Unit 7

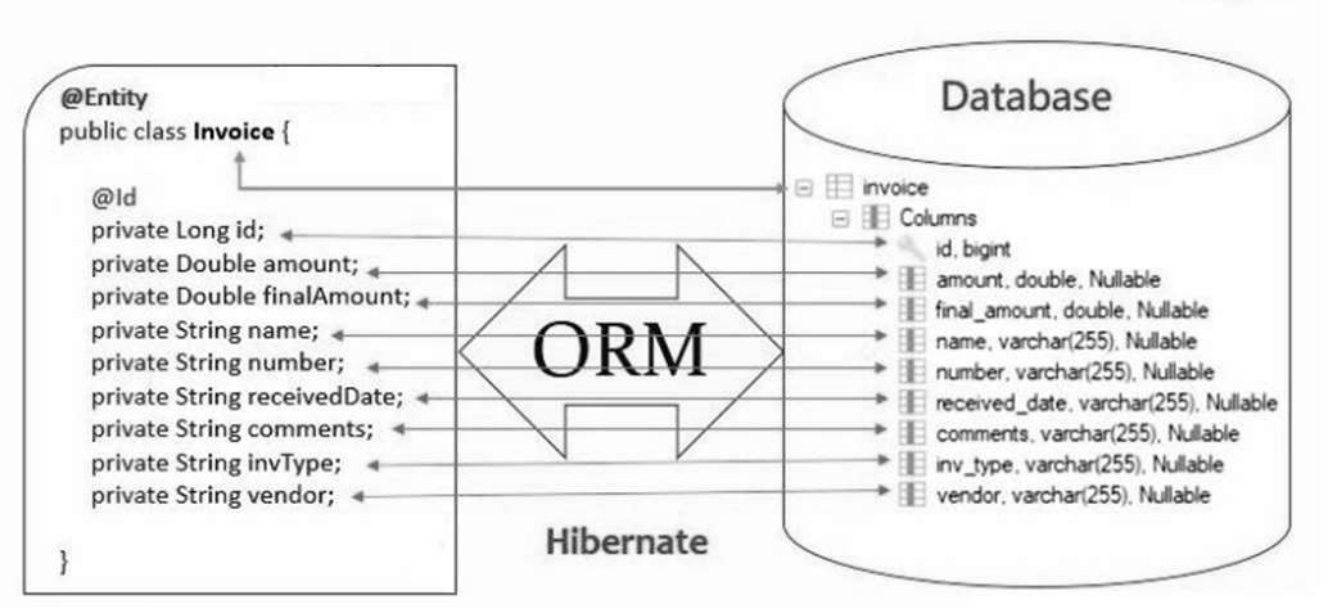
Advanced Topics in Java 7hrs

7.1 Overview of ORM

An ORM, or Object Relational Mapper, is a piece of software designed to translate between the data representations used by databases and those used in object-oriented programming. Basically, these two ways of working with data don't naturally fit together, so an ORM attempts to bridge the gap between the two systems' data designs.

From a developer's perspective, an ORM allows you to work with database-backed data using the same object-oriented structures and mechanisms you'd use for any type of internal data. The promise of ORMs is that you won't need to rely on special techniques or necessarily learn a new querying language like SQL to be productive with your data.

In general, ORMs serve as an abstraction layer between the application and the database. They attempt to increase developer productivity by removing the need for boilerplate code and avoiding the use of awkward techniques that might break the idioms and ergonomics that you expect from your language of choice.



Some of the commonly used ORM frameworks in Java are:

- Hibernate (Popular among all other)
- EclipseLink
- Apache OpenJPA
- MyBatis
- Spring Data JPA

7.2 Hibernate

Hibernate ORM is an object–relational mapping tool for the Java programming language. It provides a framework for mapping an object-oriented domain model to a relational database.

Pros:

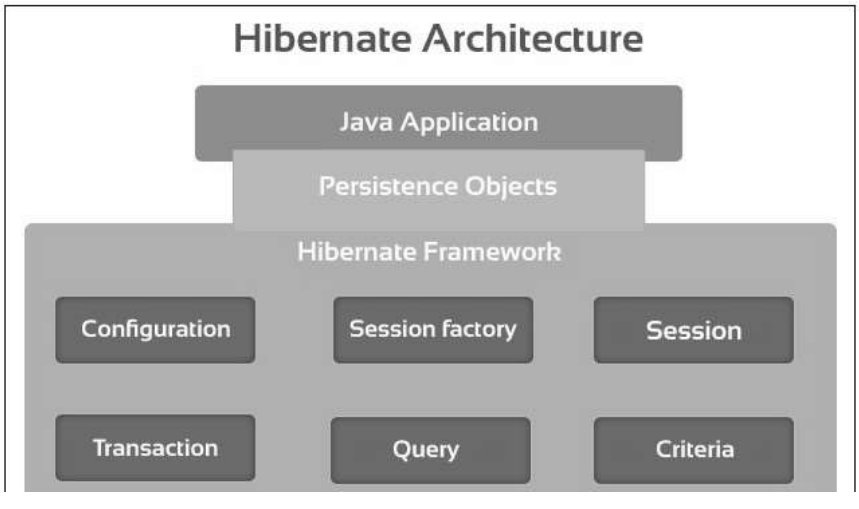
- Hibernate enables you to communicate with any database by making tiny alternations in code
- MySQL, Db2 or Oracle, Hibernate is DB independent
- Caching instrument to bug catalog with same queries
- N+1 or Sluggish loading support
- Low risk of data loss and it requires less power

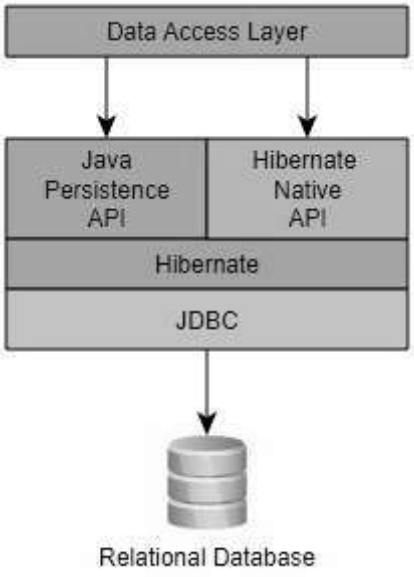
Cons:

- If power goes off, you can lose all your data
- Restarting can be extremely slow

Architecture

Hibernate, as an ORM solution, effectively "sits between" the Java application data access layer and the Relational Database, as can be seen in the diagram above. The Java application makes use of the Hibernate APIs to load, store, query, etc its domain data.





