single source file provided not more that one class are public.
4. The number of .class files depends upon the number of classes and not upon the number of source files.

Implementing OOP using Java

Object Oriented Programming mainly deals with 2 things:

1. Class
   It is a template that decides a structure for an object.
2. Object
   It is an instance of a class.

Once a class is created, it is possible to create 'n' number of objects of that class.

Data Types in Java:

Java is a strict typed language. The variables meant for holding a data must be declared of some type.

Java supports wide range of primitive data types:

1. Integers
   byte, short, int, long
2. Fractions
   float, double
3. Character (Single Character)
   char
4. Booleans (YES / NO)
   boolean

Creating Objects:

In Java, objects are always created dynamically using 'new' operator. 'new' operator allocates memory dynamically.

Implementing Encapsulation:

To implement encapsulation, first it is necessary to hide the the data.

This is possible using 'private' access modifier.

When a variable is declared as 'private', it can be accessed only within the class.

To make it accessible, it is necessary to add methods.

Course Management System

It maintains inventory of various courses and it handles information as per the following:

- Course ID (Numeric)
- Course Title (Character)
- Course Duration (Numeric in terms of Hrs)
- Course Provider (Character)
- Course Cost (Fraction)

Handle Abstraction and Encapsulation.

Store and Print information about 2 to 3 courses.

Building Classes in Java

A class may have several declarations:

- Variables
- Methods
- Constructors
- Static Members (Static Variables and Static Methods)
- Static Block

Setting and Getting values of fields:

Ideally a class must provide a functionality to work upon individual fields.

This is especially for changing the value or retrieving the value of a specific field

This is possible by adding a pair of Setters and Getters (Mutators and Accessors)

Setter methods typically follow a convention: void setXxx(field-type)

Getter methods typically follow a convention: field-type getXxx()

'this' reference:

Every member method of a Java class gets a hidden parameter known as 'this' reference.

It is a reference that refers to the current object. It is required to resolve the scope of the variable.

If local variable name conflicts with permanent field name, then local variable gets a priority

---

Implementing Polymorphism: (Reference: Planet Class from package "day2")

Polymorphism involves many forms. In Object oriented world, it is of 2 types:

Static (Compile Time) and Dynamic (Runtime)

Static polymorphism involves methods with same name but different signatures and

It is known as Method Overloading.

A signature is said to be different if one of the following is satisfied:

1. Number of parameters are different
2. Types of parameters are different
3. Sequence of parameters is different

If a corresponding matching signature method is not found, compiler generates error; hence It is known as Static (Compile Time) Polymorphism.
Return types of the methods are not taken into consideration

Constructors: (Reference: Planet Class from package "day2")

Whenever an object is created, it consists of some values which are known as "Default Values". These values depend upon the field type.

E.g.

Integers: 0
Fractions: 0.0
Character: '\u0000' (Java follows UNICODE character set to support I18N)
Boolean: false
Reference (Class Type) : null

In order to customise these default values, Java provides a special member known as Constructor.

A constructor is a special member within a class that has same name as that of class name.
It gets called implicitly as soon as an object is created. It does not return any value therefore it does not have a return type.

Like methods, constructors can be overloaded.

A constructor without any parameter is known as No-Argument constructor.
A constructor with parameters is a known as Parameterized constructor

Default Constructor Vs No Argument Constructor:
When a class is created, for every class, Java itself provides a constructor by default.
That constructor is known as DEFAULT constructor

Whenever any explicit constructor is added within a class that accepts no parameters, it is precisely called as No-Argument constructor

When any constructor is added explicitly within a class, the constructor that is provided the Java language vanishes.

Static Members

Any thing that is declared within a class is called as member of a class

Members are of 2 types:

Non Static

These are the characteristics associated with a specific object
Variables.

A copy is created for every object hence it is also called as Instance Variables
If not declared 'private', have to be accessed always using an Object reference name

Methods

Since they perform object specific task, they must be called always using an Object reference name

They can make use of 'this' keyword
They can make use of static members also

Constructors

Static (Reference: Book Class)

These are the characteristics associated with the whole class rather than with a specific object Variables.
The copy of the variable is created for the whole class and not for any specific object.
Irrespective of the number of objects, a single copy is created.
Therefore a static variable is also called a Class Variable.
If not declared 'private', can be accessed without using Object using the format: ClassName.VariableName
However it can be accessed using Object reference name also but it is not recommended.

Methods
It can be called without Object reference name using the format: ClassName.methodName()
However it can be called using an Object reference name also but it is not recommended.

They cannot make use of 'this' keyword
They can access only static members

*** Constructors cannot be static

***In case of modifiers, order is not important

Static Initialization Block:

A block of code that is invoked as soon as the class is loaded. Useful in scenarios to handle the activities during Class Loading. It can also be used to initialise static variables.

If a class contains a static block along with main() method, static block gets called first. This allows it to perform operations prior to the invocation of main().

Memory Mapping (Reference: MemoryMappingMain)

In Java, Objects are created using the 'new' operator.

The 'new' operator allocates memory dynamically from HEAP area and therefore

objects are always created on HEAP whereas references are created on STACK.

An object may have multiple references but a reference can refer to only one object at a time.

Garbage Collection

It is a background activity implemented by Java Runtime System to perform memory deallocation for the objects which are marked as UNUSED.

The object becomes eligible for Garbage Collection and marked as UNUSED when its reference count becomes 0. It is not garbage collected immediately.

There is a background process known as Garbage Collector that runs periodically and identifies the objects marked as UNUSED and starts de-allocating memory for them.

An object becomes eligible for GC under various circumstances:

1. Shifting the reference from one object to another.

   E.g.

```
Book b1 = new Book();
b1 = new Book();
```

2. Explicitly nullifying the reference.

   E.g.

```
Book b1 = new Book();
b1 = null;
```

3. Creating an object without reference

   E.g.

```
new Book();
```

4. When the object reference goes out of scope

   E.g.

```
static void create(){
        Book b1 = new Book();
}

void main(){
        create();
        //At this stage when control returns from create(), object becomes
        //eligible for GC.
}
```

TODOs: Course Management System

Enhance the CMS as per the following:

1. Make use of Constructors, Getters and Setters for Course Entity.
2. Create a class CourseProcessor that provides a static method and performs following operations:

The method must create several Course objects using Hard Coded values

and return the average of course fees.

Parameter Passing: (Reference: ParameterPassingMain)
Methods can have parameters especially in the form of Primitives and Objects.

In case of primitives, they are passed using the BY VALUE mechanism. A copy of a variable
is created on STACK and it gets passed.

In case of objects, a copy of reference of the object is created on stack rather than the actual object, hence they are passed using BY REFERENCE mechanism.

Working with Arrays: (Ref: ArrayMain)
Array is a set of similar typed elements stored at contiguous memory locations.
Used to handle multiple values at one place.
Every element in the array has an index which starts from 0.

There are 2 ways to declare arrays:

1. Just declaration : Uses '[ ]'
2. Declaration and Initialization : Uses '{ }' : It does not make use of 'new'

Every array in Java is an Object and therefore all arrays are created on HEAP.
All array objects have a property called 'length' that indicates size of the array.

Traversing through the array can be simplified using 'for-each' loop

In Java, it is possible to create an array dynamically.
E.g.

```
int size = 3;
int numbers[] = new int[size];
```

In the above fragment variable size holds hard coded value but it can be accepted at runtime as well.

Array as Argument (Reference: ArrayAsArgumentMain)

Since every array is an object, it is quite possible to send array as an argument to
the method or return array as an argument from the method.

It is also possible to create an array of Custom Object type.
(Reference: ArrayOfObjectsMain)

TODOs:
Implement something similar to BookStore for CMS

Day3:

Associations / Relationships in Java

In Java, relationships are of 2 types:

HAS-A

It deals with one object containing another object or several objects.
This is known as Containment
E.g.

```
class Engine {
        private String typeOfFuel;
        private String power;
        //Remaining code
}
class Car {
        private int id;
        private String make, model;
        private Engine engineDetails;
        //Remaining code
}
```

IS-A

It deals with Parent-Child association. That is inheriting the properties from Base class.
This is known as Inheritance

E.g.
Car and SportsCar
Player and CricketPlayer

Inheritance involves one class extending another class which is expressed using 'extends' keyword.

E.g.

class Player {....}
class CricketPlayer extends Player {...} (Reference: SimpleInheritanceMain)

In Java, inheritance is of 3 types:

1. Simple
2. Multilevel
3. Hierarchical

Java does not support Multiple and Hybrid inheritance
Inheritance in Java is always public

Constructors in Inheritance:

When constructors are defined within inherited classes, their order of invocation is from Bottom to Top whereas the order of execution is from Top to Bottom.

In case of no-argument constructor, a call to the parent class constructor is implicit because of the default mechanism of the 'super()' keyword.

The keyword 'super' is used to give a call to the super class constructor.
'Super' refers to an immediate super class.

Use of 'super' is required especially in case of parameterized constructors.

Since there might be multiple versions of parameterized constructors available in the base class, it is required to specify which constructor is to be invoked.

This is done by giving a call explicitly using 'super'

'Super' , if used, must be the first statement in the constructor.
If not used, parent class uses its no-argument constructor

Dynamic Polymorphism:

This is the 2nd type of polymorphism which is also known as Runtime Polymorphism.

It is achieved through Method Overriding using Inheritance.

The prerequisite for Method Overriding is Inheritance.

Method Overriding: (Reference: MethodOverridingMain)

Methods with the same name, same signature, same return type defined within inherited classes are called as overridden methods.

E.g.

```
class Base {
        int myMethod(String s){ ... }
}
class Derived extends Base {
        int myMethod(String s){ ... }
}
```

Why Overriding?

Overriding is required to modify the behaviour of an existing method to suit the new requirement.

How Dynamic Polymorphism takes place?

In Java, every object is manipulated or handled using some reference.

Every reference has 2 types: **static and dynamic**

In method overriding, dynamic type governs method selection.

When classes represent IS-A relationship, it is possible to declare a reference of a base class referring to an object of derived class.

E.g.

Base b1 = new Derived();

In the above fragment, the static type of 'b1' is Base whereas its dynamic type is Derived.

Hence b1.myMethod(..) results in invocation of myMethod() from Derived class. However, the reverse is not possible.

E.g.

Derived d = new Base() results into error

When an object of derived class is created using a reference of base class, it is not possible to call the methods using that reference which are not present in the base class.

To invoke such methods, an explicit cast is necessary.

Method Overriding can be used to handle 2 activities:

1. Replacement
   The method from derived class replaces the method from base class
2. Extension (Reference: MethodOverridingExtensionMain)

The method from derived class extends the functionality of the method from base class.

This can be achieved using 'super'

E.g.

super.methodName();

TODOs:

Enhance CMS as per the following

Create a class CourseOperation with following method:

float getData(Course [])

Create a subclass of CourseOperation as CourseCostOperation in such a way that

it overrides "float getData(Course [])" that returns the Average Cost of all courses.

Create one more subclass of CourseOperation as CourseDurationOperation in such a way

that it overrides "float getData(Course [])" that returns Average Duration of all courses.

Package:

Package is a collection / group of classes and other type declarations e.g. Interfaces, Annotations, Enums, Records etc.

Benefits:

Allows to keep libraries isolated from other libraries.

Used to reduce naming conflicts

Creating package:

Packages are created using 'package' statement. It must be the first statement within a source file

Accessing classes from Packages:

If 2 or multiple classes are defined in the same package, they can access each other directly irrespective of whether they are declared 'public' or not.

If one class depends upon another class and if both the classes are in different packages, then the class which is being used must be made accessible outside the package and this is done using 'public' access modifier.

Once a class is declared as 'public' it can be exported outside the package and any class from different package can import that class. This is done using 'import' statement.

How packages are stored?

Packages are stored in the form of folders whereas classes are stored in the form of '.class' files

E.g.
The statement: "class IceCream is a part of package day3_dessert" means
There is a folder: day3_dessert containing a file: IceCream.class

It is also possible to create a package inside another package. It is called a sub-package.
This can be done upto any level.

Default Package:
Whenever a class is created without any package, it becomes a part of some package by default which is called as Default Package.

The use of Default Package is discouraged.
The classes from explicitly created packages can be imported just by using 'import' statement whereas the classes from Default Package can be used only by other classes which are also from Default Package.

Access Modifiers revisited:

In Java, access modifiers are of 4 types:

1. private
   Accessible only within the class
2. public
   Accessible from anywhere
3. protected
   Accessible through out the whole package as well as outside the package if the class is a sub class.
4. <<DEFAULT>>
   Accessible through out the whole package but not outside that.

As long as same package is used, there is no difference among public, protected and DEFAULT.

When different packages are used, they behave differently.

Exploring Predefined Classes of Java:

Since having a class within a Default Package is discouraged, predefined classes of Java belong to some packages. Java follows standard practices.

Java provides number of predefined packages; but one of them is frequently used which "java.lang"

The speciality of "java.lang" is that it is imported by default. Therefore whatever public declarations present in "java.lang" are directly available to developers.

E.g.

String
System

java.lang provides wide range of classes:
E.g.

Object

It is a cosmic superclass in Java. It is the ultimate ancestor.
Every class in Java by default inherits the Object class.
Therefore a reference of type Object can refer to an object of any class.
E,g,

```
Object obj = new Employee();
Object obj2 = new Department();
```

The Object class provides a basic foundation using several methods.
E.g.

toString() (Reference: ToStringMain)

Prototype: public String toString()

Usage: Used to represent any object in the form of String

When any object is passed as a parameter to System.out.println(), JRE identifies dynamic type of the object reference and searches for toString() method in the respective class.

If found, invokes that method otherwise invokes method from the base class which is Object class.

The default implementation of toString() from Object class returns String holding Address of the object.
To get the actual contents, it is necessary to override it.

A call to the method toString() can be explicit or implicit. Any method that gets called implicitly by JRE is known as the Callback Method.

finalize()

Prototype: protected void finalize()
Usage: Used to perform clean up operations when an object is getting Garbage Collected. It has to be overridden in the respective class.

equals() (Reference: EqualsMain)
Prototype: public boolean equals(Object)
Usage: Used to test the objects for equality.
When objects are compared for equality using '==', it is checked whether the 2 references share the same memory or not. If they share same memory, returns 'true' otherwise 'false'.
It is also possible to compare objects for equality using equals().
The default implementation from Object class behaves same as that of '==' operator.
This is insufficient from Project's perspective. Hence to compare them for equality against data, it is necessary to override equals()

TODOs:

Enhance CMS as per the following:

Modify class CourseStore with additional methods:

public static boolean isAvailable(Course)

The method must accept a Course object from caller and return the status of availability. Display appropriate message

public static float getAvgDuration(String providerName)

The method must accept the name of the course provider and return the average duration accordingly.

Day4:

Wrapper Classes (Reference: WrapperClassesMain)

Sometimes there is a need to represent primitives in the form of objects.
For every primitive type, Java provides a Class counterpart known as Wrapper or Wrapper Class.

E.g.

Primitive Type	Wrapper Class
byte	Byte
short	Short
long	Long

int	Integer
float	Float
double	Double
char	Character
boolean	Boolean

E.g.
Conversion of 'int' to Integer:
int x = 100;
Integer i = new Integer(x);

Reverse Conversion: from Integer to 'int'
int y = i.intValue();

The above fragment was required till Java 1.4 but since Java 1,5 it is more simplified.
Java 1.5 offers 2 features: Auto-boxing and Unboxing

Wrapper classes are also used to convert a qualified String into the respective primitive type. This is frequently required in GUI based applications.

E.g.
int x = Integer.parseInt("1234");
In the above statement String "1234" gets converted to int with value 1234

StringBuilder (Reference: StringBuilderMain):
A class from java.lang package that is used for String manipulation.

It is also possible to manipulate String data using String class but it causes performance implications because Strings are immutable.

Every single manipulation results in a new object creation.

To address this, Java provides StringBuilder which follows a mutable sequence of characters. It provides several overloaded append() method to perform manipulation.

Abstract Classes: (Reference: AbstractClassMain)

In Java sometimes a class may have a method without definition. Any method which is not defined but declared is referred to as the Abstract method.
E.g.
public class Shape {

```
public void draw();
}
```

The above code fragment is just for understanding the concept of Abstract method but in reality it is not a valid code because whenever any method is to be declared and not defined, Java enforces to attach one modifier: 'abstract'

E.g.

```
public class Shape {
        abstract public void draw();
}
```

The above fragment uses 'abstract' modifier but still it is not valid because when any class that declares at least one method as 'abstract', it must be declared 'abstract' and such a class is known as Abstract class.

