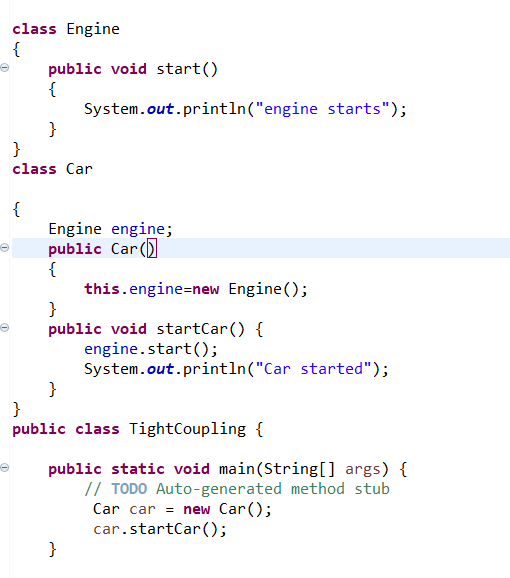
**Spring and Springboot :**

**TightCoupling :** Tight coupling occurs when classes or components are highly dependent on each other, requiring changes in one to necessitate changes in the other, thus making the system less flexible.

**LooseCoupling:** looselycoupling technique allows that there is no greater dependency or no dependency between two components or modules. like if we make changes to one component then there is a little or no impact on other other components.

**Example:**  now we have a class called gameRunner it is for running the game. So there are number of games are there like mario , spyro.

So here when we write the code in tight coupling.



So here there are three classes Engine , car and tightcouplling(main class).

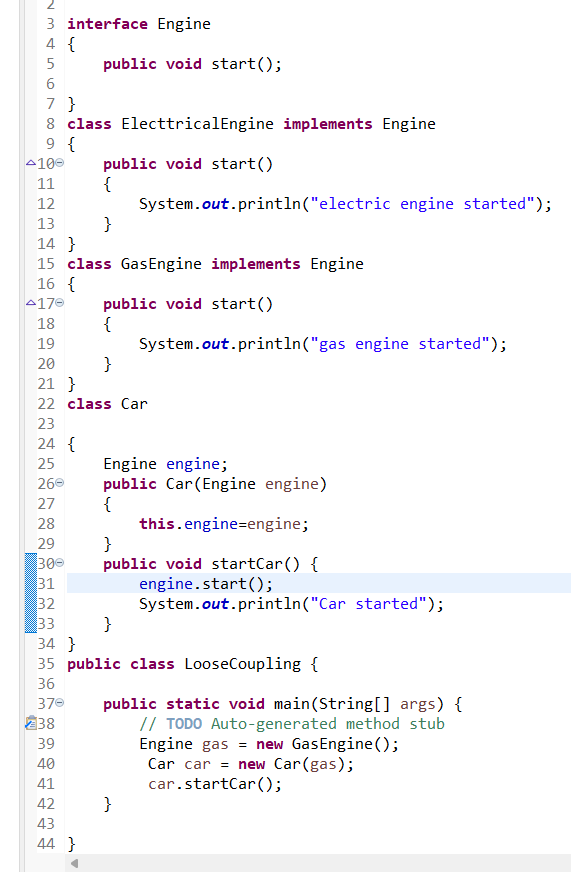
If you want to start a car you need an engine so the class car is dependent on engine. So it creates a engine object.

Then it calls the engine class method start then engine started and then car started.

It looks good but there is high dependency is there because car can start with gasengine and electric engine and may some other also.

So if we want to start a car with electric engine then you have to create that object inside the car class.so the car class is highly dependent on other engine classes.so by using loose coupling you can achieve this.

**Example for loose coupling:**



A screenshot of a computer

Description automatically generated

So here we can see that gasengine is started. So in this we just simply pass our object based on our requirement in main method so it just called the gas engine and started the car.

We write engine class(in tight coupling example) as interface so gas and electrica engines classes implement that interface. So when we create an object for gas engine it will execute or when we create an object for electrical engine it will executes.so it achieves the loose coupling.

**Dependency Injection:**

is a design pattern used in object-oriented programming that helps achieve loose coupling between classes.

Instead of creating dependencies inside a class, DI provides those dependencies from the outside, typically through:

* **Constructor Injection**: Dependencies are passed via the class constructor.
* A black screen with white text

  Description automatically generated
* **Setter Injection**: Dependencies are passed via setter methods.
* A computer screen with text on it

  Description automatically generated
* **Field Injection**: Dependencies are injected directly into the fields (not recommended for testing and flexibility purposes).

A black rectangular object with white text

Description automatically generated

**Example for Without DI :** refer Tight Coupling ex

**Example for with Di :** refer loose coupling ex.

**Dependency Injection (DI)** is a powerful design pattern that decouples components and helps achieve loose coupling.

**Spring IOC :**

IoC is a design principle where the control of object creation and dependency management is inverted, i.e., instead of the objects creating andmanaging their own dependencies, thisresponsibility is given to a framework or container. In Spring, the IOC container manages these tasks.