As a JPA provider, Hibernate implements the Java Persistence API specifications and the association between JPA interfaces and hibernate specific implementations can be visualized in the following diagram:

Hibernate Components

SessionFactory (org.hibernate.SessionFactory)

A thread-safe (and immutable) representation of the mapping of the application domain model to a database. Acts as a factory for ***org.hibernate.*Session** instances. The **EntityManagerFactory** is the JPA equivalent of a **SessionFactory** and basically those two converge into the same **SessionFactory** implementation. A **SessionFactory** is very expensive to create, so, for any given database, the application should have only one associated **SessionFactory**. The **SessionFactory** maintains services that Hibernate uses across all Session(s) such as second level caches, connection pools, transaction system integrations, etc.

Session (org.hibernate.Session)

A single-threaded, short-lived object In JPA nomenclature, the Session is represented by an EntityManager.Behind the scenes, the Hibernate Session wraps a JDBC java.sql.Connection and acts as a factory for org.hibernate.Transaction instances. It maintains a generally "repeatable read" persistence context (first level cache) of the application domain model.

Transaction (org.hibernate.Transaction)

A single-threaded, short-lived object used by the application to demarcate individual physical transaction boundaries.EntityTransaction is the JPA equivalent and both act as an abstraction API to isolate the application from the underlying transaction system in use (JDBC or JTA).

Hibernate Annotations

Annotation	**Package Detail/Import statement**
@Entity	import javax.persistence.Entity;
@Table	import javax.persistence.Table;
@Column	import javax.persistence.Column;
@Id	import javax.persistence.Id;
@GeneratedValue	import javax.persistence.GeneratedValue;
@OrderBy	import javax.persistence.OrderBy;
@Transient	import javax.persistence.Transient;
@Lob	import javax.persistence.Lob;

Hibernate Association Mapping Annotations	
@OneToOne	import javax.persistence.OneToOne;
@ManyToOne	import javax.persistence.ManyToOne;
@OneToMany	import javax.persistence.OneToMany;
@ManyToMany	import javax.persistence.ManyToMany;
@PrimaryKeyJoinColumn	import javax.persistence.PrimaryKeyJoinColumn;
@JoinColumn	import javax.persistence.JoinColumn;
@JoinTable	import javax.persistence.JoinTable;
@MapsId	import javax.persistence.MapsId;

@Table

Specify the database table this Entity maps to using the name attribute of @Table annotation. In the example below, the data will be stored in 'company' table in the database.

```
@Entity
@Table(name = "company")
public class Company implements Serializable {
	...
}
```

@Column

Specify the column mapping using @Column annotation to change the column name.

```
@Entity
@Table(name = "company")
public class Company implements Serializable {

	@Column(name = "name")
	private String name;

	...
}
```

@Id

Annotate the id column using @Id for primary key field

```
@Entity
@Table(name = "company")
public class Company implements Serializable {

	@Id
	@Column(name = "id")
	private int id;

	...
}
```

@GeneratedValue

Let database generate (auto-increment) the id column.

```
@Entity
@Table(name = "company")
public class Company implements Serializable {

	@Id
	@Column(name = "id")
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	private int id;

	...
}
```

@OrderBy

Sort your data using @OrderBy annotation. In example below, it will sort all contacts in a company by their firstname in ascending order.

```
@Entity
@Table(name = "company")
public class Company implements Serializable {

	@Id
	@Column(name = "id")
```

```
	@GeneratedValue
	private int id;

	@OrderBy("firstName asc")
	private Set contacts;
	...........

}
```

@Transient

Annotate your transient properties with @Transient.

@Lob

Annotate large objects with @Lob.

Hibernate Configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
          "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
     <session-factory>
          <property
name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property
>
          <property name="hibernate.connection.url">
               jdbc:mysql://localhost:3306/db</property>
          <property name="hibernate.connection.username">root</property>
          <property name="hibernate.dialect">
               org.hibernate.dialect.MySQLDialect</property>
          <property name="hibernate.hbm2ddl.auto">update</property>
          <property name="hibernate.show_sql">true</property>
          <property name="hibernate.format_sql">true</property>
          <property name="hibernate.connection.pool_size">2</property>
          <mapping class="entity.Student" />
     </session-factory>
</hibernate-configuration>
```

Hibernate SessionFactory Class

```
package utils;
```

```
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtils {
	private static SessionFactory sessionFactory;
	public static SessionFactory getSessionFactory(){
		if(sessionFactory!=null) {
			return sessionFactory;
		}
		Configuration cfg = new Configuration();
		sessionFactory = cfg.configure().buildSessionFactory();
		return sessionFactory;
	}
}
```

Entity class for CRUD operations

```
package entity;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name="tbl_students")
public class Student {

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private Long id;

	@Column(nullable = false, unique = true,length = 30,
name="first_name")
	private String firstName;

	private String lastName;

	public Student() {
	}
	public Long getId() {
		return id;
	}

	public void setId(Long id) {
```

```
		this.id = id;
	}

	public String getFirstName() {
		return firstName;
	}

	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}

	public String getLastName() {
		return lastName;
	}

	public void setLastName(String lastName) {
		this.lastName = lastName;
	}

	public Student(String firstName, String lastName) {
		super();
		this.firstName = firstName;
		this.lastName = lastName;
	}