E.g.

```
abstract public class Shape {
        abstract public void draw();
}
```

Now the above fragment is valid.

If a class contains any one method abstract, it must be declared abstract; however it is possible to declare a class as abstract without any single method abstract.

The modifier 'abstract' cannot be applied for:

1. Constructors
2. Static Methods
3. Variables

Once a class is declared 'abstract' it cannot be instantiated.
It is further used by creating its subclass.

Any subclass of an abstract class must implement all the abstract methods of that class otherwise it must be declared 'abstract' .

Even though an abstract class cannot be instantiated, it is possible to declare a reference of that type.

Abstract classes may have concrete (Non-abstract, Implemented) methods

---

Introducing 'final':

A keyword in Java that is used to declare variables immutable.

E.g.
final float pi = 3.14f;
The above statement makes the variable 'pi' as final. Any attempt to make changes results in compilation errors.

If a final variable is a reference to an object, it is the reference that must stay the same and not the object.

E.g.
final City c1 = new City("Pune");
c1.setName("Mumbai");
The above statement is still valid because the reference still stays the same.

c1 = new City("Mumbai");
The above statement is invalid because the reference tends to shift from 1st object to 2nd which is not possible since it is declared as 'final'

'final' can also be used in case of methods to prevent method overriding.
Any method declared as 'final' cannot be overridden.
If a method is declared as 'private', it cannot be overridden and therefore 'private' methods are implicitly 'final'

'final' can also be used in case of classes to prevent inheritance.
Any class declared as 'final' cannot be inherited.
If a class is declared as 'final', all the methods of that class become 'final' by default.

All Wrapper Classes including String are declared as 'final'.

Modifiers 'abstract' and 'final' cannot appear together.

Interfaces: (InterfaceExampleMain)

What is Interface:
Interface is a collection of abstract methods and possibly 'final' variables.
Used to declare the methods where implementation is not available.

Creating Interface:
Interface is created using 'interface' keyword

Like abstract classes, interfaces cannot be instantiated.
To use an interface further, there has to be some class which implements that interface and it is possible using 'implements' keyword.

DollarToRupeeConverter impl CurrencyConverter
RupeeToPoundConverter impl CurrencyConverter
E.g.
class MyClass implements MyInterface { ...}

Whenever a class implements an interface, it must define all the methods from that interface otherwise it must be declared 'abstract'

Even though an interface cannot be instantiated, it is possible to declare a reference of that type and that reference can refer to an object of a class which implements that interface

Why Interfaces?
Interfaces are mainly used to accomplish following things:

1.. Expanding the scope of polymorphism
2. Achieving multiple inheritance in terms of behaviours.

A class can extend only one class but can implement multiple interfaces

E.g.
public class Painting implements Drawing, Filling, Decorating { }
In the above statement, Drawing, Filling and Decorating are interfaces.

Basic Rules / Concepts:

1. Methods declared within a public interface are by default public and abstract.
2. Variables declared within a public interface are by default public, static and final.
3. A class can implement multiple interfaces
4. An interface can be implemented by multiple classes
5. An interface can extend multiple interfaces

(Java supports multiple inheritance for interfaces and not for classes)

Exception Handling:

What is exception:
An error that occurs during a program's execution is known as Exception.

In general, errors are of 2 types:

Compile Time Errors ---> errors

Runtime Errors ----> exceptions

Exception Handling is an object oriented way of handling runtime errors.
Exceptions are the actual objects created by JRE.
These objects contain error information.

Handling exceptions in Java:
To handle exceptions, Java provides 2 basic keyword:

1. try

Used to enclose the statements which are probable to fire an exception

It is used in the form of a block.

E.g.

```
try {
        Statement 1
        Statement 2
        ...
    }
```

2. catch

Used as an exception handler.

It is used in the form of a block.

Statements written inside the 'catch' block execute when an exception is raised.

E.g.

```
catch(ExceptionType var-name){
    Statement 1
    Statement 2
.   ..
}
```

How are exceptions raised?
Exceptions are raised by JRE.
JRE identifies the abnormal situation and maps it to the corresponding exception specific class.

E.g.

Abnormal Situation	Exception Class
Index out of bounds	ArrayIndexOutOfBoundsException
Access to null value	NullPointerException
Divide by Zero	ArithmeticException

Once the class is mapped, it creates an object of that class and checks whether the statement which has fired that exception; is enclosed within a 'try' block or not.

If enclosed, searches for the relevant 'catch' block and once found, executes that 'catch' block otherwise generates a predefined message associated with that exception.

If not enclosed, it generates a predefined message associated with that exception.

Exception Hierarchy:
The topmost class in Exception Hierarchy is Throwable from java.lang
It has 2 subclasses: **Error and Exception**

Error:
It indicates resource exhaustion in JVM. It is rare but fatal. It is not under the control of a developer.

Exception:
It is frequent but not fatal. It is under the control of a developer.

Types of Exceptions:
Exceptions are divided into 2 types:

1. Unchecked Exceptions (Runtime Exceptions)
   These are the exceptions for which the compiler does not enforce to handle them.
   These exceptions are expressed using a **predefined class: RuntimeException**
   Any class descended from RuntimeException is a runtime exception or unchecked exception.
   E.g.
   ArrayIndexOutOfBoundsException, NullPointerException etc.
2. Checked Exceptions
   These are the exceptions for which compiler enforces to handle them otherwise compilation error gets generated.

Any class which is not a descendant of RuntimeException is a checked exception.

E.g.

FileNotFoundException

Handling multiple exceptions (Reference: MultipleExceptionHandlerMain):

It is possible to handle multiple exceptions using multiple 'catch' blocks.

Day5:

Handling Default Exception: (Reference: DefaultExceptionHandlerMain)

Sometimes there is some exception raised which is not listed in predictions. In that case since there is no exception handler, JRE shows a system message and this needs to be fixed.
It is done by adding a 'catch' block that handles any type of exception.
E.g.

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

When multiple catch blocks are used, the 'catch' of super type must appear after the 'catch' block of subtype.

Method Invocation Vs Raise of an Exception

When a method is invoked, control jumps to its definition, executes the code and returns back to the calling environment.

When an exception is raised, provided it is handled, control jumps to the 'catch' block,executes the 'catch' block but does not return back.

Sometimes, there are some statements of which execution is very important.
Neither 'try' nor 'catch' blocks give guarantee about the execution.

To address this, Java provides a 'finally' block.
It gives full guarantee about execution. It gets invoked irrespective of whether an exception is raised or not.

E.g.

```
finally {
    //Statements
}
```

When 'finally' creates an impact?
The 'finally' block creates impact especially when the method returns a value other than 'void'

Rules about try - catch - finally:

1. All the blocks must be written one after the other without any statement in between.
2. A 'try' block must be used in conjunction with either 'catch' or multiple 'catch' or 'finally' or both.
3. 'catch' block cannot be written without 'try'
4. 'finally' block cannot be written without 'try'
5. It is possible to have nesting of the blocks as well.

Apart from try-catch-finally, Java provides 2 more keywords:

throws (Reference: ThrowsMain)

It is used at the end of the method or constructor signature to intimate the compiler that its statements may throw some exception but it is not willing to handle it; rather its calling environment should handle it.

This is applicable for Checked Exceptions and not for unchecked (Runtime) exceptions

Once a method uses 'throws' for a checked exception, it delegates a responsibility of handling that exception to its caller.

If required, the caller may further delegate that responsibility to its caller and this is done again by using 'throws'.

This chaining can be done up to any level but ideally it should stop at 'main()' because if it is not handled by 'main()', it will get propagated to its caller which is not under the control of a developer and therefore adding 'throws' declaration for 'main()' is not recommended at all.

throw: (Reference: ThrowMain)

Used to throw the exception explicitly. Sometimes there is a need to throw the exception programmatically. It is done using 'throw'

E.g.

throw <<Throwable>>

User Defined Exceptions (Reference: UserDefinedExceptionMain)

Sometimes depending upon the domain needs, it is necessary to create exceptions.

Such exceptions are called as User Defined Exceptions

To create user defined exceptions, it is necessary to create a class which is called as Exception specific class.

This class must inherit either RuntimeException or Exception or even their descendants.

To create an unchecked exception, inherit from RuntimeException and in case of checked exception, inherit from Exception.

Monitoring exception stack trace:

When the project is in the DEVELOPMENT phase, it is necessary to perform debugging and troubleshooting. Developer needs to know the point of origin of an exception and the trace of propagation.

To address this, Exception class provides a method: printStackTrace()

This is specifically meant for the projects which are in the DEVELOPMENT phase but definitely not suitable for the PRODUCTION phase.

TODOs:
Enhance CMS as per the following:

Create a class CourseEnrollment with following attributes:
- enrollmentId (int)
- enrolledBy (String) //Student Name
- courseId (int)

Provide getters, setters, constructors etc.

Create another class CourseEnrollmentData with following attribute:
- courseEnrollments (static CourseEnrollment [])

As soon as this class gets loaded the static array must be filled with some hard coded data values.

Create a user defined checked exception NoDataFoundException.

Create an interface CourseDataProcessor with following method:
- float process(Object obj) throws NoDataFoundException

Create 2 implementation classes of the interface CourseDataProcessor as per the following:

1st Implementation:
This class accepts name of the Student (enrolledBy) as input and returns the total amount of fees of all courses enrolled by that student.

The method must throw NoDataFoundException if there are no enrollments

found for that student.

2nd Implementation:
This class accepts ID of the course (courseId) as input and
returns the total amount of fees for all enrollments.

The method must throw NoDataFoundException if there are no enrollments found for that particular course.

Multithreading:

What is Multitasking?
Multitasking involves running multiple tasks simultaneously.

Multitasking is having 2 forms:
1. Multiprocessing: Running multiple processes simultaneously.
2. Multithreading: Running multiple threads simultaneously.

What is a Process?

Program is a set of instructions and every running instance of a program is known as Process.

Every process runs in its own address space and therefore the entire context including
variables, global variables, objects etc. is stored separately.
Therefore there is no sharing of data between 2 processes.

OS implements Multitasking through Multiprocessing

What is Thread?
Thread is an entity within a process. It defines the path of execution.
Java implements Multitasking through Multithreading by taking it one level lower.
Since multiple threads run within the same process, they may share the same data.

Every Java application has at least one thread: Main Thread

Implementing multithreading in Java: (Reference: SimpleThreadMain)
Since a thread is an entity within a process, it is meant to perform some task.
The task is to be defined by creating a Java class that is referred as Thread Implementation Class.

To implement this class, java.lang package provides 2 APIs:
- Thread (Class)
- Runnable (Interface)

Hence either a class inherits Thread or implements Runnable

Every thread implementation class must define a logic using a 'run()' method.

Implementing Multithreading by extending Thread class (Reference: ThreadMain)

InterruptedException is a CHECKED exception

Thread Life Cycle:
- The life cycle of thread has several stages:
  - BORN
    - When a thread is newly created, it is said to be in BORN state
  - READY
    - When a thread becomes ready to run, it is said to be in READY state
  - RUNNING
    - When thread's execution begins, it is said to be RUNNING state
  - BLOCKED
    - At any time a thread might be sent into some blocking state.
    - In general it is referred to as a BLOCKED state.
    - There are several types of BLOCKED states:
      - SLEEPING
      - WAITING
      - BLOCKED for IO Request
      - SUSPENDED
  - DEAD
    - When a thread is no longer available, it is said to be in DEAD state.
    - There are 3 possibilities under which a thread enters into DEAD state:
      1. Completes its task
      2. It is killed explicitly (forcefully)
      3. It is killed due to uncaught exception

For 1 CPU, 1 thread runs at a time. OS implements Context Switching algorithm. Therefore it creates an impact as if running multiple threads simultaneously.

Every OS has its own algorithm that decides scheduling of threads. It is known as Scheduler.
Therefore, the behavior of multithreaded programs is Platform Dependent.

Day6:

Scheduling of Threads:

Scheduling of Threads is done by an OS specific algorithm known as Scheduler. The scheduler schedules threads based upon several factors:

1. Priority

It is possible to assign a priority for threads within the range (1 to 10) being 1 as Min and 10 as Max. Thread class provides setPriority() method to set the priority. The default priority is taken as 5

2. Time Slicing

Every thread is allowed to run for a specific time period. As soon as the time out occurs, thread leaves RUNNING state and goes back to READ state.

3. Pre-empting

It combines 1st and 2nd. A higher priority thread may preempt a lower priority thread.

As soon as a thread acquires RUNNING state, a run() method for that thread gets invoked.

Implementing Multithreading using Runnable interface (Reference: RunnableMain)

Java provides 2 predefined APIs for implementing multithreading:

Thread Class

Runnable Interface

A Thread class already implements Runnable interface

A Runnable interface declares a single method: public void run(); and a Thread class defines that method because a Thread is a non-abstract class.

While implementing Multithreading using the Runnable interface, it is necessary to send Runnable as a TARGET for the Thread class object in order to invoke run() method from the respective Runnable implementation class.

This is possible by using an overloaded, parameterized constructor of Thread class. Thread(Runnable [TARGET])

Methods from Thread class:

Thread class provides several methods to handle thread related activities.

1. start()

Makes a request to the OS for the creation of a thread. Once created, brings the thread into READY state.

2. stop()

Kills the thread forcefully. Since it is not recommended, new versions of Java have declared this method as DEPRECATED

3. sleep()

A static method that causes the currently running thread to enter into SLEEPING state.

4. yield()

A static method that causes currently running thread to give up the control to some other thread.

After yield() the current thread leaves the RUNNING state and goes into READY state.

5. suspend()

Causes a thread to enter into SUSPENDED state.

6. resume()

Causes a thread to wake up from SUSPENDED state and enter into READY state.

Since it is implementation dependent, both these methods are declared as DEPRECATED.

7. currentThread() (Reference: CurrentThreadMain)

A static method that is used to obtain a handle to the currently running thread.

8. join()

A thread from which new threads are created, is called as Parent Thread and newly created threads are called as Child Threads.
It is possible that a parent thread is finished but the child thread is still alive.
Sometimes it is required to tell the main thread to continue only after the death of child thread.
This is done using join() method. It causes the parent to wait until the death of the child thread on which it is invoked.

Why run() method cannot add 'throws' declaration?

When a super type method does not add 'throws', a sub type overridden method

cannot add 'throws' because narrowing the scope is not possible.

But widening the scope is possible e.g. A super type method adds 'throws' but a sub

type overridden method does not add 'throws'

Synchronization:

Since multiple threads run within a single process, they may share the same data.

Synchronization is a technique that comes into picture when threads are sharing the same data. Sometimes it is necessary to put a restriction on accessing the object simultaneously.

If synchronization is not handled properly, threads enter into RACE condition and This leads to data inconsistency.

Implementing Synchronization: (Reference: SynchronizationMain)

Key to synchronization is a monitor. A monitor is a mutually exclusive lock.

In synchronization, a thread that enters the monitor, is said to have acquired lock on the object. Other threads have to wait until the lock is released by the current thread.

To handle synchronization, Java provides a keyword: **synchronized**.

The keyword 'synchronized' is used to create synchronized context which is possible by 2 ways:

1. Using synchronized method

When a currently running thread invokes a method on an object and if the method is declared as synchronized, then the thread acquires a lock on that object.

This ensures preventing access to that same object by other threads. As soon as the control exits from the synchronized method, the lock gets released automatically and it is available for other threads.

2. Using synchronized block

The object which is being shared, its class might have been defined by some developer and the developer that creates a thread implementation class may not have access to the source code of the class defined by the 1st developer.

In this scenario, to handle synchronization, a synchronized block is used.

The general form of synchronized block is :

```
synchronized(Object){
        //Statements
}
```

When 'synchronized' is used in case of methods?

While defining a class, a developer can go on declaring all the methods of that class

as 'synchronized' because developer wants to providing in-built functionality of synchronization to other developers.

If this happens, that class is called a synchronized class or a thread-safe class.
E.g. java.lang.StringBuilder (Since JDK 1.5) is not thread-safe where as
java.lang.StringBuffer is thread-safe

Inter-thread Communication

When threads are sharing the information, they may have to interact with each other.