**Imp:\*** The **Spring IOC container** is responsible for creating, configuring, and managing the lifecycle of beans (Java objects) in the Spring application. It does this through **Dependency Injection** (DI), which is the mechanism by which Spring injects dependencies into objects.

**Autowired:**

@Autowired is an annotation in Spring Framework used for Dependency Injection (DI). It tells Spring to automatically inject a bean (object) into another bean, so you don’t have to manually instantiate or manage dependencies.

**DI vs spring IOC vs Autowired**

**DI :** di is a just a designing principle to inject the the dependent objects into another class and **IOC** allowed this DI technique in spring application and this ioc is responsible for creating and managing the objects using **@Autowired** annotation .

By using this **@Autowired** annotation we can inject the dependencies to another class.

🡺 constructor dependency injection is promoted or suggested by spring because constructor DI is more obvious i.e if you use the constructor DI you have to pass that dependencies while creating the object of the class. And it has immutability.

🡺 Constructor Di for mandatory dependencies and setter Di for optional dependencies

There are two types of Ioc containers:

1>**1. BeanFactory:**

* **Definition**: The BeanFactory is the basic IoC container in Spring. It is responsible for instantiating, configuring, and managing beans, but it provides only the **core** features of dependency injection.
* **Features**:
  + **Lazy Initialization**: Beans in BeanFactory are initialized **only when they are requested**. This means they are created on-demand, not at startup.

**2. ApplicationContext:**

* **Definition**: The ApplicationContext is a more advanced container than BeanFactory. It is an extension of BeanFactory and includes all of its features along with several additional enterprise-level features.
* **Features**:
  + **Eager Initialization**: By default, ApplicationContext initializes all singleton beans at startup, which means the application can detect errors earlier.

**Summary:**

* **BeanFactory**: The simplest container in Spring, providing basic DI (Dependency Injection). It’s a lightweight container for use in simple applications.
* **ApplicationContext**: A more powerful container that builds on BeanFactory and adds enterprise-level features like event handling, internationalization, AOP support, and more. Ideal for complex and large-scale Spring applications.

**Model vs Modelview :**

When you send a request front controller in the mvc handle the request send that request to the appropriated or related controller and then controller went to model to add the data or to retrieve data based on the request. So model sends the response to the controller. The data/response is sent to the controller and it returns a view name i.e string value.

And when it comes to modelandview , controller returns modelandview object directly.view is nothing but we are showing a result to the user.

And there is another concept **viewresolver** is for converting view name that is sent by the controller into a template so then it would be sent to the user as a response.

**ModelAttributes and SessionAttributes:**

Modelattributes are used to pass the data from controller to view i.e it stores response getting from the controller or methods return value and sent it the view for showing the response to the user. After sending the response to the user modelattributes has nothing to store. Its temporary location to store the data for every request. After the request has completed it doesn’t have any data in it.

**Session attributes** are used to store data or method return value that needs to persist(storing data) across multiple requests within the same session. They are typically used to hold data that is required by multiple requests, like user authentication status or user preferences.

**Conclusion**

* Use **Model Attributes** for data that is required only for a single request, like the details to populate a form or display a view.
* Use **Session Attributes** for data that needs to be maintained across multiple requests, like user authentication details or preferences.

**SpringBoot**

**@SpringBootApplication :** is used to add all the features or configuration that are needed to run a springboot application. This annotation is a combination of three annotations , i.e @Configuration , @componentScan , @EnableAutoConfiguration.

🡺 **@Configuration :** is used to define the beans. When we declared a class as @configuration then spring is responsible for creating beans and their dependencies to that class.

**🡺@ComponentScan :** is used for searching the classes which are annotated with @configuration ,@component, @repository etc to create beans for that class. Spring uses componentscan to scan the classes.

We just put the package inside the @componentscan annotation.so that whole package and its subpackage classes are to scanned, and their responsibility will taking care by spring ioc.

🡪**@EnableAutoConfiguration:**

Auto-configuration attempts to automatically configure your Spring application based on the dependencies you have added in your project. For instance:

* If you have spring-boot-starter-web in your classpath, Spring Boot will automatically configure components required for a web application (like an embedded Tomcat server, DispatcherServlet, etc.).
* If you have spring-boot-starter-data-jpa, it will configure a DataSource and a JPA EntityManager.

**Why embedded servers are need and what are embedded servers are there in springboot ?**

Before springboot , if you want to run a application on server you have to pack your all application code as a war file and then you have to deploy that war file on a external server. For that external server you have to install it manually and you have to configure it to run application.

But after springboot embedded server no need install and configure the external server. You just convert springboot application into a jar file(java archive) and that jar file contains application code and embedded server so that you can run on any machine.