}
```

Create record or save record to database

```
Student s = new Student("Mr Sunil Bahadur", "Bist");
try(Session session = HibernateUtils.getSessionFactory().openSession()){
	session.getTransaction().begin();
	//save to database
	session.persist(s);
	session.getTransaction().commit();
}catch(Exception e) {
	System.out.println(e);
}
```

Update Record from a database

```
try(Session session = HibernateUtils.getSessionFactory().openSession()){
	session.getTransaction().begin();
	Student s = session.get(Student.class, 1);
	s.setFirstName("Mr. Sunil");
	session.persist(s);
	session.getTransaction().commit();
```

```
}catch(Exception e) {
	System.out.println(e);
}
```

Delete Record from a database

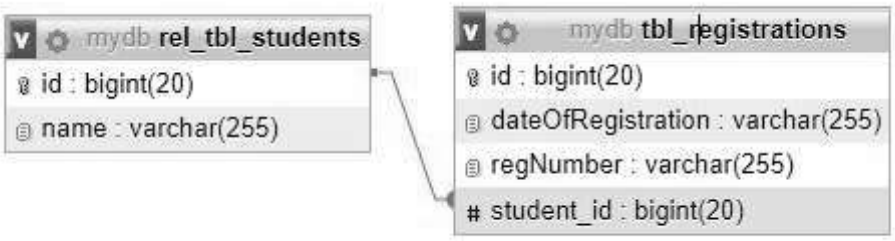
```
try(Session session = HibernateUtils.getSessionFactory().openSession()){
	session.getTransaction().begin();
	Student s = session.get(Student.class, 1);
	session.remove(s);
	session.getTransaction().commit();
}catch(Exception e) {
	System.out.println(e);
}
```

Select all records

```
try(Session session = HibernateUtils.getSessionFactory().openSession()){
	session.getTransaction().begin();
	List<Student> students = session.createQuery("SELECT s FROM Student
s", Student.class).getResultList();
	for(Student std: students) {
		System.out.println(std.getFirstName());
	}
	session.getTransaction().commit();

}catch(Exception e) {
	System.out.println(e);
}

//OR
try(Session session = HibernateUtils.getSessionFactory().openSession()){
	//List<Student> students =
session.createCriteria(Student.class).list();
	//List<Student> students =(List<Student>) session.createQuery("FROM
Student").list();
	CriteriaBuilder builder = session.getCriteriaBuilder();
	CriteriaQuery<Student> criteria = builder.createQuery(Student.class);
	criteria.from(Student.class);
	List<Student> students=
session.createQuery(criteria).getResultList();
	for(Student s: students){
		System.out.println(s.getFirstName());
	}
```

```
		System.out.println("Successful...");
	}catch(Exception e){
		System.out.println("Error On Reading All: "+e.getMessage());
			}
		}
	}
```

One to One Relation



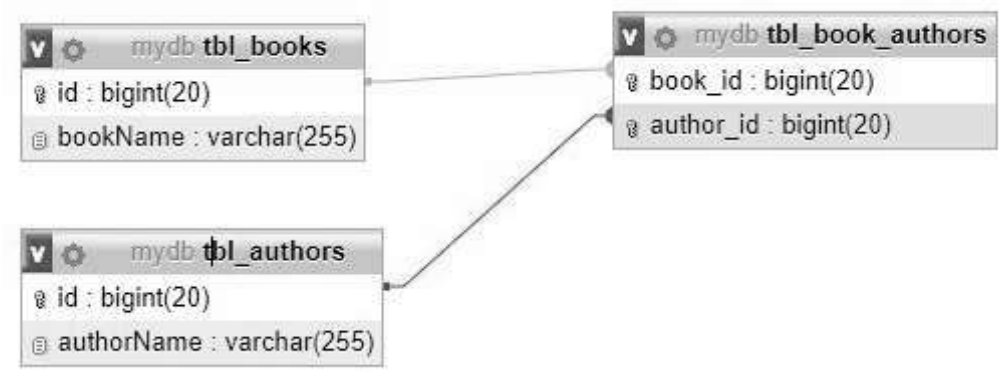

```
@Entity
@Table(name="tbl_registrations")
public class Registration {
	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private Long id;
	private String regNumber;
	private String dateOfRegistration;

	@OneToOne(cascade = CascadeType.ALL)
	@JoinColumn(name="student_id", nullable = false,referencedColumnName
= "id")
	private Student student;

}

@Entity
@Table(name="rel_tbl_students")
public class Student implements Serializable {
	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private Long id;
	@Column(nullable = false)
	private String name;

	@OneToOne(mappedBy = "student")
	private Registration registration;
```

```
}
```

Entity for One-to-Many Relationship mapping


```
@Entity
@Table(name="tbl_students")
public class Student implements Serializable {
	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private Long id;
	@Column(nullable = false)
	private String name;

	@OneToMany(mappedBy = "student", cascade = CascadeType.ALL)
	private List<Address> address;
}
@Entity
@Table(name="tbl_address")
public class Address implements Serializable {

	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private Long id;

	private String country;

	@ManyToOne
	@JoinColumn(name="student_id", nullable = false, referencedColumnName
= "id")
	private Student student;
}
```

Entity for Many-to-Many Relationships mapping




```
@Entity
@Table(name="tbl_books")
public class Book implements Serializable {

	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private Long id;
	private String bookName;

	@ManyToMany(cascade = { CascadeType.ALL })
	@JoinTable(
		name = "tbl_book_authors",
		joinColumns = { @JoinColumn(name = "book_id") },
		inverseJoinColumns = { @JoinColumn(name = "author_id") }
		)
	private Set<Author> authors = new HashSet<>();
}

@Entity
@Table(name="tbl_authors")
public class Author implements Serializable {
	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private Long id;
	private String authorName;