This is known as Inter-thread Communication.

To implement Inter-thread Communication, 3 methods are provided:

wait()
notify()
notifyAll()

These methods are defined by java.lang.Object and they are declared 'final'
They must be invoked within a synchronized context otherwise IllegalMonitorStateException is raised.

wait()

It is used to release the lock explicitly. When invoked, releases the lock and sends the currently running thread into WAITING state,

notify()

It is used to send the notification to the thread which is WAITING state. When invoked, causes the thread to leave the WAITING state and go back to READY state. It notifies a single thread

notifyAll()

It notifies all waiting threads.

TODOs:

1. Write a program that creates 2 threads to perform following operations:
The first thread prints all the English alphabets in Uppercase
The second thread prints the values of the alphabets
E.g.
First Thread: A, B, C and so on
Second Thread: 65, 66, 67 and so on
Both the threads must print their values using a same time gap.

2. Create a class PaytmAccount with following attributes:
mobileNo (String)

name (String)
balance (int)

Implement 2 additional methods:
void addMoney(int)
void makePayment(int)

Write a program that creates 2 threads in such a way that both the threads use the same object of PaytmAccount and one thread adds the money whereas the another thread makes the payment.

Display messages with current balance using some time gap while performing both the transactions.

Note***
(When one thread is using the object, access should be denied for the other thread.)

Day7:
IO and File Handling

Every application needs to read some data and write some data.
To read the data, application needs some source whereas to write the data, it needs some destinations.

E.g.
In case of READ operation, an application can fetch input from the user using a keyboard. In that case Input Device (Keyboard) acts as a SOURCE.

Sometimes an application may want to fetch data from a file, in that case a file acts as a SOURCE.

Even an application may want to fetch data across a network, in that case a socket acts as a SOURCE.

This indicates that sources can be of different types e.g.:
Input Device, File, Socket and so on.

In case of WRITE operation, the application can send output to the user using some Output Device.
In that case an Output Device like Console or Printer acts as a DESTINATION.

Sometimes, an application may want to write data to some file, in that case a file acts as a DESTINATION.

Even an application may want to send data across a network, in that case a socket acts as a DESTINATION.

This indicates that destinations can be of different types e.g.
Output Device, File, Socket and so on.

Whatever is the operation to be performed, whether READ or WRITE, there is some connector required known as Stream.

Since there are 2 operations: READ and WRITE, there are 2 types of streams available:
InputStream and OutputStream.

These streams are meant for performing generic operations e.g. InputStream for READ and OutputStream for WRITE but still they do not indicate the specifics e.g. InputStream does not indicate anything about SOURCE similarly OutputStream does not indicate anything about DESTINATION.

Therefore Java provides 2 base classes : InputStream and OutputStream which are declared as 'abstract'.

This entire IO specific API (library) belongs to the java.io package.

Depending upon the source and destination, java.io package provides relevant sub classes.

E.g.
FileInputStream and FileOutputStream

FileInputStream: A subclass of InputStream used to perform READ operation on File.

FileOutputStream: A subclass of OutputStream used to perform WRITE operation on File.

Reading contents from File: (Reference: FileReadMain)
To read the contents from a file, an object of FileInputStream is required. But while creating the object, the constructor may throw FileNotFoundException which is a CHECKED exception and therefore it must be handled or delegated.

If the file is not available, FileNotFoundException gets fired otherwise an object of FileInputStream is created and it maintains a file pointer which starts pointing to BOF (Beginning OF File) position.

To start reading the contents, read() method is used. There are several overloaded read() methods. One of them reads one character at a time and returns its value.

If EOF (End Of File) position is encountered, read() method returns -1.

Writing contents into File (Reference: FileWriteMain)
To write data into a file, an object of FileOutputStream is required.
If the specified path or filename is not available, it gets created.

To perform WRITE operation, OutputStream provides several overloaded write() methods.
One of them accepts data in the form of 'byte [ ]'.

If original data is in String format, it needs to be converted into 'byte [ ]' which is possible using getBytes() method of String class.

When FileOutputStream is used, by default the file gets opened in WRITE mode.
It means, if the file does not exist, it gets created otherwise it gets overwritten.

To avoid overwriting the contents, it is necessary to enable APPEND mode. This is done using 2 parameterized constructor of FileOutputStream with 2nd parameter as boolean 'true'

TODOs:

1. Write a Java program that accepts file path as a command line argument and
   prints its contents in reverse order.
   E.g.
   Original contents: Hello, How are you?
   Output: ?ouY era woH, olleH

   (Hint: Use StringBuilder with its appropriate methods)

   Display appropriate messages if the file does not exist.

2. Write a Java program that accepts 2 command line arguments:
   1. Source File Path and 2. Destination File Path.
   The program must copy the contents from one file to another.
   (Hint: Use StringBuilder with its appropriate methods)

   Display appropriate messages if the source file does not exist.

While working with IO, streams are handled using basic 3 steps:

1. Open
2. Use
3. Close

If the stream is not closed, whatever system level resources consumed by that stream are kept opened unnecessarily. Therefore it is a standard practice to close the stream.

It is better if the stream gets closed automatically once its job is finished. This is made possible by Java version 1.7 with the help of "try-with-resources".

A 'try' block that declares one or multiple resources. Any class that implements java.lang.AutoCloseable interface, its object can be used as a resource.

"try-with-resources" Basic Format: (Reference: FileReadUsingTryWithResourcesMain)

```
try(....Resources.....){
        //statements
}
catch(...){
        .....
}
```

Exploring java.io

java.io package provides a wide range of classes and interfaces to handle IO operations.

Typically to work with files: FileInputStream and FileOutputStream.

Several other classes are as per the following:

- FilterInputStream
- FilterOutputStream
- File
- RandomAccessFile and so an.

FilterInputStream:

It acts like a filter to transform raw bytes into desired form. It provides a number of several subclasses.

E.g.

LineNumberInputStream

Used to work on the basis of line numbers

SequenceInputStream

Used to read contents from several InputStreams arranged in a sequence.

BufferedInputStream (Reference: BufferedInputStreamMain)

Used to apply a buffering model to improve performance. It uses a buffer that maintains a data. When all the characters are read from the buffer, it is refilled with next set of characters.

DataInputStream

It is used to read Java's primitive data types.
It provides relevant readXXX() methods.
E.g.
readInt()
readLong()
readShort()
readFloat()

File Class (Reference: FileMain):

It does not belong to any InputStream or OutputStream. It directly extends Object.

It is used to perform file specific operations like:

1. Checking whether the file is present or not
2. Creating a new file
3. Checking whether the path refers to file path or directory path
4. Creating a new directory

It provides relevant methods:

exists()
createNewFile()
isFile()
isDirectory()
mkDir()

Object of type java.io.File does not actually open any file.
File class provides a utility method: length() that returns file length.

*** It is possible to read the file contents and transfer the same into a byte array.

Day8:

RandomAccessFile (Reference: RandomAccessFileMain)

Like java.io.File, RandomAccessFile does not belong to any InputStream or OutputStream.

It directly inherits the Object class.
It is used to place a file pointer anywhere in the file using seek() method.
It can be used to perform reading as well as writing

Stream Categories:

Stream are divided into 2 categories:

Byte Streams

They work upon 8-bit data.

They fall under InputStream and OutputStream

Character Streams

They work upon 16-bit data.

Especially used to read unicode characters.

They fall under Reader and Writer

Like byte streams, java.io package provides a variety of character streams.

E.g.

FileReader, FileWriter

BufferedReader, BufferedWriter

and so on.

BufferedReader (Reference: BufferedReaderMain)
It is a character stream used to apply a buffering model on some Reader.
It provides a utility method: readLine() that is used to read the whole line. When EOF is encountered, readLine() returns 'null'.

Since BufferedReader can be used to read the contents line-by-line, it makes possible to fetch information from the file and build Java objects accordingly. (Reference: BufferedReaderForObjectCreationMain)

System Class:

A class from java.lang package.

Useful to print something on the console or to accept input from the end user.

It provides several public static final variables e.g. 'in' and 'out'.

'out' is declared of type java.io.PrintStream whereas 'in' is declared of type java.io.InputStream

'out' represents the console and 'in' represents the Input Device which is the keyboard.

Hence, while accepting user input, it is necessary to open an InputStream which is specific to the Input Device (Keyboard).

It is done by using "System.in".

In early days, it was done using BufferedReader but in that case a conversion was required explicitly.

E.g.

String to int: Integer.parseInt()
String to float: Float.parseFloat()

This is simplified by Java 1.5 through a utility class known as Scanner which belongs to the java.util package.

It handles conversion implicitly using several nextXXX() methods.
E.g.

To read 'int': nextInt()
To read 'float': nextFloat()

(Reference: UserInputMain)

***If a String data is being captured after some other information, an extra call to nextLine() is necessary.

Serialization:

State of an object:

State encapsulates current values available in the object. These values are temporary because eventually the object becomes eligible for Garbage Collection.Once it gets destroyed, all the information is lost.

Sometimes, it is necessary to preserve this information for future use. This is possible using Serialization.

What is Serialization:

The process of storing the state of an object to some permanent persistent store is known as Serialization.

Serialization does not store only data of an object; rather it stores the entire object in a byte-stream format.

This avoids performing extra processing while retrieving the state back.

Once a state is serialized, any time it can be retrieved back using De-Serialization.

What is De-Serialization?

Retrieving the state back is known as De-Serialization

Implementing Serialization (Reference: SerializationMain) and Deserialization (Reference: DeserializationMain)

To implement Serialization and Deserialization, java.io package provides an API known as Serialization API.

It mainly consists of:

1. Serializable

An interface that needs to be implemented by a class of which an object is to be serialized.

It is a marker interface. It does not have any method.

If not implemented and if the object is about to get serialized, JRE throws exception: NotSerializableException

2. ObjectOutputStream

A subclass of OutputStream which is used to actually perform serialization.

It uses OutputStream as a target.

It provides a method: void writeObject(Object) to perform serialization.

3. ObjectInputStream

A subclass of InputStream which is used to actually perform de-serialization.

It uses InputStream as a source.

It provides a method: Object readObject() to perform deserialization.

Serialization Further Concepts:

Sometimes an object may hold some confidential or sensitive information. In such cases, it is required to perform encryption while serialization and decryption while deserialization.

Serializable interface does not have control over encryption and decryption because it is a marker interface (Without any method).

To have full control over encryption and decryption, java.io package provides an interface: Externalizable

It is a sub interface of Serializable and it provides 2 methods:

writeExternal() and readExternal()

writeExternal() is used to provide encryption logic and it gets invoked implicitly when the object is getting serialized.

readExternal() is used to provide decryption logic and it gets invoked implicitly when the object is getting deserialized.

If a class of which an object is being serialized, declares a 'static' variable, its state does not get serialized.

It gets initialized to default value during deserialization.
E.g. int - 0, float - 0.0, boolean - false and so on.

Sometimes it might be required to avoid serialization of a specific variable which is not 'static'.

To accomplish this, Java provides a keyword: 'transient' which avoids serialization for that particular variable.

E.g.

```
public class A implements Serializable {
        private int x;//Gets serialized
        private static int y;//Static Variable: Does not get serialized
        private transient int z;//Non static Variable: Does not get serialized
        ....
}
```

Transient variable gets initialized to default value during deserialization.

TODOs:

Enhance CMS as per the following:
Write a Java program to accomplish following things:

1. Declare an array of 3 Course objects.
2. Accept user input to fill the information into this array.
3. Store this entire array into some file using Serialization

Write another program to accomplish following things:

1. Deserialize the entire array.
2. Print average cost of the courses

NIO Basics (Reference: NioBasicsMain):

NIO stands for New IO
A separate API provided by Java to optimize performance.
NIO provides non-blocking capabilities due to which performance is gained.

NIO specific API belongs to java.nio package which is at the base.
Mainly it is provided by:

java.nio

It provides several types of buffers which are used as containers for data.

E.g.

ByteBuffer

java.nio.channels

It provides several types of channels which act as mediators for transferring the data.

E.g.

FileChannel

java.nio.file

It provides several utilities which are used to handle file operations.

E.g.

Path:

An interface to indicate some path.

Paths:

A class that acts as a factory for the creation of Path

Files:

A class that is used to handle file operations.

When a buffer is filled with information, it is necessary to set the buffer pointer to the beginning position otherwise it causes a BufferUnderflowException. This is done using flip() method.

Collections Framework

Collection is a data structure that is meant for holding several elements.

Collection Framework is used to handle mainly 3 basic operations:

1. ADD
2. RETRIEVE
3. REMOVE

Array Vs Collection

1. Array has fixed dimension whereas collection grows dynamically.
2. Array can hold primitives as well as objects whereas collection accepts only objects.
3. Array can hold similar typed elements only whereas collection can hold different types of objects.

Java provides several types of collections using Collections API which belongs to the java.util package.

Legacy Classes:

These classes are available right from the 1st version of Java.

E.g.

Stack (Reference: StackExampleMain)
Vector
Dictionary
Hashtable
Properties

Stack:

It is a data structure that follows the LIFO algorithm.
(LIFO --> Last In First Out)
Apart from add() and get() it provides some other operations.
E.g.

push(): Used to push the element onto the stack
pop(): Used to remove the element from the top.
peek(): Used to retrieve the element from the top.

Day9:

Vector (Reference: VectorExampleMain)

This is another collection that belongs to the Legacy Class category.
It maintains elements using indexes.

Typically, every collection has 2 properties:

1. size
   Indicates number of elements available in that collection
2. capacity
   Indicates number of elements the collection can accommodate at the current time.

By default, a vector has a capacity 10. As soon as size exceeds capacity, the capacity becomes double.

ArrayList (Reference: ArrayListExampleMain)

Like Stack and Vector, ArrayList uses indexes to store the data.
In fact, ArrayList behaves exactly the same as that of Vector.
The only difference is Vector is by default Thread-Safe and ArrayList is not.

LinkedList (Reference: LinkedListExampleMain)

It behaves the same as that of ArrayList.
E.g.

Like ArrayList, LinkedList is not thread-safe.
But it provides additional capabilities because it follows Double-Ended Queue pattern.

This pattern allows to add the element directly at theTOP, remove the element directly from the top and so on.

All these collections like Stack, Vector, ArrayList and LinkedList behave the same till a certain level.
E.g.
All of them are ordered (Use Index)
All of them accept duplicate values.

However, beyond that level there are slight variations.

Collections Framework Big Picture

Java Collections Framework provides several predefined interfaces.

The most important is Collection itself which has 2 sub interfaces:

1. List
2.Set

There is another category called as **Map**

Each interface has some contract (specification).

E.g.
List interface has following contract:
1. Ordered Collection
2. Permit Duplicates.

Stack, Vector, ArrayList and LinkedList classes implement List interface.

Type Safe Collection

(Reference:TypeUnsafeCollectionMain,TypeSafeCollectionMain)
This is a feature introduced by Java 1.5 with the help of another feature known as Generics.

Generics feature allows to declare classes or interfaces which are working upon some data but the type is not mentioned.

It is especially used by interfaces that declare generic type and

their classes specify the type.

E.g.

```
interface Processor<T, R> {
        R doProcess(T t);
}
```

In the above fragment, interface Processor provides a process() method that accepts some value and returns some value but the signature and the return type are yet to be decided.

Hence this interface is a generic interface. Its implementation class is supposed to provide the specific type.

E.g.

```
class ProcessorImpl implements Processor<String, Integer> {
        Integer doProcess(String s){ ...}
}
class ProcessorImpl2 implements Processor<Float, Double> {
        Double doProcess(Float f){...}
}
```

Here 'T' and 'R' are just placeholders or arbitrary names.

When a collection is declared as type safe, compiler understands that the collection must hold objects of a specific type.

Any attempt to add objects of a type other than the specified one, results in compilation error.

Since compiler is aware about the type, there is no explicit cast needed.

Set Interface:

It is another sub interface of Collection.
It has following contract:

1. *Unordered*
2. *Prevents Duplicates*

It is implemented by HashSet (Reference: HashSetExampleMain)

Why Set is unordered?

Set does not use index to store the element rather it uses some hashing algorithm to store the element.
(Reference: HashSetWithUserDefinedObjectExampleMain)

While adding objects in a hash based collections, the implementation class of the object must override equals() and hashCode() to ensure the proper behavior.