Spring Boot supports several embedded servers, including **Tomcat** (default), **Jetty**, **Undertow**, and **Netty**, each suited for different performance and use case needs.

SpringBoot starter projects: In Spring Boot, **starter projects** are predefined Maven or Gradle dependencies that bundle common libraries and configurations, making it easier to set up and use specific features of Spring.

some of the most commonly used **Spring Boot Starter Projects**:

1. **spring-boot-starter-web**

if you want to develop a web application, you only need to include the spring-boot-starter-web starter, and it will automatically pull in dependencies like Spring MVC, Jackson (for JSON handling), and an embedded Tomcat server**.**

**2. spring-boot-starter-data-jpa**

* Purpose: For working with relational databases using Spring Data JPA and Hibernate.

Example Use Case: Building applications that interact with relational databases like MySQL, PostgreSQL, etc.

Key Dependencies: Spring Data JPA, Hibernate, Spring ORM.

**3. spring-boot-starter-security**

* Purpose: For adding Spring Security to your application for authentication and authorization.

**4. spring-boot-starter-test**

* Purpose: For testing Spring applications, including unit tests, integration tests, and mocks.
* Key Dependencies: JUnit 5, Mockito, Spring Test, AssertJ.

**5. spring-boot-starter-actuator**

* Purpose: For adding production-ready features to your application, such as monitoring, metrics, health checks, etc.
* Key Dependencies: Spring Boot Actuator.
* Example Use Case: Monitoring the health and status of a Spring Boot application.

**spring-boot-starter-parent** :

The **spring-boot-starter-parent** is a special parent project in Spring Boot, which provides common configurations for Spring Boot applications.

*  For example, when you include spring-boot-starter-web in your project, you don’t need to specify the versions for Spring MVC, Tomcat, or Jackson. The parent takes care of versioning for you.

 **JPA (Java Persistence API) vs Spring Data JPA :**

**JPA:**

The main aim of jpa is to mapping a java object into a row in a table. It just provides guidelines and rules for mapping

And the implementation can done by ORM tools like Hibernate , structs etc.

We have to Create a persistence.xml file to configure the database connection and JPA settings.

We Use the EntityManager to perform CRUD operations.

**Spring Data JPA:**

**Spring Data JPA** is built on top of JPA, so it uses the same concepts and annotations. it automates many common tasks like creating repositories, managing queries, and dealing with relationships between entities.

Using spring data jpa there is no need configure the database connections in persistence.xml file. Spring is configure all the database connections.

We just provide database details like database name, username,password inside application.properties.

There is no need to use entitymanager to perform crudoperations. We just create interface and extends the crudrepository , jparepository or pagingsortingrepository which provides all the inbuilt methods like create , delete, update, findall,findbyId

**Controller vs RestController :**

Controller and restcontroller both are used to handle the web requests but controller returns the views or templates and the we have to use@responseBody annotation on a method to convert view into json object.

But the @Restcontroller is the combination of @controller and @ResponseBody

**1. @PathVariable**

**Purpose**:

* Binds a method parameter to a variable within the URI path.

A screen shot of a computer code

Description automatically generated

For any finding operations we use this.

**2. @RequestParam**

**Purpose**:

* Binds a method parameter to a query parameter in the URL or form data.

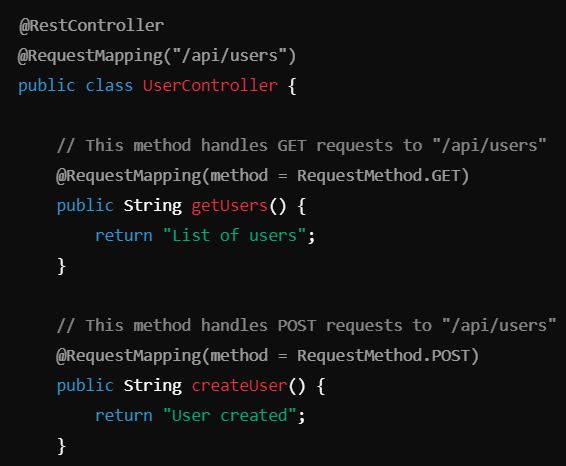
**Usage Scenario**:

* When the data is not part of the URI path but is instead passed as a query parameter.
* Suitable for filtering, searching, pagination, and optional parameters



**@RequestMapping**: it maps the http requests to the appropriate methods inside the @controller class. It is a class level annotation or methodlevel annotation.

Key features of requestMapping is @getMapping, @putMapping, @postmapping, @DeleteMapping



**@ControllerAdvice:** Is a powerful annotation in a spring that it is used to handle the exceptions globally . and this common behavior is applied to all the controllers in application.

It allows you to define methods to handle specific exceptions or all exceptions thrown by controllers**.**