	@ManyToMany(mappedBy = "authors")
	private Set<Book> books = new HashSet<>();
}
```

7.3 Web Framework Introduction

In software development, a framework is a tool that provides a basic structure and support for building robust software applications. You can think of a framework as a sturdy scaffolding that provides structure and support to the entire software development process.

Its main purpose is to simplify and streamline the tedious task of building applications. Frameworks come with pre-built libraries, tools, and APIs that address common programming challenges. They provide standardized approaches and conventions for tasks such as web development and database interactions. By using frameworks, programmers can save time by avoiding repetitive code and focusing on writing the core logic of their applications. Frameworks also help maintain consistency and increase productivity by providing best programming practices.

Overall, frameworks streamline the development process and make it easier to create scalable and efficient applications.

Some popular java programming frameworks are as:

- Spring
- Spring MVC
- Spring Boot
- Hibernate
- Apache Struts
- Thymeleaf
- Grails
- Google Web Toolkit (GWT)
- Vaadin etc

7.4 Basics of Spring Boot

Overview of Spring Boot

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".

We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration.

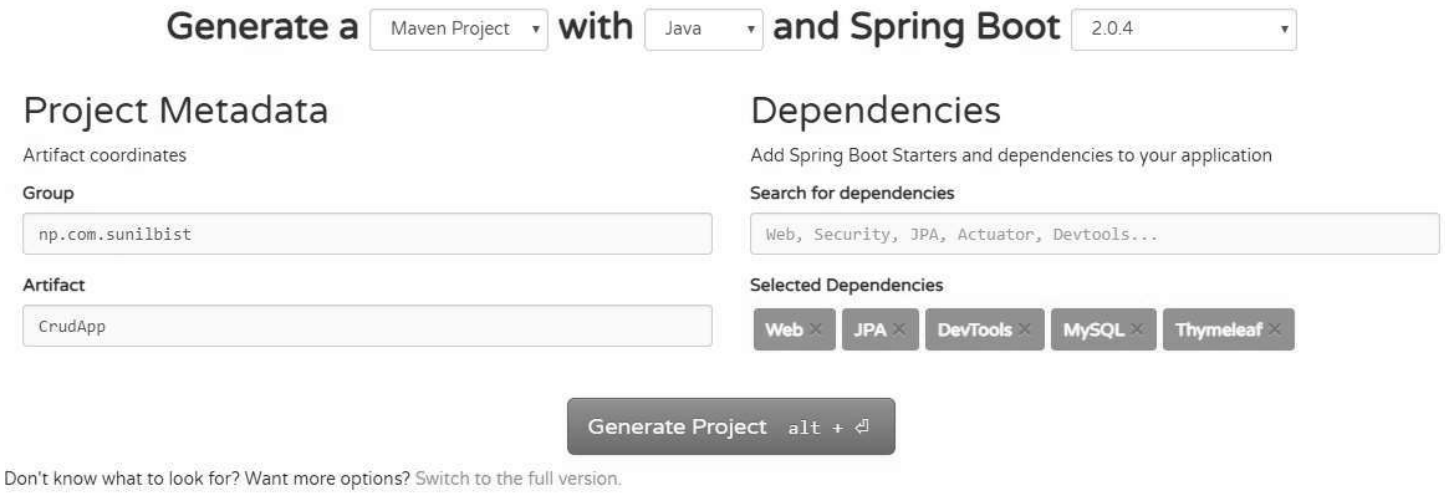
Features

- Create stand-alone Spring applications
- Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
- Provide opinionated 'starter' dependencies to simplify your build configuration
- Automatically configure Spring and 3rd party libraries whenever possible

- Provide production-ready features such as metrics, health checks and externalized configuration
- Absolutely no code generation and no requirement for XML configuration

Bootstrap your application with Spring Initializers

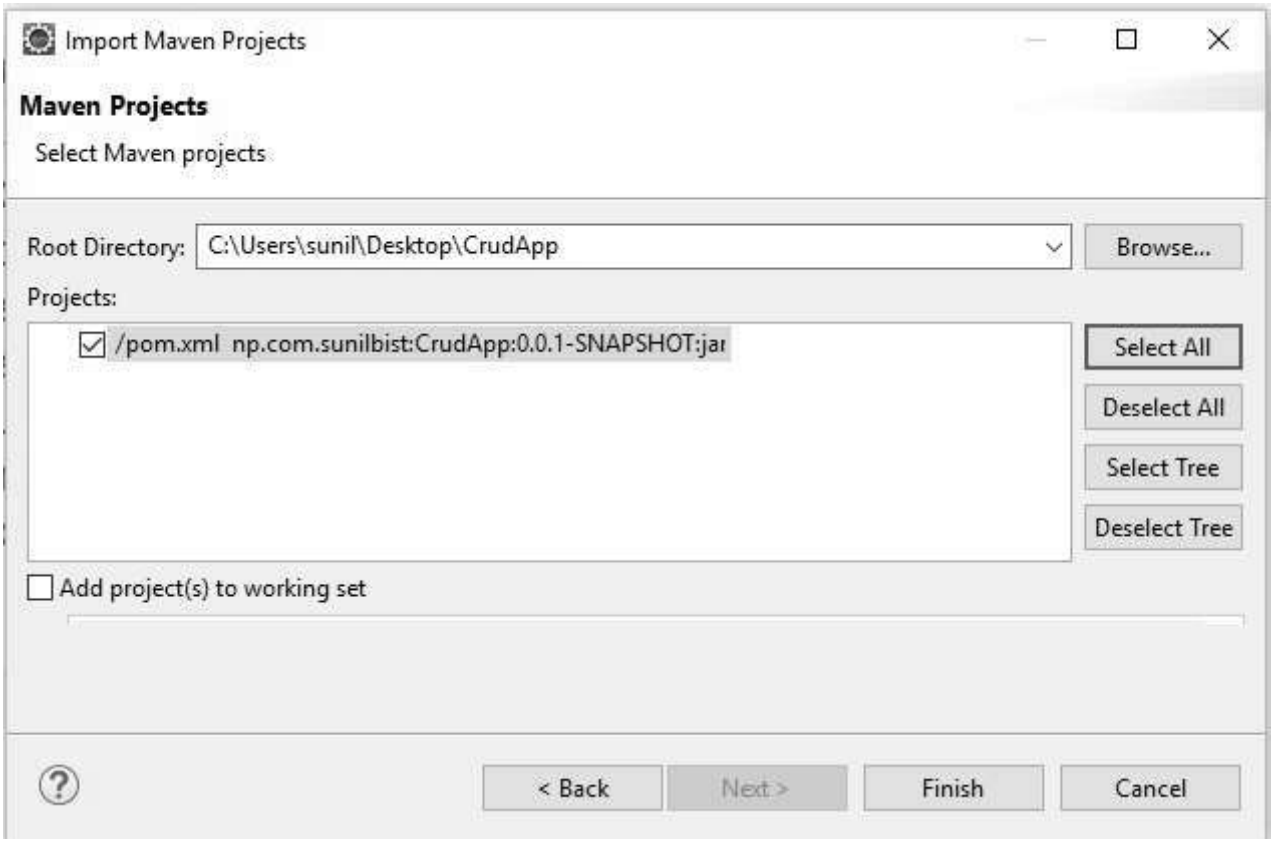
Step 1: Go to https://start.spring.io/ and you will see a screen and make changes as shown below



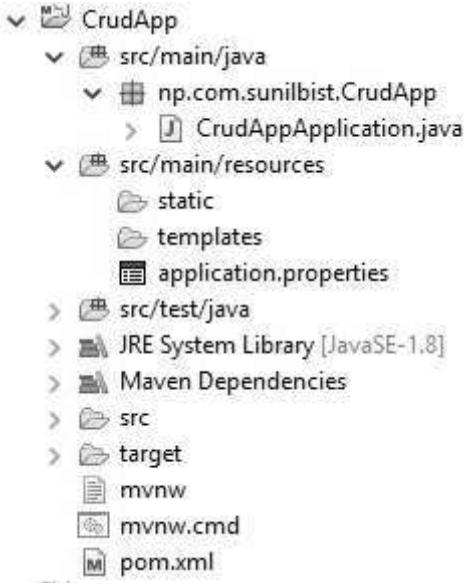
Step 2: Download and Extract to your desired location for my case I am extracting to desktop

Step 3: Now open your JavaEE eclipse version and import maven existing project as below.

Note: - This will require internet connection to your computer because it will download all the required libraries from maven repository from the remote server to your local computer. Once all the required libraries are downloaded next time you do not need to connect for the same jar file. It will search from your .m2 folder under your user name folder.



When down load completes its folder, structure looks like below screen

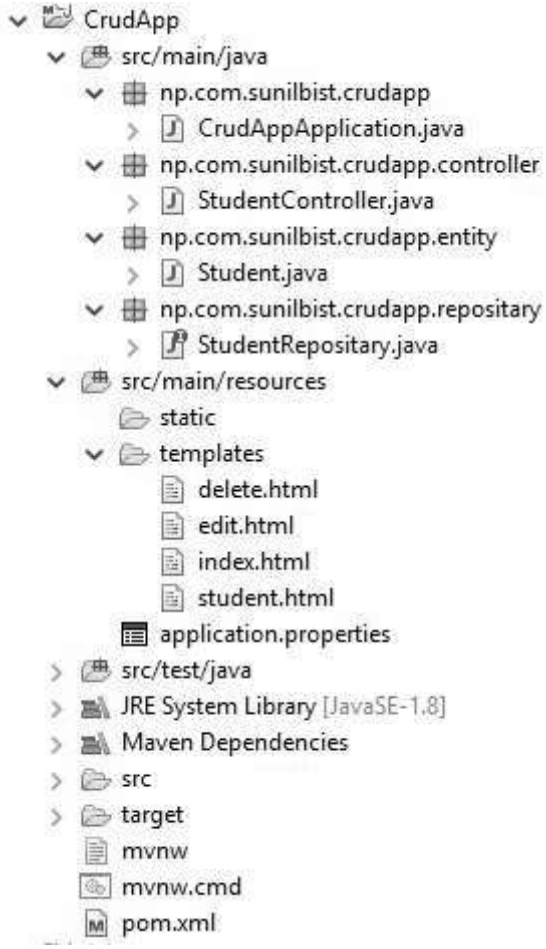


Step 4: - Now Setup your database and hibernate properties in the ***application.properties*** file as below: -

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/crudapp
spring.datasource.username=root
```

```
spring.datasource.password=
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto = update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.thymeleaf.cache=false
```