Understanding hashCode():

Object class provides a method: public int hashCode().
It is used to provide hashing algorithm which is further used by Hash based collections.

This algorithm is decided by the Owner of the class; however in most cases it is auto generated using IDEs like Eclipse, IntelliJ, JDeveloper and so on.

Sample algorithm of hashCode():

```
public int hashCode(){
        return name.length() + capital.length();
}
```

If 2 objects are equal, definitely their hash codes are equal.
E.g.

```
Country c1 = new Country("India", "Delhi");
Country c2 = new Country("India", "Delhi");
```

In the above fragment, both the objects are equal (from contents perspective) and hence they have same hash code which is 10 in this case.

Whereas, if 2 objects are unequal, still they may have same hash codes.

E.g.

```
Country c1 = new Country("India", "Delhi");
Country c2 = new Country("Japan", "Tokyo");
```

In the above fragment, both the objects are unequal (from contents perspective) but still they have same hash code which is 10 in this case.
When any object is getting stored in a hash based collection, its hashcode is calculated based upon the algorithm provided by hashCode() method.

Once the hash code is found, depending upon the requirement, a container is created which is called a hash bucket.

Map (Reference: HashMapExampleMain):

Although List and Set type of collections are completely opposite to each other, they have one similarity.

Both of them are VALUE based collections.

Whereas Map maintains data in the form of KEY-VALUE pairs.
Every element has a unique KEY. The VALUE may repeat.

It has several implementations.
E.g.

Hashtable

It is a legacy class and it is by default thread-safe.
It does not accept 'null'

HashMap

It is provided by later versions of Java. It is not thread-safe.
It accepts one 'null' key and several 'null' values.

Properties (Reference: PropertiesExampleMain)

It is a separate Map implementation that allows to work upon properties which are mentioned in the form of name-value pairs.

It is a utility class especially used to load configuration specific properties which are defined within some PROPERTIES file in the form of 'name=value' pairs.

TODOs:

Enhance CMS as per the following
Modify previous TODO to accomplish following thing:

When the application starts, it must display following menu

1. Create New Course
   Accepts User Input and stores the course into collection
2. Store Courses
   Serializes the entire collection
3. Show All Courses
   Deserializes the collection and displays the values.
4. Exit
   Terminates the program

Enter your choice:

This program must be running endlessly.

Using Collections as a Property

(Reference: CollectionAsPropertyExampleMain)

Depending upon the domain requirement, classes may declare a property of some collection type.

E.g.

Department holds Many Employees
TrainingProgram holds Many TrainingModules

Ideally, when a class which declares a property of some collection type, it must provide some method that allows to add one value at a time to update that collection.

This becomes a convenient approach to managing these properties. E.g.

Providing addEmployee(Employee) method within Department class.

Whenever a class declares a collection type property, it is required to be initialized otherwise it may lead to NullPointerException

Sorted Collections:

Since a collection holds multiple values, it is a very common requirement to sort these values.

To accomplish this, Collection Framework provides 2 interfaces:

SortedSet and SortedMap

SortedSet (Reference: TreeSetExampleMain):

It is an extension to Set but stores elements in a sorted order.
It is implemented by *TreeSet.*

SortedMap:

It is an extension to Map but stores elements according to their *keys* in sorted order. Performs sorting on the basis of *keys* rather than values.
It is implemented by *TreeMap.*

In general, all collections have a common characteristic:

They can accept objects of different types unless they are declared as type-safe.

Exception to this is *Sorted Collection*. To perform sorting,

objects must be of same type otherwise it results into *ClassCastException*

Adding User Defined objects into Sorted Collections:

(Reference: TreeSetUsingUserDefinedObjectExampleMain)

While adding elements in a sorted collection, the element a specific class has to implement a java.lang.Comparable interface.

It is used to provide a default sorting algorithm.

It is a generic interface that declares a single method e.g.

```
interface Comparable<T> {
    int compareTo(T t);
}
```

Although Comparable can be used to provide sorting algorithm, it has a limitation. It provides a default sorting algorithm. Any further change in the algorithm needs change in the code which leads to Tight Coupling and not recommended at all.

To address this, Java provides one more interface:

java.util.Comparator:

It is used to further customize the sorting algorithm.

Day10:

Comparator interface is a generic interface and it provides a method:

```
int compare()
```

E.g.

```
interface Comparator<T>{
    int compare(T t1, T t2);
}
```

(Reference: PlanetSortingUsingComparatorExampleMain)

Once a Comparator implementation class is defined, it needs to be lined with *TreeSet* so that *TreeSet* will use the sorting algorithm given by that particular Comparator implementation.

If not linked, it uses a default sorting algorithm given by Comparable.

To link Comparator with TreeSet, use overloaded constructor of TreeSet which accepts some Comparator.

E.g.

TreeSet(Comparator)

Collection Utilities (Reference: CollectionUtilitiesExampleMain):

Apart from several collection oriented classes and interfaces, java.util package provides classes which are frequently used while working with collections.

E.g.

Arrays

Useful to populate collections in simplified manner

E.g.

It provides asList() method that accepts variable argument list and returns a List.

Collections

Useful to perform collection specific operations.

It provides several methods to handle the operations.

E.g.

static sort()

Language Essentials

Inner Classes
Annotations
Enums
Reflection API
DAO Pattern
Java 8 Features

Inner Classes:

A class defined within another class is an inner class.

E.g.

```
class A {
    class B{

    }
}
```

Usage:

Inner classes are useful when complex logic is to be isolated but handled locally within the main enclosing class.

They are frequently used in GUI programming model for building components, handling events etc.

Types of Inner Classes (Reference: InnerClassExampleMain):

1. Static Inner Class

An inner class declared with 'static' modifier is a Static Inner Class.

E.g.

```
class A {
        static class B {

        }
}
```

It can access only static members of an outer class.

It can be instantiated without an object of outer class.

2. Nested Class (Non Static Inner Class)

An inner class declared without 'static' modifier is called as a Nested Class.

E.g.

```
class A {
        class C {

        }
}
```

It can access static as well as non-static members of an outer class. It must be instantiated using an object of outer class.

3. Local Class

A class defined within a method is known as Local Class.

E.g.

```
class A {
        void m1(){
                class D {
                }
        }
}
```

It is loaded, instantiated and used for every method call.

In Java, 3 combinations are allowed:

Defining method inside a class.

Defining class inside another class

Defining class inside method

Defining method inside another method is not allowed

4. Anonymous Inner Class

An inner class without any name is known as Anonymous

Inner Class.

It is especially used in case of Abstract classes and Interfaces.

It allows you to adopt new requirements without having the overhead of creating a separate class. (Reference: AnonymousInnerClassExampleMain)

For every type declared in Java whether it is a class, interface, annotation, enum or even record, a class file is created.

Hence, class files are created for inner classes as well. These file names follow general form as per the following: <<OuterClassName>>$<<InnerClassName>>.class

E.g.
OuterClass$StaticInnerClass.class
OuterClass$NestedClass.class

Annotations:

It is a feature introduced by Java 1.5
It is frequently used to handle configurations, metadata and so on.

Annotations are denoted by '@' symbol.
E.g.
@Override

They have mainly 2 properties:

1. Target

It specifies the location where an annotation can be applied.
E.g.

TYPE
METHOD
FIELD
etc.

2. Retention Policy

It specifies the scope of the annotation that means till what moment the annotation will have its impact.
E.g.

SOURCE
CLASS
RUNTIME

TODOs:

Enhance CMS as per the following:
Create an interface CourseDataSorter with following method:

Collection<Course>
sortCourses(Collection<Course>);

Create 2 implementation classes of CourseDataSorter as per the following:

1st Implementation:

Accepts a collection of Course types and returns a sorted collection using

default algorithm.

2nd Implementation

Accepts a collection of Course types and returns a sorted collection using

some customized algorithms.

***Note: The default and customized algorithms are of your choice.

Enums (Reference: EnumExampleMain):

A feature introduced since Java 1.5
It is known as enumerated type.
It allows to have a value which is one of the several possible values.

Enum is declared using 'enum' keyword.
E.g.

```
enum Nationality {
    INDIAN, AMERICAN, GERMAN, OTHER
    }
```

Once an enum is created, it can be used as a property of some class.

Reflection
(Reference: StaticReflectionExampleMain, DynamicReflectionExampleMain):

Sometimes there is a need to perform operations depending upon the runtime situation.

E.g.

Creating an object where the class name is decided at runtime.

Sometimes there is a need to fetch information about the class of an object at runtime.

E.g.

Name of the class

Methods available in that class
Fields available in that class
Interfaces implemented by that class
and so on.

To handle such activities, Java provides an API known as Reflection API.

The Reflection API mainly consists of 4 classes:

1. java.lang.Class
2. java.lang.reflect.Method
3. java.lang.reflect.Field
4. java.lang.reflect.Constructor

Implementing Reflection:

There are 2 ways to implement Reflection:

1. Static

The class of which information is to be fetched is predetermined.

This is done by manually obtaining a reference of 'Class' class for some specific class.

When any type name is suffixed with '.class', the whole value becomes compatible with 'Class' class type.

2. Dynamic

The class of which information is to be fetched is decided at runtime

There are 2 ways to handle dynamic reflection.

1. Based upon an object of a class of which information is to be fetched:

   This is made possible using getClass() method of Object class which return a reference of type 'Class' class.

   E.g.

   ```
   Class cls = obj.getClass();
   ```

2. Based upon name of the class of which information is to be fetched:

   This is made possible using 'forName()' static method of 'Class' class.

   It accepts class name as String and loads the class explicitly. Once the class is loaded, it returns the

reference of 'Class' type itself.
If .class file is not in the classpath,
ClassNotFoundException is raised.
It is a CHECKED exception, needs to handled somewhere.

E.g.
Class cls = Class.forName(...String....)

'Class' class makes it possible to create an object of some class where the class name is decided at runtime.

This is done by using newInstance() which returns Object type.

Day11:

DAO Pattern:

DAO stands for Data Access Object.
It is a commonly used Design Pattern that handles persistence layer of a business application.

Typically a business application is composed of several layers.

1. Presentation Layer: Handles User Interface
2. Business Layer: Handles Business Logic
3. Persistence Layer: Handles interaction with Data Store

In a business application, there are 4 common operations:
Create, Read, Update, Delete (CRUD)

Implementing DAO (Reference: DaoExampleMain):

Typically this pattern is implemented by providing some interface and its corresponding implementation class.

For better design, this interface is a generic interface.

TODOs:

Enhance CMS in such a way that it uses DAO pattern to perform CRUD operations.

Java 8 Features:

1. Interface Enhancements (Reference: InterfaceEnhancementMain)

Prior to JDK 8, it was not possible to define methods within an interface.
JDK 8 allows to defines methods within an interface provided they are declared as either 'default' or 'static'

This eliminates the need for writing abstract classes when some template is to be defined.

2. Functional Interfaces

This is a new terminology introduced by Java 1.8.
Any interface that declares one and only one abstract method then that interface is called a Functional Interface.
E.g.

Runnable
    run();
Comparable
    compareTo();

To cross check whether the interface is a functional interface or not, Java 1.8 provides an annotation: @FunctionalInterface that can be applied at the interface level.

Apart from legacy interfaces which are technically functional interfaces, Java 1.8 adds a new library for functional interfaces. It belongs to package: java.util.function
E.g.

Consumer
Supplier
Predicate
Function
UnaryOperator
BinaryOperator

3. Lambda Expressions (Reference: LambdaExpressionMain)

This is a new feature introduced by Java 1.8 that relies upon functional interfaces.

It brought a revolution in Java Programming Style.

Due to this feature, Java developers can adopt Functional Programming.

A lambda expression is composed of 3 things:

1. Lambda Parameter

    It must match with the signature of the method of the interface for which lambda expression is defined.

2. Lambda Operator

It divides Lambda Parameter and Lambda Body.

It is denoted by '->'

E.g.

[Lambda Parameter] -> [Lambda Body]

3. *Lambda Body*

It provides the actual logic.

It is of 2 types:

A) Single Expression Lambda

This involves a single statement.

When a method of the interface returns a value,

'return' statement is not to be added.

B) Blocked Lambda

Sometimes there is a need to write complex logic

which needs multiple lines of code.

Blocked lambda serves this purpose.

The use of the 'return' statement is a must.

Whenever a lambda expression is defined for a functional interface which has a method that accepts some parameters,the lambda parameter may explicitly define the type or the type can be inferred implicitly.However it is not possible to declare the type explicitly partially.

4. Stream API *(*Reference: StreamAPIExampleMain)

It is an API given by Java 1.8 to perform operations on the data held inside collections.It provides support for several operations like:

Iterations

forEach(Consumer)

Filtering

filter(Predicate)

Mapping

map()

Reducing

reduce()

Sorting

sorted()

The API belongs to package java.util.stream which provides several libraries but at the core, there is an interface: *Stream.*

All collection specific libraries have been modified in such a way that they provide a method 'stream()' that returns a

reference of type 'Stream'

5. Date and Time API (DateTimeExampleMain)

A separate API that allows handling Date and Time.
Prior to 1.8, Java provided a support for Date and Time functionalities using java.util.Date and java.util.Calendar classes.

Although these classes were present, they were not that much developer friendly. E.g. Year starts with 1900.

To address such problems, Java adds a Date and Time API that belongs to the java.time package.

It mainly provides 3 classes:

- LocalDate: Handles Date
- LocalTime: Handles Time
- LocalDateTime: Handles Timestamp

Day11:

Advanced Java

JDBC

JDBC stands for Java to Database Connectivity
It is an API that allows Java applications to interact with Database

Every application has 2 paradigms:

Front End and Back End

Depending upon the requirement, front end can be developed using various platforms e.g. Java, Python, DOT NET etc.

Similarly, back end can be handled using various types.
Typically a File System or a Database.

In large scale applications, generally Database is preferred because it provides several services.

E.g.

- Simple Design (Tabular Format)
- Relational Model (Primary Key, Foreign Key etc)

Constraints (Not Null, Check, Default, Unique)
Joins, Optimization, Security

The way Java provides an API known as JDBC API, every DB Vendor provides its own API known as Vendor Specific API.

This API simplifies access for those programs which want to get connected with that database.

JDBC API is written as per Java specific standards whereas Vendor Specific API is written as per Vendor specific standards. Hence there is a conflict which is resolved using some mediator known as Driver.

Driver is a program that converts JDBC calls into some format that is compatible with Vendor Specific API.

In JDBC, there are mainly 4 types of drivers:

1. Type 1

   JDBC ODBC Bridge
   It uses 3rd party library ODBC: Open Database Connectivity which is given by Microsoft.
   It converts JDBC call into some format which is compatible with ODBC.
   ODBC uses its own driver that converts the call further into Vendor specific format.

   Drawbacks:

   1. Since it is a 2 phase conversion, it is slowest.
   2. It is Platform Dependent. Specific for Windows.
   3. Every client machine needs ODBC configuration setup.Therefore it is used up to limited extent.
      E.g. Small Scale Applications, Development Phase

2. Type 2

   Native API, Partly Java Driver.
   It uses a combination of Java and Database specific standards.
   It does not use 3rd party library.
   It uses Database specific native API to make a call.

   Benefits over Type 1:
   Faster

Platform Independent

Drawback

Every client machine must have Database specific native API.

3. Type 3

Net Protocol, Intermediate DB Access Server
This is used totally in a different context.

It is used when Java Application wants to interact with multiple database servers.

It uses a middleware that routes the call.

4. Type 4

Vendor Specific, Pure Java Driver.
Every DB Vendor provides its own driver for a specific database.

It makes use of TCP/IP socket connections to make a call.

The socket consists of 2 properties:
IP Address and Port No

Benefits:

- Fastest
- Platform Independent
- No configuration setup required on the client machine.
- Highly recommended for Production Environment and Large Scale Applications.

JDBC Core API:

The core API of JDBC belongs to the java.sql package.
It mainly consists of following:

- DriverManager (Class)
- Driver
- Connection
- Statement
- PreparedStatement
- CallableStatement
- ResultSet

Steps in JDBC:

1. Load the Driver

This is done by explicitly loading the driver class using Class.forName(...)
In case of type 4 driver, the required .class file must be added into the CLASSPATH.