Change your package name all in lowercase and create various packages as shown in figure below



Student.java

```
package entity;
import javax.persistence.*;
@Entity
public class Student {

	@Id
	@GeneratedValue (strategy=GenerationType.AUTO)
	private int id;

	private String name;
```

```
	public Student() {
	}

	public void setId(int id) {
		this.id = id;
	}
	public int getId() {
		return id;
	}
	public void setName(String name) {
		this.name = name;
	}
	public String getName() {
		return name;
	}
}
```

StudentRepositary.java

```
package repositary;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import np.com.sunilbist.crudapp.entity.Student;

@Repository
public interface StudentRepositary extends CrudRepository<Student, Integer>
{

}
```

StudentController.java

```
package controller;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;
import np.com.sunilbist.crudapp.entity.Student;
import np.com.sunilbist.crudapp.repositary.StudentRepositary;


@Controller
@RequestMapping("/student")
public class StudentController {
```

```
	@Autowired
	private StudentRepositary studentRepo;

	@GetMapping("/list")
	public String list(Model model){
		model.addAttribute("studentList",studentRepo.findAll());
		return "index";
	}

	@GetMapping("/add")
	public String add(Model model){
		model.addAttribute("title", "New Student");
		model.addAttribute("student", new Student());
		return "student";
	}

	@PostMapping("/add")
	public String add(@ModelAttribute("student") Student student){
		studentRepo.save(student);
		return "redirect:/student/list";
	}

	@GetMapping("/edit/{id}")
	public String edit(@PathVariable("id") Integer id, Model model){
		model.addAttribute("student", studentRepo.findById(id));
		return "edit";
	}

	@PostMapping("/edit")
	public String edit(@ModelAttribute("student") Student student){
		studentRepo.save(student);
		return "redirect:/student/list";
	}

	@GetMapping("/delete/{id}")
	public String delete(Model model, @PathVariable("id") Integer id){
		model.addAttribute("student", studentRepo.findById(id));
		return "delete";
	}

	@PostMapping("/delete")
	public String delete(@ModelAttribute("student") Student student){
		studentRepo.delete(student);
		return "redirect:/student/list";
	}
}
```