2. Establish Connection
   This is done either by using DriverManager class or Driver interface.

   Typically, Connection is established based upon 3 properties:
   1. URL (Uniform Resource Locator)
   2. Username
   3. Password

3. Obtain some Statement
4. Execute SQL Query
5. If the query is a SELECT query, obtain a set of records and perform navigation

(Reference: SimpleSELECTQueryExampleMain)

Typically, Type 4 specific Driver implementation classes exist within some archive file given by the DB vendor and it is known as Java Archive (.jar)

This .jar file contains a bunch of several .class files.

Day12:

All JDBC URLs follow some common format up to a certain extent.
Rest depends upon the Database.

E.g.

Sample URL for MySQL:
jdbc:mysql://localhost:3306/db1
Sample URL for Oracle:
jdbc:oracle:thin:@localhost:1521/orcl

In the above sample URLs, the common format:
<<Main-Protocol>>:<<Sub-Protocol>> [Rest depends upon Database]

All JDBC URLs have Main Protocol as 'jdbc' itself
Obtaining Statement

In JDBC, queries are mainly divided into 2 types.

1. **Simple Query**
   A query without any parameter is a simple query.
2. **Parameterized Query**
   A query with parameters is a parameterized query.

In the case of Simple Query, 'Statement' interface is used.

To obtain 'Statement', Connection interface provides a method: createStatement() which returns 'Statement' type.

Executing SQL Query to fetch records:

In order to fire SELECT query, Statement interface provides a method: executeQuery(). which accepts a query String in String format and returns ResultSet interface type.

Performing Navigation:

A data fetched from database table using SELECT query, is stored in a Java Object of type ResultSet.

ResultSet is a representation of the data on the Client Side.
It is just a table-like structure containing rows and columns.
Every row has a record position and every column has a column index.
Both the record position as well as column index start from 1.

Apart from the actual record positions, every ResultSet has 2 additional positions: *BeforeFirst and AfterLast*.

By default the cursor pointer of ResultSet starts pointing to BeforeFirst.

To read the actual contents, it is necessary to move the cursor in the forward direction. This is done using next() method of ResultSet.

The next() method checks for the availability of the next record and returns boolean value.
If available, moves the cursor 1 step ahead and returns true otherwise returns false.

Fetching Data from ResultSet:

The data fetched from database table is stored inside the columns of the ResultSet and every column has an index which starts from 1.

***The index of the column of a ResultSet depends upon the SQL Query rather than the original table design.

To actually read the data, ResultSet interface provides several getXXX() methods.

E.g.

getInt()
getString()
getFloat()
getDouble()

All these methods accept a parameter which indicates column index and return the relevant value.

What happens when driver is loaded
(Why driver loading is required)

DriverManager is able to provide Connection with the Database using some driver only when the driver itself is registered with DriverManager.

Typically, all Driver oriented classes provide 'static' block which handles the logic of Registration.

Hence, to perform registration, the only thing needed is invoke that 'static' block which is possible by loading the class which is possible by using Class.forName(...);

Is Class.forName() really needed?

Since JDBC specification 4, one important feature has been introduced: Auto-loading of Drivers.

This is possible only when the respective Driver Implementation follows some criteria.

If the driver implementation provides :
/META-INF/services/driver-file which holds an entry of the Driver implementation class, it gets loaded implicitly.

Improvements: (Reference: RestaurantCRUDExampleMain)

Call to getConnection() is repeated.
Connection is being closed explicitly

Lacks DAO Pattern

Working with Parameterized Queries:

A query can have parameters. These parameter values may change at runtime.

Whenever a parameterized query is to be fired, PreparedStatement interface is used.

It does not mean that parameterized query cannot be fired using Statement. If Statement is used, as soon as the parameter value changes, a new version of query is created and it gets compiled again.

E.g.

"select * from restaurant_master where rest_id = " + <<rest_id>>

This leads to performance implications.

The ideal way would be: Let the query get created only once, compiled only once and cached.

This is possible using PreparedStatement.
Connection interface provides a method: prepareStatement() which accepts a query string as String and returns the type of PreparedStatement

JDBC uses '?' for unknown parameters.

Substituting value for unknown parameter: '?'

Every unknown parameter of parameterized query has parameter index which starts from 1 and increments by 1 from left to right.

E.g.

select * from emp where sal between ? and ? and location = ?

To substitute value in place of '?', PreparedStatememt interface provides several setXXX() methods.

E.g.

setInt()
setString()
setFloat()
setDouble()

These methods accept 2 parameters:

1. Parameter Index
2. Parameter Value

Obtaining a Connection using 'Driver' interface

All JDBC driver implementation classes implement an interface:

java.sql.Driver

To obtain a Connection using 'Driver' interface, it is necessary to create the object of the driver implementation class.

The Driver interface provides connect() method that accepts 2 parameters:

1. String

   Specifies the URL
2. java.util.Properties

   Specifies the User Credentials in the form of 'Properties' object

   There are 2 predefined properties used while working with JDBC:

   1. user
   2. password

   Finally it returns the object of type 'Connection'

Executing DML Queries:

Depending upon the requirement, an application may want to execute DML queries like INSERT, UPDATE and DELETE

Since DML queries do not return anything, ResultSet is not required in this case.

Since ResultSet is not used, the method executeQuery() is also not used while performing DML operations.

In fact, DML operations are performed using another method:

executeUpdate() which returns 'int' that indicates the number of records affected.

TODOs:

Enhance CMS to handle CRUD operations using DAO along with Database

Making a call to Database:

There are 3 ways to make a call to Database. This is possible using methods provided by Statement interface:

ResultSet executeQuery()	Used to fire SELECT query
int executeUpdate()	Used to fire DML query
boolean execute()	Used to perform several operations: 1. Firing dynamic queries 2. Firing DDL queries 3. Invoking PL SQL stored procedures It is done by using CallableStatement The return value indicates whether the outcome of the query is an object of type ResultSet or not.

Managing Transactions:

Transaction is a set of operations that must execute in a single unit.

Transactions need to be ATOMIC (Execute ALL or NONE)

If anything fails, the transaction must be rolled back.

Whenever any DML operation is performed through Java application by default changes are Auto Committed.

This leads to data inconsistency problems. To avoid this, it is necessary to disable auto-commit.

Connection interface provides relevant methods to manage the transactions:

- setAutoCommit(boolean)
  - Can be used to disable Auto Commit: setAutoCommit(false)
- commit()
  - Used to commit the transaction
- rollback()
  - Used to rollback the transaction

Day13:

Java EE

Java EE stands for Java Enterprise Edition.
It is a platform meant for developing Web Based as well as Enterprise level applications.

What is a Web Application?

Any application that is accessed using a Web Browser is a Web Application E.g. Google, Flipkart, Amazon etc.

A Web Application is of 2 types:

Internet

A typical website which is accessible to every one.

Intranet

A web application that is accessible to limited number of users.
E.g. Online portal of every organization where employees can perform several tasks like:

Leave Requisition
Filling up the time sheet etc.

What is Enterprise level application?

Whenever an application needs higher end services (Enterprise services) then that application is known as Enterprise level application.

List of Higher End Services:

Scalability
Availability
Performance
Security
Asynchronous Messaging
Persistency
Transaction Management

Java EE is a platform that is based upon Component Driven Architecture.

What is a Component?

A component is an Application specific reusable piece of code.
A component is mainly of 2 types:

1. Unmanaged

A component of which an object is created explicitly, is called as an Unmanaged component.
E.g.
FileInputStream fin = new FileInputStream(....);
Vector v1 = new Vector();
Typically, Java SE platform comes up with Unmanaged Components.

2. Managed

A component of which an object gets created implicitly, is called as a Managed component.

Typically, Java EE platform comes up with Managed Components.

In Java EE, there are 2 categories of Managed Components:

1. Web Components

A component that accepts Web Request and generates Web Response is called a Web Component.

In Java EE, mainly there are 2 Web Components:

a) Servlet
b) JSP

2. Business Components

A component that is responsible for handling Business Logic or Domain Specific Logic, that component is called as a Business Component.

In Java, Business Components are handled using:

EJB (Enterprise Java Beans)

In Java EE, managed components are taken care of by some runtime environment called Container.
Container is a runtime environment that is responsible for managing the life cycle of Java EE components.
Since there are 2 categories of Java EE components, there are 2 types of containers.

1. Web Container

It is responsible for managing the life cycle of Web Components like Servlet and JSP.

2. EJB Container

It is responsible for managing the life cycle of Business

Components like EJB.

How are these containers made available?

These containers are made available by 3rd party softwares known as Web Server and Application Server

Web Server:

It is a 3rd party software that provides an environment called a Web Container.

There are several Web Servers available in the market; but the most commonly used Web Server is given by Apache Software Foundation, known as Tomcat.

Application Server:

It is a 3rd party software that is an extension to Web Server because it provides Web Container as well as EJB Container.

There are several Application Servers available in the market:

1. Weblogic by Oracle Corporation
2. WebSphere by IBM
3. JBoss by Red Hat
4. Glassfish by Oracle Corporation

What is Servlet?

Servlet is a Server side component that is used to extend the functionality of Web Server.

It performs processing on the server side and can be used to generate dynamic web contents.

It is responsible for accepting web request and generating web response.

Why Servlet?

Server side processing can be done using various technologies like:

Microsoft Dot Net Framework
PHP
CGI
Node JS

Servlets became popular because they are written in Java and hence, they can inherit all the features of Java e.g.

Platform Independent
Secured

Robust
Architecture Neutral
etc.

Developing Servlets:

To implement Servlets, it is necessary to create a Java Class which is referred to as Servlet Implementation Class.

However unless the class makes use of any Servlet-API, it won't be considered as Servlet Implementation Class.

Servlet API:

It is a basic API required to implement Servlets.
It mainly consists of 2 packages:

1. javax.servlet
2. javax.servlet.http

Java SE is an implementation and not a specification.
Java EE is a specification and not an implementation.

The implementation is provided by 3rd party softwares called as Web Server and Application Server

The Servlet API mainly consists of following:

1. Servlet

   An interface that belongs to javax.servlet package and must be implemented directly or indirectly by a Servlet Implementation Class.

   Since this interface declares 5 methods, directly implementing it is not recommended because the Servlet implementation class has to implement all these methods.

2. GenericServlet

   A class that belongs to javax.servlet package and it already implements Servlet interface.

   Therefore instead of implementing Servlet interface, a Servlet implementation class may extend GenericServlet.

   This class is used to handle any type of request.
   It is used to make servlet protocol independent.

3. HttpServlet

A class that belongs to the javax.servlet.http package.
It is a subclass of GenericServlet.
In order to make Servlet specific for HTTP request and response, the Servlet implementation class extends HttpServlet.

What is HTTP?
HTTP stands for HyperText Transfer Protocol.
It is the most commonly used protocol in a web based application.

In general, protocols are of 2 types:

1. Stateless

When a protocol is not able to maintain the state between of the 2 requests, it is called Stateless Protocol.
E.g. HTTP

2. Stateful

When a protocol is able to maintain the state between the 2 requests, it is called Stateful Protocol.
E.g. FTP

Every Servlet mainly has 3 properties:

1. Class Name
   It specifies the actual name of the class.
2. Name
   Every servlet has a unique name. By default it is same as that of class name; however it can be different.
3. URL Pattern
   It specifies against which URL the servlet will get accessed.

   By default it is taken as the name of the servlet, but it can be different.

   Any value is permissible provided it begins with '/'

Once an application is built, it is necessary to host that application on the server. This process is known as Deployment.

Constituents of URL:

http://localhost:8080/JavaEE_WebApp/hello

Every URL is mainly composed of 5 constituents:

1. Protocol : http
2. IP Address : localhost
3. Port No : 8080
4. Context Root : JavaEE_WebApp
5. Resource Name : /hello

Day14:

Where Servlet specific configurations are stored?
Every Web Application built using Java EE platform contains a configuration file known as web.xml.

It is also called the Deployment Descriptor of the Application. It is the single file that handles all the application specific configurations e.g. Servlet specific configurations.

It is located under the ..../WEB-INF folder.

What is Hot Deployment?
Whenever any change is made to the application, it gets reflected automatically. There is no need to restart the server. This type of deployment is called Hot Deployment.

Is a web.xml file really necessary?

Until the 2.5 version of Servlet Specification, web.xml file was mandatory. Since 3.0 it is made optional because 3.0 and above support Annotation based metadata.

E.g.
@WebServlet annotation is applied at the class level to configure servlet name, url etc.

Life Cycle of Servlet:
The life cycle of Servlet consists of 3 stages:

1. Instantiation and Initialization

   When the servlet is requested for the very first time, it gets loaded by Web Container and once loading is finished, it gets instantiated.

2. Service

   When the servlet is requested, it needs to be processed and

response needs to be generated.

This phase gets encountered every time when the servlet is requested.

3. Destroy

When the servlet is no longer needed, its object is removed by container.

This generally happens under 2 circumstances:

1. When the server stops
2. When the application is undeployed.

To manage the life cycle, there are 3 life cycle methods available:

init()

Gets called only once. Can be overridden to perform initialization.

service()

Gets called every time when the servlet is requested.

destroy()

Gets called only once. Can be overridden to perform clean up operations.

How service() and doGet() methods are interlinked?

The life cycle methods: init(), service() and destroy() are declared inside a super interface: Servlet

In fact, the Servlet interface provides total 5 methods:

3 Life Cycle methods + getServletConfig() and getServletInfo()

GenericServlet class implements Servlet interface and overrides all the methods from Servlet interface except: service().

Hence if a servlet implementation class is inherited from GenericServlet, it must override service() method.

HttpServlet is a subclass of GenericServlet and it already overrides service() method. Hence if a servlet implementation class is inherited from HttpServlet, it is not necessary to override service() method.

The service() method from HttpServlet class identifies the type of incoming HTTP Requests.

HTTP Request can be of several types:

GET
POST
HEAD
PUT
DELETE
OPTIONS
TRACE

Depending upon the type, it calls the respective doXXX() method.
E.g.

GET => doGet()
POST => doPost()
PUT => doPut()

E.g.
When a servlet is requested by typing the URL in the browser's address bar, always HTTP GET request is made.

In that case, service() will invoke the doGet() method.

These doXXX() methods are already defined inside HttpServlet class. To provide our own logic for response generation, it is necessary to override them in the sub class.

Other Resources in Web Application:
A web application is composed of several resources.
E.g.
HttpServletRequest, HttpServletResponse

Apart from these, other resources come into picture like:

ServletConfig:
An interface from javax.servlet package.
Along with an object of Servlet Implementation Class, the web container creates one more object of type ServletConfig.

This object maintains configuration specific information of that servlet.

E.g.
Initialization Parameters

ServletContext:
An interface from javax.servlet package.
The object of type ServletContext is created for the entire

web application.

Therefore it can be used to maintain application specific information.

The interface provides several methods out of which 2 are frequently used:

- void setAttribute(String, Object)
  - Used to bind the information in name-value pairs.
- Object getAttribute(String)
  - Used to retrieve the information against its name.

Where does the method getServletContext() come from?
The method getServletContext() is declared by interface: ServletConfig

GenericServlet class actually implements 3 interfaces:

1. Servlet
2. ServletConfig
3. java.io.Serializable

Therefore method getServletContext() can be used with GenericServlet as well as its subclasses.

HTML Form Processing:
Typically, in a web application, end user provides some information using some HTML form.

Once SUBMIT button is clicked, that information is sent to the server and it needs to be processed.

This is possible by processing the HTML form.

Capturing Request specific data:
Whenever a SUBMIT button of HTML form is clicked, all the form specific information gets sent towards server in the form of name-value pairs.

To capture this information, HttpServletRequest type object is used. It provides a method: String getParameter(String)

This method accepts parameter name as String and returns parameter value as String

Linking HTML form with Servlet:
The linking between HTML form and servlet is done using 'action' attribute of 'form' tag.

The value of the 'action' attribute must be the URL of the servlet that needs to be linked excluding '/'.

TODOs:

Enhance CMS as per the following:

Create a HTML page that accepts information about Course and upon hitting submit, displays the details of that course.

Display the details using some HTML style
// just creating an HTML page.

Handling POST Request

Why POST and Why not GET?

While working with HTML form, a user may want to send some confidential information like:

- User Credentials (Username and Password)
- Credit Card / Debit Card No
- ATM PIN
- etc

In case of GET request, this information gets appended to the URL and it can be read easily through some malicious Java Script code.