CrudAppApplication.java

```
package crudapp;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class CrudAppApplication {

	public static void main(String[] args) {
		SpringApplication.run(CrudAppApplication.class, args);
	}
}
```

index.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="ISO-8859-1">
<title>Student List</title>
</head>
<body>
	<a th:href="@{/student/add}">Add New</a>
	<table border="1">
		<thead>
			<tr>
				<th>#</th>
				<th>Name</th>
				<th>Action?</th>
			</tr>
		</thead>
		<tbody>
			<tr th:each="s : ${studentList}">
				<td th:text="${s.id}"></td>
				<td th:text="${s.name}"></td>
				<td><a
th:href="@{'/student/delete/'+${s.id}}">Delete</a> <a

	th:href="@{'/student/edit/'+${s.id}}">Edit</a></td>
			</tr>
		</tbody>
		<tfoot>
			<tr>
```

```
                    <th>#</th>
                    <th>Name</th>
                    <th>Action?</th>
                </tr>
            </tfoot>
        </table>
</body>
</html>
```

student.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="ISO-8859-1">
<title th:text="${title}">Student</title>
</head>
<body>
        <h3 th:text="${title}"></h3>
        <hr />
        <form action="#" method="post" th:action="@{/student/add}"
              th:object="${student}">
              <input type="text" th:field="*{name}" />
              <button type="submit">Save</button>
        </form>
</body>
</html>
```

edit.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="ISO-8859-1">
<title th:text="${title}">Edit</title>
</head>
<body>
        <h3 th:text="${title}"></h3>
        <hr />
        <form action="#" method="post" th:action="@{/student/edit}"
              th:object="${student}">
              <input type="hidden" th:field="*{id}" />
              <input type="text" th:field="*{name}" />
              <button type="submit">Update</button>
```

```
      </form>
</body>
</html>
```

delete.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="ISO-8859-1">
<title th:text="${title}">Delete</title>
</head>
<body>
      <h3 th:text="${title}"></h3>
      <hr />
      <form action="#" method="post" th:action="@{/student/delete}"
            th:object="${student}">
            <input type="hidden"  th:field="*{id}" /> <input type="hidden"
                  th:field="*{name}" />
            <h3>Are you sure you want to delete?</h3>
            <button type="submit">Yes</button>
            <a th:href="@{/student/list}">No</a>
      </form>
</body>
</html>
```

Important note: - Don't forget to create database to your mysql database crudapp and update application.properties file based on your mysql database credentials

Now its time to run and test your CrudApp

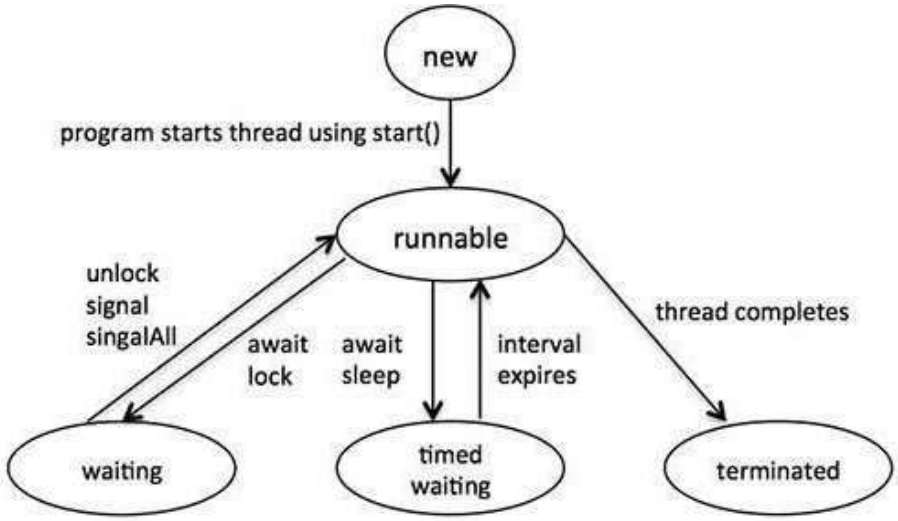
Right click on **CrudAppApplicaiton java** file and Run as Java Application and you it works everything is good to go

7.5 Concurrency and Multithreading in JAVA

A program in execution is known as process. An instance of process is known as *thread. Multithreading* is a process that can run concurrently with other processes. In other words, multithreading is the mechanism of running more than one thread concurrently. In a multithreaded program each thread can handle different task at the same time making optimal use of the available resources. By definition multitasking is when multiple processes share common processing resources such as a CPU. Multithreading extends the idea of multitasking into applications where we can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application. It makes the best usage of the CPU and keeps it busy maximum time.

Thread Life Cycle:

A thread goes through the various stages of the life cycle. It has stages such as new stage or born stage, running stage, waiting stage and dead stage. It is as explained in the diagram given below;



- **New:** It is a stage where a new thread begins its life cycle. It remains in this state until the program starts the thread. It is also referred to as a born thread or born state.
- **Runnable:** It is the second state of the thread life cycle, after a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task. In simpler terms, a thread actually executes in this state.
- **Waiting:** There may be a condition where sometimes a thread in transitions may have to wait while the thread executes another thread, so this state is known as waiting state. A thread transitions may come back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state, transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated:** A runnable thread enters the terminated state when it completes its task or otherwise terminates. It is also known as dead state.

Creating multithreaded application:

In java multithreaded applications can be created by using two processes. One process is to implement Runnable interface and other process is to extend Thread class. Both of them are explained as follows in detail.