To address this, it is necessary to make HTTP POST request.
To make POST request and handle the same, following steps are required:

1. Add 'method' attribute of 'form' tag with value set to 'POST'
2. Implement doPost() method in the respective servlet class.

GET	POST
Parameters are appended to URL	Parameters are sent with body of the page
There is a limit on data transfer: 8KB	There is no limit on data transfer
There is a limit on URL length: 255 characters	There is no limit on URL length.

When to use GET and POST?

1. GET is typically used to access static web pages like HTML
2. If servlet is requested using Browser's Address Bar or using HTML's anchor tag (<a href..) then always GET request is made.
3. If servlet is requested using HTML form without 'method' attribute, always GET request is made.
4. If servlet is requested using HTML form with 'method' attribute having value set to 'POST', then POST request is made.
5. POST is generally used while submitting form data, file uploads etc.

Collaboration:

When one component interacts with another component of the web application, it is known as Collaboration.

To implement Collaboration, javax.servlet package provides an interface: RequestDispatcher

The component which wants to collaborate, needs to obtain the reference of type RequestDispatcher

Obtaining RequestDispatcher:
It is possible to obtain RequestDispatcher using ServletRequest interface. It provides a method getRequestDispatcher() which accepts String as URL of the resource with which collaboration is to be done and returns a reference of type RequestDispatcher.

Using RequestDispatcher:

Once a RequestDispatcher is obtained, it can be used by its 2 methods:

forward(ServletRequest, ServletResponse)
include(ServletRequest, ServletResponse)

forward():
When invoked, control is forwarded to the target resource and the target resource generates the final response back to the client.

include():
When invoked, control is forwarded to the target resource, the response is captured from the target resource, the control gets returned back to the source resource and source

resource generates the final response back to the client.

In Collaboration, it is possible to attach extra information to the existing request and forward the request.

This is done by using setAttribute() method of ServletRequest interface.

Day15:

Session Management (State Management)

What is Session?

When client gets connected to server, there is a logical contract established between client and server that is known as Session.

E.g.

When the client logs in, ideally session should begin for that client and when that client logs out, the session should terminate.

Whatever operations are performed by client throughout multiple requests, there is a need to maintain the conversational state with the client.

This technique is known as Session Management or State Management or even Session Tracking.

It is required because HTTP is a Stateless protocol.

In general, One Session corresponds to One User.

One application can have many sessions
One session can have many requests

How is Session Management done?

There are 4 ways to handle Session Management:

1. URL Encoding (URL Rewriting)
2. Hidden Fields
3. Cookies
4. Servlet API - HttpSession

HttpSession:

An interface from javax.servlet.http package.
An object of type HttpSession is used to maintain user level or session level information.

It provides relevant methods:

void setAttribute(String, Object)
Object getAttribute(String)

Obtaining HttpSession:

A session can be obtained using HttpServletRequest interface.
It provides 2 methods:

HttpSession getSession()
HttpSession getSession(boolean)

The 1st method returns an existing session if available otherwise, creates a new one and returns the same.

The 2nd method works same as that of 1st if boolean 'true' is used.

In case of 'false', it returns the session only if it is available otherwise it does not create a new session and it returns 'null'

How HttpSession internally works?

Whenever a call is given to the method getSession(), Web Container always checks whether there is any Cookie coming from client or not which stores some session related information.

If Cookie is coming, server understands that this client is not a new client, it is an old client and has a session otherwise it creates a new session.

When a Web Container creates a new Session (Creates an object of type HttpSession), assigns a unique identity to that session which is known as Session ID in the form of String.

Once the Session ID and the actual session object are generated, web container stores them in a Map based collection.

Then, Web Container creates a Cookie (Simple Text File) and stores the Session ID into this cookie and sends the cookie back to the client through response.

When the client makes subsequent request, server tries to find whether any cookie is coming through request or not.
If it is coming, gets that cookie, reads Session ID from that and cross check with the ID available in the Map based collection.

Once found, returns the value which is the HttpSession object.

This means, by default HttpSession works on the top of cookies.

What if Cookies are disabled on Browser Side?

If cookies are disabled on browser side, to make application client independent, HttpSession can be combined with URL Encoding.

This is possible using encodeURL() method of HttpServletResponse interface.

Other methods from HttpSession:

boolean isNew()

Returns whether the session is new or not

setMaxInactiveInterval(long seconds)

Specifies the time in terms of seconds within which a client has to make a request in order to retain the session.
If no request is made, Session times out.

invalidate()

Used to terminate the session explicitly.

Steps to Interact with Database

1. Modify JdbcUtils by adding Class.forName()
   Explicitly loading the type4 driver
2. Place mysql connector JAR file into /WEB-INF/lib folder.
   Once placed, it gets added automatically into the CLASSPATH

JSP

JSP stands for Java Server Pages.
It is a component that runs on server side and used to extend the functionality of Web Server.

Used to generate dynamic web contents.

Servlet	JSP
It is mainly used to handle navigation and processing	It is mainly used to handle presentation

Why JSP:

Web Designers can create elegant web pages using Java Language Semantics without writing Java Code.

Even though a person who does not have much knowledge on Java, still can proceed with JSP.

However, it is possible to write a Java code inside JSP but it is recommended to have less Java Code as much as possible.

Life Cycle of JSP:

- During 1st Request:
  - The page gets loaded by a web container.
  - As soon as loading is finished, it is compiled and translated into Servlet.
  - It means that JSP is an abstraction over Servlet.
  - The time at which translation occurs, is known as Translation Time.

  - If JSP is modified, re-translation happens.

  - Web Container invokes jspInit() method.
  - Web Container invokes _jspService() method

- During subsequent requests:
  - Web Container invokes _jspService() method

- When Server is stopped or application is undeployed, Web Container invokes jspDestroy() method.

- The JSP life cycle methods are declared in 2 interfaces:
  - JspPage
    - An interface from javax.servlet.jsp.
    - It provides 2 methods: jspInit() and jspDestroy()
  - HttpJspPage
    - An interface from javax.servlet.jsp.
    - It is a sub interface of JspPage.
    - It provides a method : _jspService()

What can be written within a JSP page?

- It is possible to write 3 things within a JSP Page:
  1. All HTML Tags
  2. Any valid Java Code
  3. JSP Tags

JSP Tags:

JSP Tags are divided into 3 categories:

1. Directives

They are denoted by <%@ .... %> delimiters.

They are of 3 types:

a) page

Used to describe some information about the JSP Page.

It has several attributes:

i) session

Tells whether the page participates in a session or not. By default it is true.

ii) import

Allows to import public declarations from the package other than java.lang

iii) errorPage

Specifies the name of the JSP page to which control is to be diverted if the current JSP page has some Java code which fires an exception.

iv) isErrorPage

Specifies whether the JSP page is an error page or not. If so, can make use of predefined object 'exception' otherwise not. The default value is 'false'

b) include

Used to include a resource in the JSP page.

A resource can be HTML, Text File or JSP page.

It provides a single attribute: 'file'

c) taglib

2. Scripting Elements

They are meant for writing a Java Code inside JSP.

They are of 3 types:

a) Declaration

It is denoted by <%! ... %> delimiters

Used to declare variables as well as define methods.

b) Scriptlet

It is denoted by <% .... %> delimiters

Used to write any valid Java Code.

c) Expression

It is denoted by <%=...%> delimiters

Used to evaluate some expression.

Some rules / concepts about Declaration, Scriptlet and Expression:

a. Variables declared inside Declaration Section become Instance Variables of the container generated servlet class

b. Variables declared inside Scriptlet become local variables of the service() method of the container generated servlet class. It is because of whatever Java Code is written inside a Scriptlet, directly goes into the service() method. Therefore, it is not possible to define a method inside a Scriptlet.

c. Whenever any Expression is used, it directly gets processed inside the service() method of the servlet class.It is not possible to invoke methods using Expression if their return type is 'void'

3. Standard Actions

There are several tags available in JSP to perform specific tasks. These are called Standard Actions.

All JSP standard actions follow the format:

<prefix:suffix> where suffix is the actual tag name.

TODOs:

Enhance CMS as per the following:
Modify previous Servlet in such a way that it accepts course information coming from HTML and stores that information as a record into the database.
(Use DAO Pattern)

Create one more HTML page that accepts Course ID and when SUBMIT is clicked, hits another servlet and the servlet must fetch course information based upon Course ID and display that using some HTML Format.

Display appropriate message if Course ID does not exist and redirect to the same HTML page to re enter Course ID.
(Hint: RequestDispatcher)
(Use DAO Pattern)

Implicit Objects:

These are the objects available to the Page authors by default.
Every object's name is fixed, case sensitive and it is of some specific type.

Some of the frequently used implicit objects are:

1. request (HttpServletRequest)
2. response (HttpServletResponse)
3. session (HttpSession)

4. application (ServletContext)
5. out (javax.servlet.jsp.JspWriter)
6. exception (Throwable)
Available only when isErrorPage is set to 'true'

Standard Actions:

1. <jsp:useBean>
Allows to instantiate a Java Bean (Java Class)
It provides mainly 4 attributes:
a) id : Specifies ID of the bean. It has to be unique.
b) class : Specifies CLASS of the bean
c) type : Specifies the super type. If omitted, defaults to CLASS type.
d) scope : Specifies scope of the bean from one of the 4 possible values:
page, request, session, application
If omitted, it defaults to 'page'

2. <jsp:setProperty>
Allows to set values for the properties of the bean using setter methods.

It follows standard format:
<jsp:setProperty name = "<<bean-name>>"
property = "...property details...."/>

3. <jsp:getProperty>
Allows to get values for the properties of the bean using getter methods.
It follows fixed format:
<jsp:getProperty name = "<<bean-name>> "property="<<prop-name>>"/>

How useBean, setProperty, getProperty actions work?

Standard actions 'setProperty' and 'getProperty' are always to be used in conjunction with 'useBean'

When request comes on the server side, web container retrieves request parameter names and tries to match them with the target property names of the bean class.

These property names are identified using Java Reflection API.
In fact, web container uses this API.

When 'setProperty' is used with 'property="*"', web container tries to find setter methods in the bean class with matching Java's

naming conventions.

Once the method is found, it gets called by web container through Reflection API.

Whatever type conversions are required, e.g. conversion between String and int using Integer.parseInt(..), will be managed by Web Container.

When 'getProperty' is used, web container will do the same thing using Reflection API but the only difference is it invokes getter methods.

Options for setting properties using 'setProperty'

1. property="*"
   Used to set all the properties.
2. property = "<<prop-name>>"

Both these options mentioned above work only when following condition is satisfied:
Property names must match with Request parameter names.

By any chance if they mismatch, the 3rd option is used.

3. property = "<<prop-name>> param = "<<param-name>>
   E.g.

```
<jsp:setProperty name = "movieBean" property="title" param="t_title"/>
```

4. property = "<<prop-name>>" value = "<<value>>"/>
   Used to assign value directly.
   This value can be a hard coded value or an expression.
   E.g.

```
<jsp:setProperty name = "movieBean" property="title" value = "Life of PI"/>
```

```
<%
    String movieName = "Life of PI";
%>
<jsp:setProperty name = "movieBean"
        property="title" value = "<%=movieName%>"/>
```

Other Actions:

<jsp:forward>
Used to forward the request to the next resource e.g.

Servlet, JSP and so on.
E.g.
<jsp:forward page = "next.jsp"/>

Request is forwarded to 'next.jsp' and it generates the response.

<jsp:include>

Used to include the response of the next resource e.g. Servlet, JSP and so on.

E.g.
<jsp:include page = "next.jsp"/>

Request is forwarded to 'next.jsp', takes the response of that and comes back to the current page and then the response is generated.

Difference between <%@ include ... and <jsp:include...

The 'include' directive' includes the resource at translation time whereas the 'include' action includes the resource at request time.

Therefore, generally while including static pages like HTML, 'include' directive is used whereas in case of dynamic pages like JSP, 'include' action is used.

<jsp:param>

Used in conjunction with either 'forward' or 'include' to send additional parameters.

E.g.

```
<jsp:forward page = "next.jsp">
        <jsp:param name = "<<name>>" value = "<<value>>"/>
        <jsp:param name = "<<name>>" value = "<<value>>"/>
        <jsp:param name = "<<name>>" value = "<<value>>"/>
</jsp:forward>
```

Custom Tags in JSP:

Sometimes, depending upon the domain needs, it is necessary to create user defined tags. Such tags are called Custom Tags.

Limitations of useBean:

Although useBean can be used to instantiate Java Classes, it is capable of working only upon Value Objects (VO).

Any object that contains values and up to maximum, provides a functionality to set or get them is referred as Value Object.

If there is any other functionality available, developer is still required to write Scriptlets.

Custom tags eliminate the need for scriptlets.

Creating Custom Tags:

Whatever operation the tag is supposed to perform, needs to be intimated to the Web Container.

Web Container is responsible for processing the tags given by JSP specification but it does not understand about the custom tag. The creator of the tag must provide that logic which will be used by Web Container in order to process the tag.

This is done by creating a Java Class which is referred as Tag Handler Class.

Once a Tag Handler Class is created, it needs to be mapped with the name of the tag so that web container uses the class as and when the respective tag is encountered.

This is done inside a .tld file.
TLD: Tag Library Descriptor

This file holds all the information about the tag in XML format.

Creating Tag Handler Class:
In order to create Tag Handler Class, it is necessary to use Tag API which belongs to the package:

javax.servlet.jsp.tagext.

It provides mainly 2 APIs:

1. Tag

   It is an interface that can be implemented directly or indirectly by a Tag Handler Class
2. TagSupport

It is an implementation class of Tag interface.
Ideally, Tag Handler Class inherits this class.

In order to generate a response from this Tag Handler Class, an object of type JspWriter is to be obtained.

It is obtained using an object of type 'PageContext' known as 'pageContext'

Creating TLD file:

Unlike HTML, JSP or even .class files, there is no standard location of a TLD file. It can be kept anywhere.

Therefore it is necessary to tell Web Container about the location of TLD file.

It is done using the 'taglib' directive.

TODOs:

Enhance CMS as per the following:
Create a Custom Tag <show:allCourses> in such a way that when used in a JSP, it shows all the courses.

Hibernate:

What is Hibernate?

Hibernate is an Open Source Java based Framework that is used to handle Persistence Layer of an application.

Hibernate is a Framework and not a Specification.

Framework is a partial solution used to address common problems.

There are several Java based Frameworks available:

E.g.

- JUnit
- Hibernate
- Spring
- Spring Boot
- JSF
- Struts

In Java, prior to Hibernate there is only one API that is used to interact with Database which is JDBC.

Why Hibernate?

Hibernate is an abstraction over JDBC.
It reduces the boiler-plate code and adds simplicity.

Hibernate is based upon ORM principles. Therefore an application which uses Hibernate can take full advantage of those principles.

What is ORM?
ORM stands for Object to Relational Mapping.
It is the process of mapping domain specific objects to the relational database tables.

ORM is a set of principles.

1. Automated Persistence
   Performing CRUD operations without SQL.
   Performing CRUD operations without single JDBC code.
2. Full support for Object Modeling.
   Can make use of Composition, Inheritance, Polymorphism efficiently.
3. Full support for Query Operations which work upon
   Classes and Properties rather than Tables and Columns
4. Full support for Optimization
   Optimization is possible through efficient fetching, caching etc.

Any framework follows all these ORM principles is said to be an ORM Framework.

E.g

- Hibernate
- Toplink (by Oracle)
- EclipseLink (Eclipse)
- IBatis

Hibernate Core API:

The core API of Hibernate belongs to the package:

org.hibernate

The API mainly consists of following:

1. Configuration
   A class from org.hibernate.cfg package.
   It is used to configure Hibernate based upon some metadata.

2. Session
    An interface from org.hibernate package.
    A lightweight object that is used to handle CRUD operations.
    It is an abstraction over JDBC Connection.

3. SessionFactory
    An interface from org.hibernate package.
    Acts as a Factory for producing Session.
    A heavyweight object that's why it is recommended to have one per application.

4. Transaction
    An interface from org.hibernate package.
    Used to manage transactions.
    Maintains atomicity.

5. Query
    An interface from org.hibernate package.
    Used to perform query operations.

First Hibernate Example:

1. Environment Setup

To use Hibernate, it is necessary to add Hibernate specific libraries into the project's CLASSPATH.