Multithreading (Method-1) by implementing Runnable interface:

```
public class FirstThread implements Runnable{
     @Override
     public void run() {
```

```
            for(int i=1;i<4;i++){
                  System.out.println("First Thread : " + i);
                  try{
                        Thread.sleep(1000);
                  }catch(Exception e){
                        System.out.println("First Thread exception
occurred:");
                  }
            }
      }
}

public class SecondThread implements Runnable {
      public void run(){
            for(int j=1; j<4; j++){
                  System.out.println("Second Thread: "+ j);
                  try{
                        Thread.sleep(2600);
                  }catch(Exception e){
                        System.out.println("Second Thread exception
occurred:");
                  }
            }
      }
}


public class SimpleThread{

      public static void main(String args[]){
            FirstThread ft=new FirstThread();
            Thread t0=new Thread(ft);
            t0.start();

            SecondThread st = new SecondThread();
            Thread t = new Thread(st);
            t.start();
      }
}
```

Above program invokes the constructor at first which starts a first thread and makes it sleep for the 1000 milliseconds. While this process is in sleeping state the other thread invoked by the start() method is executed. It then sleeps for the 2600 milliseconds by invoking sleep() method. While this process is in sleep it switches back to the other free thread and the process continues until all the threads terminate the processing.

Output:

```
Second Thread: 1
First Thread: 1
First Thread: 2
First Thread: 3
Second Thread: 2
Second Thread: 3
```

Multithreading (Method-2) by extending Thread class:

```
public class A extends Thread{
	public void run(){
		for(int i=1;i<4;i++){
			System.out.println("Thread A: " + i);
			try{
				Thread.sleep(1000);
			}catch(Exception e){
				System.out.println("Thread A exception occurred:");
			}
		}
	}
}
```

Save above code as A.java. Open new file in notepad and type following code and save it as B.java;

```
public class B extends Thread{
	public void run(){
		for(int i=1;i<4;i++){
			System.out.println("Thread B: " + i);
			try{
				Thread.sleep(2600);
			}catch(Exception e){
				System.out.println("Thread B exception occurred:");
			}
		}
	}
}

public class ThreadDemo{
	public static void main(String args[]){
		A aa=new A();
		B bb=new B();
		aa.start();
		bb.start();
	}
}
```

Save this class as simpleThread.java and run this class which contains main method. It works similar

to the runnable program as stated above but the working mechanism is slightly different. Here, we invoke both threads from the main methods by calling start() method.

Output:

```
Thread A: 1
Thread B: 1
Thread A: 2
Thread A: 3
Thread B: 2
Thread B: 3
```

Thread Priorities:

Every java thread has a priority that helps the operating system determine the order in which threads are scheduled. Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Suppose we need to set priority in above example then we can set priorities as follows;

```
aa.setPriority(Thread.MIN_PRIORITY);
bb.setPriority(Thread.MAX_PRIORITY);
```

by setting above priorities the bb object of B class has maximum priority and the object aa of A class has the least priority. Instead of using the constants MIN_ PRIORITY, MAX_ PRIORITY and NORM_PRIORITY we could use the integer value between 1 to 10. 1 being the lowest priority and 10 being the highest priority. It could take the following forms;

```
aa.setPriority(1);
bb.setPriority(10);
```

Thread Synchronization:

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource concurrently and finally, they can produce unexpected results. For example; if multiple threads try to write within a same file then they may corrupt the data because one of the threads can overwrite data, or while one thread is opening the same file at the same time another thread might be closing the same file. So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor. Java programming language provides a very handy way of creating threads and synchronizing their task by using **synchronized** blocks. We keep shared resources within this block. Following is the general form of the synchronized statement:

```
synchronized(objectidentifier){
```

```
// Access shared variables and other shared resources
}
```

Here, the **objectidentifier** is a reference to an object whose lock associates with the monitor that the synchronized statement represents. For example; when threads are not synchronized, they print counter value which is not in sequence, but when we print counter by putting inside synchronized() block, then it prints counter very much in sequence for both the threads.

```
public class ThreadDemo{
      public void printCount(){
            try{
                  for(int i=5;i>0;i--){
                        System.out.println("counter: "+i);
                  }
            }catch(Exception e){
                  System.out.println("Exception created");
            }
      }
}


public class ThreadTest extends Thread{
      private Thread t;
      private String threadName;
      private ThreadDemo td;
      ThreadTest(String name, ThreadDemo td){
            threadName=name;
            this.td=td;
      }
      public void run(){
            synchronized(td){
                  td.printCount();
            }
            System.out.println("Thread: "+ threadName + " Existing");
      }
      public void start(){
            System.out.println("thread: " +threadName + " starting");
            if (t==null){
                  t=new Thread(this, threadName);
                  t.start();
            }
      }
      public static void main(String args[]){
            ThreadDemo td1=new ThreadDemo();
            ThreadTest t1=new ThreadTest("Thread-1",td1);
            ThreadTest t2=new ThreadTest("Thread-2",td1);
            t1.start();
```

```
            t2.start();
        }
}
```

7.6 Design Patterns: Singleton, Factory and Abstract Factory

Software design patterns are general, reusable solutions to common problems that arise during the design and development of software. They represent best practices for solving certain types of problems and provide a way for developers to communicate about effective design solutions.

Design Patterns gained popularity after the book Design Patterns: Elements of Reusable Object-Oriented Software was published in 1994 by Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm (also known as Gang of Four or GoF).

There are three types of Design Patterns:

- Creational Design Pattern
- Structural Design Pattern
- Behavioral Design Pattern

Creational Design Pattern abstract the instantiation process. They help in making a system independent of how its objects are created, composed and represented.

Structural Design Patterns are concerned with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to compose interfaces or implementations.

Behavioral Patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that's difficult to follow at run-time.

Creational Design Patterns are concerned with the way in which objects are created. They reduce complexities and instability by creating objects in a controlled manner.

The *new* operator is often considered harmful as it scatters objects all over the application. Over time it can become challenging to change an implementation because classes become tightly coupled.

Creational Design Patterns address this issue by decoupling the client entirely from the actual initialization process.