These libraries are in the form of JARs which are bundled into a single ZIP file that can be downloaded from the internet.

Instead of manually downloading these libraries, it is better to get them through some tool.

There are 2 such tools available in market:
1. Maven
2. Gradle

Create a Maven Based project.

Every Maven Based Project contains a configuration file:
pom.xml
(Project Object Model)
All the project specific configurations are done inside this file.

To add Hibernate specific JARs through Maven, it is necessary to add Hibernate Core Maven Dependency.

How Maven adds JARs:

Maven maintains a bunch of JARs into a container called a Repository.

There are 3 types of Repositories:

1. Local
2. Central
3. Remote

Day17:

2. Create entity class

In the context of Hibernate, any class of which object gets stored as a record into the database table, is known as entity class.

3. Provide Mapping Metadata

Once an entity class is created, it is necessary to tell Hibernate about to which table it is going to be mapped, to which columns its fields are going to be mapped etc.

This is possible by providing a Metadata in one of the 2 forms:

a) XML
b) Annotations

In case of XML, the mapping file is termed as HBM file.
HBM: Hibernate Mapping

This file can have any name, but by convention it is:
<<Entity-Class-Name>>.hbm.xml.

In Maven based project, this file must be placed under "src/main/resources" folder.

In case of Hibernate Mapping, an entity class must declare at least one field which will be considered as an identity of the entity. This is similar to PRIMARY KEY constraint in database table.

4. Provide Configuration Metadata

Hibernate hides the complexity of the application by reducing

the boiler-plate code e.g. loading the driver, establishing the connection etc.

But in order to get it done from Hibernate, it is necessary to provide Database specific configuration metadata:
E.g. Driver Class Name, URL, Username, Password etc

In Hibernate, it is termed as CFG file.
CFG: Configuration

In Hibernate by default, configurations are taken from:
hibernate.cfg.xml

It is also possible to load the configurations from a file called as "hibernate.properties" but it's not much use.

In Maven based project, this file must be placed under "src/main/resources" folder.

5. Write a Main Program

a) Configure hibernate using Configuration class
When Configuration class is instantiated, Hibernate looks for the file: "hibernate.properties" and once found, loads all the configuration properties from that file.

By any chance, if configuration is present inside "hibernate.cfg.xml", it must be informed to Hibernate.
This is done using configure() method.

By any chance, if the XML file name is different, use overloaded configure() method.

b) Obtain SessionFactory:
Once Hibernate is configured, it is necessary to obtain SessionFactory.
This is done using buildSessionFactory() method of Configuration class.

c) Obtain Session:
Once SessionFactory is built, at any time a Session can be obtained. It is required to perform CRUD operations.
This is done using openSession() method of SessionFactory

d) Create an entity class object with some information that

needs to be added as a record into the database table.

e) Obtain Transaction:
In Hibernate, while performing any DML operation, it is necessary to obtain a Transaction.
This is done using beginTransaction() method of Session.

f) Persist (Store) the object of entity class.
To perform DML INSERT operation, the object of entity class needs to be stored.
This is done using save() method of Session.

g) Commit the Transaction:
Whatever object is stored in Session using save(), the changes are reflected into the database upon committing the transaction.
This is done using commit() method of Transaction.

h) Close the Session and SessionFactory

What is the role of Dialect?

In Hibernate based application, it is necessary to set the dialect property.

For several popular databases, Hibernate provides respective dialect implementation classes.

E.g.

MySQL 5: MySQLDialect
MySQL 8: MySQL8Dialect
Oracle 10g: Oracle10gDialect

This dialect is used by Hibernate to understand the database capabilities and behave accordingly.

Retrieving Data:
In order to retrieve data against ID, Session interface provides load() method.

There are several overloaded load() methods available. The most frequently used accepts 2 parameters :

1. java.lang.Class
   It indicates the Class type of the entity class.
   Tells Hibernate which class object is to be populated

after retrieval.

2. java.io.Serializable
   It indicates the ID.
   In Hibernate, the rule is the type of the field which is registered as ID, must be Serializable.

Loading of an entity object:

To load an entity object, Session interface provides load() method.
In Hibernate, loading takes place by 2 ways:

1. LAZY (Default)
   When Session.load() is invoked, Hibernate simply creates an object and returns it back; without connecting to database.

   This object returned by Hibernate is not the object of entity class; rather it is a proxy.

   It means when Lazy Loading occurs, Hibernate always generates a proxy.

   Unless and until client demands for the data, Hibernate does not hit the database.

   As soon as client calls a getter method on any field which is other than ID, Hibernate decides to initialize the proxy by connecting to database.

2. EAGER
   Sometimes there is a need to load the data eagerly.
   This means, as soon as Session.load() is invoked, Hibernate must hit the database and load the data immediately.

   This is possible by enabling EAGER loading.

   In this case, Hibernate does not create proxy.

   To enable EAGER loading, use 'lazy=false' in HBM file.

   How Proxies are generated?
   Typically, there are 2 algorithms to generate proxy:
   1. Containment
   2. Inheritance

It depends upon the Framework which approach is to be used. Hibernate goes for Inheritance.

Therefore, if an entity class is declared as 'final', Hibernate cannot generate proxy and it results into EAGER loading by default.

TODOs:

Enhance CMS as per the following:

Handle this by creating a Maven Project (Simple Project with main() method)

1. Use Hibernate to perform CRUD operations.
2. Configure Lazy or Eager loading and monitor the output.

Using Session Interface:

Hibernate's Session interface is primarily used to handle CRUD operations.

It provides relevant methods:

E.g.

save()	: DML INSERT
load()	: DQL SELECT
delete()	: DML DELETE
update()	: Not for DML UPDATE

Instance States:

The object (Instance) of an entity class is in one of the several states with respect to Persistence Context.

Persistence Context is the object that is solely responsible for handling CRUD operations.

E.g.

In JDBC it is Connection whereas in Hibernate it is Session

With respect to Persistence Context, there are 3 states:

1. TRANSIENT

When the object is not yet associated with the Session, it is in the TRANSIENT state. It does not have any identity.

2. PERSISTENT

When the object is associated with the Session, it enters into the PERSISTENT state. Now the object has its identity.

3. DETACHED

The session with which the object is associated, if gets closed, the object enters into DETACHED state.

Whenever any change made to the object, in order to reflect that change into the database, the object must be in the PERSISTENT state.

However, it might happen that an object is already detached. After detachment, some changes are made in that object. Now it might be required to reflect those changes into the database.

In this case, the detached object needs to be brought back into PERSISTENT state. This is done using update() method.

Hibernate Mapping and Configuration:
Hibernate mainly works upon 2 Metadata:
Mapping and Configuration.

It is possible to provide these metadata by 2 ways:

1. **Declaratively**
   Generally, makes use of XMLs for declarations.
2. **Programmatically**
   Generally makes use of Annotations and Java Code.

Hibernate supports 2 types of annotations:

1. **Hibernate Specific**
2. **JPA Specific**

JPA stands for Java Persistence API
It is a specification and it is implemented by several ORMs like Hibernate, Toplink, IBatis and so on.
In the context of annotations, entity specific class must be annotated with @Entity annotation
The entity class is mapped with the DB table using @Table annotation.
To declare an ID field, @Id annotation is used.
In order to configure Hibernate programmatically,
It provides a class called Environment that belongs to the org.hibernate.cfg package.

It provides several constants (public, static and final variables) to simplify configurations.

In case of Annotation based metadata, @Entity and @Id annotations are mandatory.

Whereas @Table and @Column are optional.
@Table if omitted, by default refers to the class name.
@Column if omitted, by default refers to the field name.

Day18:

HQL:

HQL stands for Hibernate Query Language

In order to load the data from Database, Session interface provides a load() method.

The load() method loads an entity based upon ID. But if, multiple data values (records) are to be loaded, then load() method cannot serve that requirement.

To address this, Hibernate provides a Query Language known as HQL. It is to be used through an API known as Query interface.

HQL works with Classes and Properties rather than Tables and Columns.

It makes use of Clauses, Functions, Expressions and so on.

There are mainly 2 clauses:

1. FROM

   Used to load a list of entity objects. This is especially useful when all the details are to be fetched.

   If some of the details are to be fetched, it leads to performance implications. To address this, Hibernate provides SELECT clause.

2. SELECT

   Used to retrieve specific specific fields of an entity with high performance.

   The returned List contains an array of type Object rather then the objects of entity classes.

This reduces the overhead of loading unwanted data unnecessarily, ultimately resulting into performance optimization.

Sometimes depending upon the requirement, it is necessary to use a SELECT clause but in the returned List, objects of some class are required rather than array of type Object.

This is possible using Constructor Expression.

HQL Limitation:

Although HQL is well placed to perform query operations to fetch multiple records, it still needs basic query syntax.

Developers who do not have much background of querying may face difficulties in using HQL.

To address this problem, Hibernate provides a mechanism that allows developers to perform query operations without writing a query.

This is possible using another API known as Criteria API.
It mainly provides an interface: Criteria

Once a Criteria is obtained, by default it fetches all the records.
To fetch selected records, restrictions are to be added on that Criteria.

TODOs:

Enhance CMS as per the following:

Use HQL to handle following operations:

1. Show all courses
2. Show long duration courses. (Display only the course names)
   (Definition of Long Duration Course is of your choice)
   (Hint: Use SELECT clause)
3. Show courses with High Cost (Display course name and provider)
   (Definition of High Cost is of your choice)
   (Hine: Use Constructor Expression)

Spring Framework:

Spring is a Java based framework that is used to handle several

aspects of a business application.

E.g.

Enterprises Services
Support for Data Access
Support for Web

Like Java EE, Spring follows Component Driven Architecture.
It provides life cycle management support for the components even though they are POJOs.

POJO: Plain Old Java Object
Any simple class which is independent is referred as POJO.
E.g.
public class User { ... }

Features of Spring:

1. Life cycle management support for POJOs.
2. Dependency Injection
3. Aspect Oriented Programming (AOP)
4. Data Access
5. MVC

First Spring Example:

1. Environment Setup

   Like Hibernate, Spring is a Framework which provides several libraries in the form of JARs.

   These JARs can be downloaded manually or through some build tools like Maven or Gradle.

   Create a Maven project and add Spring specific dependencies.

2. Create some Interface:
   Spring Framework promotes Programming by Interfaces.
   It is not mandatory but in some cases it is.

3. Create an implementation class of that interface.

4. Provide Configuration:

   In order to get managed the life cycle of a component through Spring, it is necessary to declare that component to Spring's environment: Container

This is done by providing configuration metadata using 2 options:

a) XML

b) Annotations

In case of XML, it can be a file with any name but by convention, it is "spring-config.xml".

5. Write main program:

Since components are registered as beans in the configuration unit, first it must be told to Spring about the configuration unit so that Spring can find the components within that unit.

In order to handle these activities, Spring Framework provides an interface: ApplicationContext.

It has several implementations. The one which is frequently used is FileSystemXmlApplicationContext.

It is used to load the configurations from XML file located in the file system.

Once context is built, at any time a request can be made to Spring using getBean() method. It accepts an ID of the bean as String and returns Object associated with that ID.

Types of ApplicationContext:

Spring provides several types of ApplicationContext.

E.g.

FileSystemXmlApplicationContext:
Used to load bean configuration from XML file available in the file system.

ClassPathXmlApplicationContext:
Used to load bean configuration from XML file available in the CLASSPATH of the project.

In a Maven based project, if a file is placed under: "src/main/resources", it gets added into the CLASSPATH of the project.

Whenever any bean is registered in the configuration unit, if the bean class declares any attribute, it must be configured so that Spring can inject value for the relevant attribute for that

particular object.

This is possible by using Dependency Injection.
Dependency Injection is a technique which ensures that values for the dependencies to be injected.

This is possible by 2 ways:

1. Setter Injection
   Values for the attributes are injected using Setter methods.
2. Constructor Injection
   Values for the attributes are injected using Parameterized constructors.

Sometimes, a bean may have dependency on another bean or several beans.

In this case, Spring first has to create the inner bean, inject it into the main bean and return the main bean back to the client.

The process of injecting a bean into another bean, that one into the next one and so on is referred to as Bean Wiring.

To implement Bean Wiring, either 'ref' attribute or'ref' element is used.

Instantiation of a Bean:
In Spring, instantiation of a bean can take place by 2 ways:

LAZY
   Object is created by Spring only when client demands it.

EAGER (DEFAULT)
   Object is created by Spring when the ApplicationContext is built.

To enable LAZY, use [lazy-init="true"] in XML.

Bean Scopes:
A bean registered in the configuration unit of Spring has some scope.

In Spring, these scopes are of 5 types:

a) singleton (Default)
   It is the default scope.
   Even if the bean is requested multiple times,Spring

creates a single object only for the first time.
Subsequent times, it simply returns its reference.

b) prototype
It is a contrast to singleton.
Spring creates a new object for every request.

c) request
This is applicable only in case of Spring MVC.
The bean gets associated with HttpServletRequest.
The bean gets destroyed as soon as response is generated.

d) session
This is applicable only in case of Spring MVC.
The bean gets associated with HttpSession.
The bean survives even though a response is generated. The bean gets destroyed when the session is over.

e) global-session
This is applicable only in case of Spring Portlet environment

To modify the scope, use [scope = "<<scope-type>>"]

TODOs:
Enhance CMS as per the following:
Use Spring container to register the beans of type Course using various DI techniques.

Day 19:

Annotation Based Configuration:
In Spring, beans can be configured using annotations as well.
This enables quick start the applications.

Annotation based configuration is divided into 2 types:
a) Java Based Configuration
b) Pure Stereotype Annotation Based Configuration

Java Based Configuration:
Since XML does not get used at all, the entire configuration metadata is provided by writing a Java class which is referred as configuration specific class or configuration unit.

Once a class is created, it must be declared as a Configuration Unit to Spring.

This is done using @Configuration annotation.
It is applied at the class level.

Once a configuration unit is created, it can hold several bean configurations.

This is done by adding a method which returns an object of a type of the bean which is to be configured.

Once a method is defined, it must be informed to Spring that this method is not an ordinary method; rather it is a bean creation method.

This is done using @Bean annotation.
It is applied at the method level.

By default the bean gets registered against the ID which is taken as the method name.

Loading Beans:
Since the configuration is done using annotation rather than XML, Spring provides a separate class to load beans from that configuration.

It is AnnotationConfigApplicationContext.

Once AnnotationConfigApplicationContext is instantiated, it is necessary to register the Configuration Unit with that AnnotationConfigApplicationContext.

This is done by calling register() method of AnnotationConfigApplicationContext.

Once the registration is done, changes are not reflected.
To reflect the changes, it is necessary to refresh the context using refresh() method.

Enabling LAZY Loading:

In annotation based configuration it is done using @Lazy annotation.

Providing Bean Scope:

In annotation based configuration it is done using @Scope annotation.

Bean Wiring:

It is the process of resolving dependencies.

E.g.

Bean A depends upon Bean B. In this case, Bean B needs to be wired with Bean A.

In Spring, Bean Wiring is of 2 types:

1. Explicit

   Which bean is to be wired is mentioned in the configuration unit explicitly.

2. Implicit

   Which bean is to be wired is not mentioned in the configuration unit; rather Spring figures it out of its own.

   In Spring, this type of implicit bean wiring is known as: Auto-Wiring

   The use of Auto-Wiring is limited. If the requirement is beyond that limit, it needs to be done explicitly.

   To implement Auto-wiring, Spring provides @Autowired annotation.

   It is applied at the field or setter method level of the field for the property that is to be auto-wired.

How Auto-wiring works:

Spring identifies the type of the property which is being auto-wired and checks for the configuration of the bean with that type in the configuration unit.

Once found, wires that bean.

If bean is not found, it results into an exception because when @Autowired is used, by default the behavior is MANDATORY.

The behavior can be changed to OPTIONAL by using

'required=false'
In this case, if the bean is not found, the property remains unwired.

If multiple beans of that type are found in the configuration unit, it creates ambiguity and results in an exception.

This ambiguity can be resolved by 2 ways:

1. Using @Primary
   Used in the configuration unit
2. Using @Qualifier
   Used in the bean class

Pure Stereotype Annotation Based Configuration:

In the first approach which is Java Based Configuration, beans are configured and eventually they get managed by Spring.

However, in order to configure the beans, developer still needs to use the 'new' operator to create the object explicitly.

The second approach can be used to configure beans without 'new' operator. In fact, Spring uses Reflection API to build the object and this is used in most of the Spring based projects.

To handle this, Spring provides an annotation:
@Component

Applied at the class level to indicate that this class is not an ordinary class; rather is a component that is to be managed.

Beans configured using stereotype annotations like @Component can be obtained either by using the CLASS type of the bean or ID.

Once a class is annotated with @Component, the configuration the unit must be aware about that.

In order to register the classes annotated with stereo-type annotation like @Component, they need to be scanned.
This process is known as Component Scanning.

This is done using an annotation: @ComponentScan.
It is applied at the class level of the configuration specific class.

If classes belong to different packages, then in order to get them scanned, the package names must be mentioned with @ComponentScan.

However, if classes reside inside sub-packages, they get scanned just using the name of the super package.

This is because @ComponentScan not only picks up the classes annotated with @Component from that package but also from its sub packages.

Retrieving beans using ID:
Even though classes are annotated with @Component, their beans can be accessed using ID as String rather than CLASS type.

By default, the bean gets registered by lowercasing the first character of the bean class name.

However, it is possible to override this default setting by supplying ID as a parameter to @Component

TODOs:

Enhance CMS as per the following:

Try configuring the bean of type Course using Annotation based configurations through both the options:

1. Java Based
2. Stereo Type Based

AOP:

AOP stands for Aspect Oriented Programming

AOP is a programming pattern that allows to separate Cross Cutting Concerns from the Business Logic.

Any logic that cuts across several components (points) is referred as a Cross Cutting Concern. It is also called Secondary Concern.

Benefits:

Since secondary concerns are decoupled from primary concerns, there will not be any code repetition.

Application can adopt changes without modifying the existing code.

Developers can focus now upon the core business logic.

Spring's AOP module allows to decouple the secondary concerns from primary concerns.
It is done by providing a separate class that is known as aspect.

AOP Terminologies:
Spring's AOP module comes up with several terminologies:

1. Advice
2. Joinpoint
3. Pointcut
4. Aspect
5. Weaving

1. Advice
   It defines WHAT and WHEN of an aspect.
2. Joinpoint
   It specifies the probable locations where an aspect CAN BE plugged-in.
3. Pointcut
   If advice defines WHAT and WHEN of an aspect, pointcut defines WHERE.
   It specifies the exact locations where an aspect IS to be plugged-in.
4. Aspect
   It is the merger of Advice and Pointcut.
   It defines WHAT, WHEN and WHERE
5. Weaving
   When the target bean method is invoked, it needs to be intercepted so that Spring can handle the aspect logic.
   This is done by some mediator known as Proxy.

   This proxy is generated by Spring AOP container and the the process of this proxy generation is called weaving.

   In general, weaving takes place at 3 intervals:
      1. Compile time
      2. Classload time
      3. Runtime

   Spring AOP goes for Runtime Weaving

Types of Advices:

1. Before Advice
   It gets encountered before the execution of the target bean method.
2. After Returning Advice
   It gets encountered after the successful execution of the target bean method.
3. After Throwing Advice
   It gets encountered when some exception is raised during the execution of the target bean method.
4. After Advice
   It gets encountered irrespective of whether the target bean method executes successfully or not.
5. Around Advice
   It is not a new advice; rather it is a single advice that is merger of all other advice implementations.

AOP Implementation:

Step 1:

Spring AOP module is well placed with a 3rd party library which is known as AspectJ. Therefore, to implement Spring AOP, it is necessary to install AspectJ. It is done by adding 2 dependencies:

a) aspectjweaver
b) aspectjrt

Step 2:

Spring AOP mandates Programming by Interface. Therefore it is necessary to create an interface.

Step 3:

Create implementation class(es) of the interface. These classes are considered as Target classes.

Step 4:

Create aspect specific classes. This can be any Java class with @Aspect annotation. Any class with @Aspect annotation is considered as aspect specific class. Define advices by adding relevant annotations at the method level.

Define a pointcut by adding relevant annotation at the method level.

Typically this is done by using some designator. The most commonly used designator is 'execution'.

Step 5:

Configure the beans: Target beans and Aspect specific beans in the configuration unit.

Step 6:
Enable proxy support so that Spring can generate proxy. This is done by using @EnableAspectJAutoProxy annotation.
It is to be applied at the configuration specific class level.

Step 7:
Write a main program.

How does Spring generate a proxy?
Spring generates a proxy using Containment.

Around Advice:

It is not any separate advice; rather it is a merger of all 4 advice.

In Around Advice, since all the methods are invoked at same place, there has to be a call made to the target bean method explicitly.

This is possible using ProceedingJoinPoint with proceed() method. When invoked, from aspect class, control goes to the target Method.

Day20:

Spring MVC:

MVC:
MVC stands for Model-View-Controller
It is a commonly used design pattern for building web based applications.

M - Model:
It is responsible for handling Business Logic of the application.
It is responsible for storing domain specific data as well.
Typically, in a web based application it is implemented using either a simple Java Class (Java Bean) or EJB.

V - View
It is responsible for handling Presentation Logic of the application.
Typically, in a web based application it is implemented using HTML or JSPs.

C - Controller
It is responsible for handling the navigation and workflow.
Typically, in a web based application it is implemented using Servlets

That's why Java EE is designed keeping MVC in mind.
It has mainly 3 components:

- Servlet (Controller)
- JSP (View)
- EJB (Model)

MVC pattern can be implemented by 2 ways:

1. MVC 1 (MVC Model 1)
2. MVC 2 (MVC Model 2)

MVC 1:
In MVC 1, all the 3 aspects are handled using JSPs, discarding Servlets at all.

This type of model has several drawbacks e.g. Tight Coupling, Difficult to re-factor etc.

This kind of architecture is called Page Centric.

MVC 2:
In MVC 2, all the 3 aspects are handled separately.
Model using Java Beans or EJBs, View using JSPs, and Controller using Servlets.

This type of model has several benefits e.g. Loose Coupling, Easy to re-factor etc.

This kind of architecture is called Servlet Centric.

Spring MVC:
Spring Framework provides a separate module known as Spring MVC.
It is used to build Web Based Applications.

Spring MVC is based upon MVC 2 architecture.
Spring MVC follows one more pattern : Front Controller

Front Controller is a single controller servlet which gets called

for any request made to that application.

In Spring MVC, this Front Controller is predefined.
It is DispatcherServlet.

It needs to be configured in such a way that any incoming request will hit this controller first.

Spring MVC Major Components:

1. Front Controller
2. Controller
3. Model
4. View
5. View Resolver

1. Front Controller:

   It is DispatcherServlet which is given by Spring. It is just required to be configured.

2. Controller:

   It is a user defined component that is responsible for processing the request.
   In Spring MVC, it is handled by using a simple Java Class.

3. Model:

   It is a user defined component that is responsible for handling the business logic.
   In Spring MVC, it is handled by using a simple Java Class.

4. View:

   It is a user defined component that is responsible for handling the presentation logic.
   In Spring MVC, it is handled by using JSPs.

   Spring MVC supports other view technologies as well
   e.g. Freemarker, Velocity, Excel, PDF

5. View Resolver:

   It is a predefined component that is responsible for resolving views.

   Spring MVC provides a wide range of view resolvers. The most commonly used is:
   InternalResourceViewResolver.

Spring MVC Application:

To use predefined libraries e.g. DispatcherServlet, InternalResourceViewResolver, it is necessary to add Spring WebMvc Maven Dependency.

Since Spring MVC is meant for building web applications, it is necessary to create a Maven Based web project.

Step 1:

Create a Maven based web project and set up the required things like Java version, Target runtime etc.

Step 2:

Add a Maven Dependency for Spring Web MVC.

Step 3:

Configure Front Controller: DispatcherServlet
In Spring MVC, it is necessary to configure the front controller DispatcherServlet in such a way that it will handle all the incoming requests.

There are 2 ways to configure Front Controller:

1. Using web.xml
2. Using Java Class (Programmatic)

If web.xml is not present, it is possible to configure Front Controller by writing a Java Class.

This Java Class must handle bootstrapping that means the Front Controller must be configured at the time of application is loaded.

This is made possible using an interface: WebApplicationInitializer

Therefore the Java class has to implement WebApplicationInitializer.

Step 4:

Configure View Resolver: InternalResourceViewResolver
Spring MVC makes use of View Resolver to resolve the view.
There are 2 ways to configure View Resolver:

1. Using XML Configuration File
2. Using Java Class through Annotation

In case of XML configuration file, the configuration entry is by default loaded from XML file having a name in the following format:
[name of DispatcherServlet]-servlet.xml

In case of Java based configuration, it is configured as a bean using @Configuration and @Bean annotations.

However, sometimes it is necessary to catch life cycle events in a Web Application. To handle this, Spring MVC provides an interface: WebMvcConfigurer

Typically, the Java class implements this interface.

Step 5:

Provide a Controller:
In Spring MVC, controller is not a Servlet; rather it is a simple Java Class.

Once a class is created, it must be informed to Spring MVC that this class is a Controller implementation class.

This is done by using @Controller annotation.
It is applied at the class level to mark that class as a Controller implementation class.

Once the controller class is created, it must define some method that is meant for request handling.

The method must return the name of the view as 'String'.

Once the method is defined, it must be told to Spring MVC that this method is not an ordinary method; rather it is meant for Request Handling or Request Processing.

This is done by using @RequestMapping annotation.
It is applied at the method level.

Form Processing

In a Web Application, end user can send information via HTML form.

This information needs to be processed on server side

and dynamic views need to be generated.

In order to accomplish this, it is necessary to process the form.

When a data is sent by the client, it is to be bound with the variables declared in the signature of the request handling method.

To bind this information, @RequestParam annotation is used. It is equivalent to request.getParameter() in Servlet.

The annotation is applied at the method parameter level.

R & D: Is @RequestParam really necessary?

TODOs:

Enhance CMS as per the following:

Implement the following using Spring MVC.

Creates 2 JSPs:

1. Add new course

   Must accept the details of the course and after addition, render some response indicating Addition Successful.

2. Show Course

   Must accept the ID of the course and render the response with details of that course.

   Render another response if ID does not exist.

Note: Use In-memory structure for Store courses.

Day21:

REST:

In recent years, web development has changed drastically. Several organizations, business applications opt for building inter-operable services which are referred as Web Services.

What is a Web Service?

Web Service is a software system designed to support inter-operable machine-to-machine interaction over the network.

The service provider can implement the service without knowing the type of consumer.

Web Services are of 2 types:

1. SOAP Based

It uses XML as a medium for data transfer and the protocol used for communication is HTTP.

HTTP combined with XML = SOAP
SOAP: Simple Object Access Protocol

2. RESTful

REST stands for REpresentational State Transfer.
It is another way of developing web services.
It mainly focuses upon the application specific data and works with plain HTTP.

The consumer of REST API can consume the resources in any format that is suitable to the consumer.

There are several formats available but the most commonly used format is JSON which stands for:
Java Script Object Notation

Implementing REST API:

Several popular languages like Java, DOT NET, Python and so on provide their own way of implementing REST API.

Implementing REST API using Java:

There are several options available to build REST API using Java.

a) Jersey Framework

An open source Java based framework used to build REST APIs.

b) Spring MVC Framework:

REST APIs can be built efficiently using Spring MVC capabilities.

This is especially used when REST resources are to be combined with typical Spring MVC based web application.

c) Spring Boot Framework

Spring Boot framework is an extension to Spring Framework. It is the most popular framework used when the focus is only upon REST APIs.

Spring Boot provides several features:

1. Easy to use
2. Minimal Configuration
3. Provides a default embedded Tomcat Server (Container)
4. Provides several starters in order to bootstrap the application.

Building REST API using Spring Boot:

Spring Boot provides several options to build REST API

1. Spring Boot Starters
   Uses some starter project to build REST API.
   Comes by default STS (Spring Tools Suite)

2. Spring Boot CLI
   A command line interface for building REST API

3. Spring Initializer
   A web interface for building REST API
   (start.spring.io)

Every Spring Boot project by default provides a single class with main() method and the class is annotated with @SpringBootApplication annotation.

Spring Boot mainly provides 2 APIs:

1. @SpringBootApplication
2. SpringApplication: A Java Class

1. @SpringBootApplication:

It is meant for handling the configuration unit of Spring Boot for declaring beans.

It is a combination of 3 annotations:

@Configuration
@ComponentScan
@EnableAutoConfiguration

2. SpringApplication:

It is a class that is used to kick start the application.

Changing the Port No:

In order to configure Spring Boot project e.g. changing the port number etc, the project provides a configuration file: "application.properties" which is located under:

"src/main/resources"

Creating REST API:

To create REST API, it is necessary to create a Java Class that is meant for providing REST API or Implementations.

Once the class is created, it must be informed to Spring Boot that this class is not an ordinary class; rather it is a Restful Controller.

This is done by using @RestController annotation.
It is applied at the class level.

It is provided since Spring 4.x that eliminates the need of @ResponseBody because it is a merger of @Controller and @ResponseBody.

@RestController = @Controller + @ResponseBody

Single JSON

```
{
	"name":"Earth",
	"moons":1
}
```

Array of type JSON

```
[
	{
		"name":"Earth",
		"moons":1
	},
	{
		"name":"Mars",
		"moons":2
	},
	{
		"name":"Jupiter",
		"Moons":16 } ]
```

How Java Object is converted into JSON?

The conversion between Java to JSON takes place via some API known as Jackson API.

Handling HTTP Requests:

In order to create endpoints, @RequestMapping annotation is user.

Depending upon the type of the HTTP request to be handled, while working with @RequestMapping, an enum: RequestMethod is to be used.

In case of HTTP GET request, it is not required because it is the default type of HTTP request.

It is especially required in case of other types
e.g. POST, PUT, DELETE etc.

This can be simplified by using other several annotations which are meant for specific types of HTTP requests.

E.g.

@GetMapping:	Used to handle GET request
@PostMapping:	Used to handle POST request
@PutMapping:	Used to handle PUT request
@DeleteMapping:	Used to handle DELETE request

Creating a Service:

In REST API, business logic is taken care by a service implementation class.

This class is used by Restful Controller. The reference of Service Implementation is to be injected into Restful Controller.

For this injection, the Service Implementation must be managed by Spring Container. This is done using @Service annotation.

http://localhost:9090/restaurant-api --> All restaurants

http://localhost:9090/restaurant-api/4

Accepting Parameters:
A REST API may have to accept additional parameters with the URL.

E.g.
Loading something based upon ID
To create an end point for the URL which accepts a parameter, following format is used.

/<<URL>>/{param-name}

Binding the request parameter with the variable:
Whatever value is present in the request parameter denoted by '{ }', needs to be bound with the receiving parameter of the method.

This is done by using @PathVariable annotation.
It is applied at the parameter level.

How to get JSON into Java Object?
Whatever JSON data is coming from client, needs to be bound with the corresponding Java Object.

Since JSON data is traveling towards server via HTTP Request, in order to bind it, @RequestBody annotation is used.

It is to be applied at the parameter level.

Interaction with Database:

In real business applications, data is stored, loaded, deleted or even updated by performing operations towards Database.

To interact with Database, REST API can use several options:

1. JDBC
2. Hibernate
3. JPA

The way Hibernate is an abstraction over JDBC, JPA is an abstraction over Hibernate.

Increasing the level of abstraction is always beneficial because it adds more flexibility, loose coupling.

Therefore in most cases, REST API interacts with Database

using JPA.

However, still developer is required to write some code for handling CRUD operations.

This can be further taken away from the developer, adding more simplicity.

It is possible using Spring Data JPA.
It provides a standard library which abstracts away all the code for CRUD operations.

The main API provided is JpaRepository.
It is an interface and its implementation is provided by Spring Boot.

In order to use JpaRepository, Spring Data JPA dependency must be added in the Maven Project.

JpaRepository is already a Managed Component. Hence it can be injected into Service using Autowiring.

Handling HttpStatus:
While building REST API, the implementation must be done from the perspective of Consumer.

It is always a good practice to return correct HTTP response back to the consumer.

E.g.
If consumer tries to make a GET request against ID and if the relevant resource is not available, it is better to generate HTTP Status as 404 : Not Found.

In order to handle this, @ResponseStatus annotation is used.