1. Singleton – Ensures that at most only one instance of an object exists throughout application
2. Factory Method – Creates objects of several related classes without specifying the exact object to be created
3. Abstract Factory – Creates families of related dependent objects

Singleton Design Pattern

The Singleton Design Pattern aims to keep a check on initialization of objects of a particular class by **ensuring that only one instance of the object exists throughout the Java Virtual Machine.**

A Singleton class also provides one unique global access point to the object so that each subsequent call to the access point returns only that particular object.

Example

Although the Singleton pattern was introduced by GoF, the original implementation is known to be problematic in multithreaded scenarios.

So here, we're going to follow a more optimal approach that makes use of a static inner class:

```
public class Singleton  {
	private static Singleton instance;
	private Singleton() {
	}

	public static Singleton getInstance() {
		if(instance==null) {
			instance=new Singleton();
		}
		return instance;
	}

	public void greeting() {
		System.out.println("Greeting to you!");
	}
}


public class Main {
	public static void main(String[] args) {
		Singleton instance=Singleton.getInstance();
		instance.greeting();

	}
}
```

Here, we've created a *static* inner class that holds the instance of the *Singleton* class. It creates the instance only when someone calls the *getInstance()* method and not when the outer class is loaded.

This is a widely used approach for a Singleton class as it doesn't require synchronization, is thread safe, enforces lazy initialization and has comparatively less boilerplate.

Also, note that the constructor has the *private* access modifier. **This is a requirement for creating a Singleton since a *public* constructor would mean anyone could access it and start creating new instances.**

Factory Method Design Pattern

The Factory Design Pattern or Factory Method Design Pattern is one of the most used design patterns in Java.

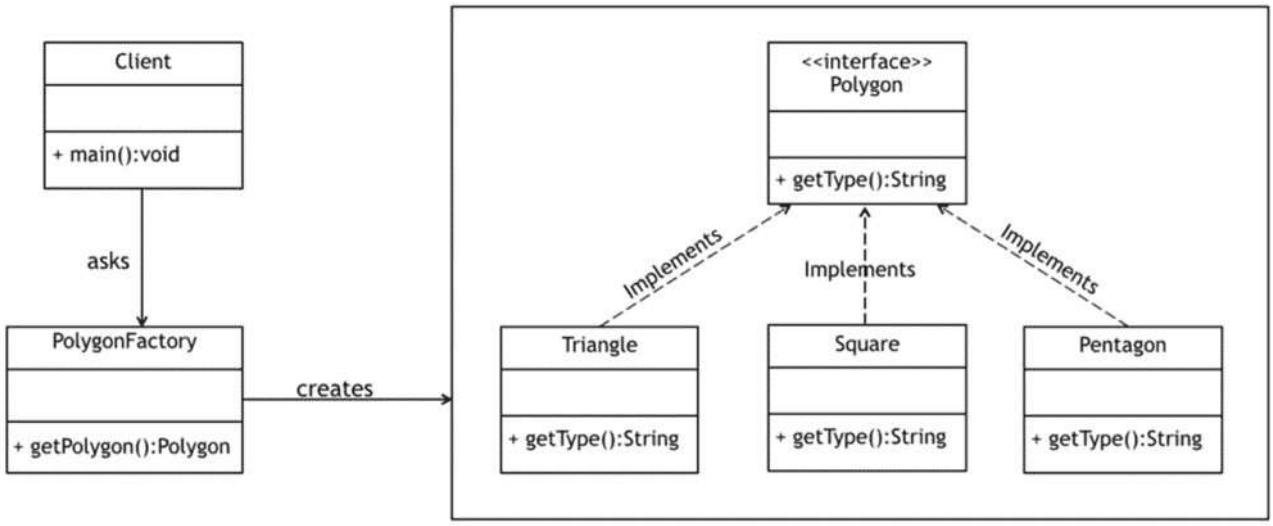
According to GoF, this pattern **"defines an interface for creating an object, but let subclasses decide which class to instantiate.** The Factory method lets a class defer instantiation to subclasses".

This pattern delegates the responsibility of initializing a class from the client to a particular factory class by creating a type of virtual constructor.

To achieve this, we rely on a factory which provides us with the objects, hiding the actual implementation details. The created objects are accessed using a common interface.

Example

In this example, we'll create a *Polygon* interface which will be implemented by several concrete classes. A ***PolygonFactory*** will be used to fetch objects from this family:



Let's first create the *Polygon* interface:

```
public interface Polygon {
    String getType();
}
```

Next, we'll create a few implementations like *Square*, *Triangle,* etc. that implement this interface and return an object of *Polygon* type.

```
public class Triangle implements Polygon{
     @Override
```

```
	public String getType() {
		return "Triangle";
	}
}

public class Square implements Polygon{
	@Override
	public String getType() {
		return "Square";
	}
}

public class Pentagon implements Polygon{
	@Override
	public String getType() {
		return "Pentagon";
	}
}

public class Heptagon implements Polygon{
	@Override
	public String getType() {
		return "Heptagon";
	}

}

public class Octagon implements Polygon{
	@Override
	public String getType() {
		return "Octagon";
	}
}
```

Now we can create a factory that takes the number of sides as an argument and returns the appropriate implementation of this interface:

```
public class PolygonFactory {
	public static Polygon getPolygon(int numSides) {
		if(numSides == 3) {
			return new Triangle();
		}
		if(numSides == 4) {
			return new Square();
```

```
            }
            if(numSides == 5) {
                  return new Pentagon();
            }
            if(numSides == 7) {
                  return new Heptagon();
            }
            else if(numSides == 8) {
                  return new Octagon();
            }
            return null;
      }

}


public class PolygonDemo {
      public static void main(String[] args) {
            Polygon  p = PolygonFactory.getPolygon(8);
            System.out.println(p.getType());
      }
}
```

Notice how the client can rely on this factory to give us an appropriate *Polygon*, without having to initialize the object directly.

Abstract Factory Design Pattern

In the previous section, we saw how the Factory Method design pattern could be used to create objects related to a single family.

By contrast, the Abstract Factory Design Pattern is used to create families of related or dependent objects. It's also sometimes called a factory of factories.

(Note: You can go google for the example)