Dependency Injection Pattern:

- The **dependency injection** is an **Enterprise Design Pattern**, which aims to separate the responsibility of resolving object dependency from its behaviour.
- **Note:** An Enterprise Design Pattern is a Design Pattern used in enterprise applications.
- The dependency injection design pattern allows us to remove the hard-coded dependencies and make our application loosely coupled, extendable and maintainable.
- If you want to use this technique, you need classes that fulfill 3 basic roles. These are:
  1. The service you want to use.
  2. The client that uses the service.
  3. The injector which creates a service instance and injects it into the client.
- E.g.
  Let's say we have an application where we consume EmailService to send emails.
  We implement EmailService below:

```
public class EmailService {

	public void sendEmail(String message, String receiver){
		//logic to send email
		System.out.println("Email sent to "+receiver+ " with
					Message="+message);
	}
}
```

  EmailService class holds the logic to send an email message to the recipient email address. Our application code will be like below.

```
public class MyApplication {

	private EmailService email = new EmailService();

	public void processMessages(String msg, String rec){
		//do some msg validation, manipulation logic etc
		this.email.sendEmail(msg, rec);
	}
}
```

  Our client code that will use MyApplication class to send email messages will be like below.

```
public class MyLegacyTest {

	public static void main(String[] args) {
		MyApplication app = new MyApplication();
		app.processMessages("Hi Pankaj", "pankaj@abc.com");
	}
}
```

  There are a few problems with this design:
  1. The MyApplication class is responsible to initialize the email service and then use it. This leads to hard-coded dependency. If we want to switch to some other advanced email service in the future, it will require code changes in MyApplication class. This makes our application hard to extend and if email service is used in multiple classes then that would be even harder.

In general, letting a class to explicitly create an instance of another class tightly coupled the two implementations, increasing the dependency between them.

2. If we want to extend our application to provide an additional messaging feature, such as SMS or Facebook message then we would need to write another application for that. This will involve code changes in application classes and in client classes too.

3. Testing the application will be very difficult since our application is directly creating the email service instance. There is no way we can mock these objects in our test classes.

One possible solution is to use the factory design pattern to create the objects.
The intent is to create objects without exposing the instantiation logic to the client and to refer to the newly created object through a common interface.
However, this is still not good because we can get a chain of growing dependencies.
Then, there are two possibilities:
1. The chain blows up in finite time.
2. The chain stabilizes (i.e. the nightmare comes to a limit).

Instead, we take the factory idea to the limit and we create a module that is responsible to resolve dependency among classes.
The injector module is called dependency injector and it is, to all effects, a container.
Applying the Inversion of Control (Dependency inversion) pattern, the injector owns the life cycle of all objects defined under its scope.
The only things we still miss are the following:
  - A way to signal to the injector that a class has a certain dependency.
  - A way to instruct or configure the injector to resolve dependencies.
- Since the definition of the Java programming language, the aim of the Sun Microsystem was standardization.
- The Java Bean Specification stated that a java bean must have:
  - A default constructor.
  - A couple of getter and setter methods for each attribute it owns.
- Having defined a standard way of creating objects, the Sun built a plethora of standards and interfaces on it, that are the basis of the JEE specification.
- Since dependencies are nothing more then objects attributes (more or less), the injector has to be instructed to know how to fulfill these attributes.
  In Java there are three ways to set attributes of an object, which are:
  1. Using the proper constructor.
  2. Using setters after the object was built using the default constructor.
  3. Using Reflection mechanisms
- Hence, there are 3 ways of doing the dependency injection principle:
  1. Using an interface. Here, the dependency provides an injector method that will inject the dependency into any client passed to it. Clients must implement an interface that exposes a setter method that accepts the dependency.
  2. Using a constructor. Here, the dependencies are provided through a class constructor.
  3. Using setter methods. Here, the client exposes a setter method that the injector uses to inject the dependency.

  Once you have selected your preferred way, you will annotate the corresponding statement with the @Inject annotation.

The annotation and its behaviour are defined inside a JSR. Dependency injection is so important in the Java ecosystem that there are two dedicated Java Specification Requests (JSRs), specifically JSR-330 and JSR-299.

- If you want the injector to use the constructor to inject the dependency of a class, annotate the constructor with @Inject.
  E.g.

```
public class MovieLister {
    private MovieFinder finder;

    @Inject
    public MovieLister(MovieFinder finder) {
        this.finder = finder;
    }
}
```

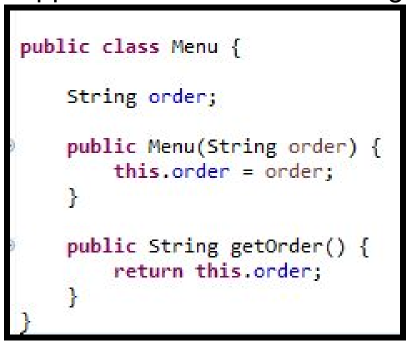
  Every time you will ask the injector an instance of a MovieLister, it will know that it has to use the constructor annotated with the @Inject annotation to build the object.
  So, the dependencies are identified as the annotated constructor parameters.
- To instruct the injector to use setter methods to create an object, annotate the setter methods with the @Inject annotation.
  E.g.

```
public class MovieLister {
    private MovieFinder finder;

    // The injector first uses the default
    // constructor to build an empty object
    public MovieLister() {}
    // Then, the injector uses annotated setter
    // methods to resolve dependencies
    @Inject
    public void setFinder(MovieFinder finder) {
        this.finder = finder;
    }
}
```

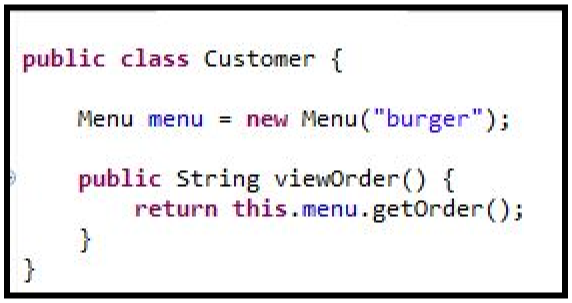
- There are many implementations in the Java ecosystem of dependency injectors. Each implementation differs from each others essentially for these features:
    - How the injector is configured to find the beans it has to manage.
    - How it resolves the dependencies DAG.
    - How it maintains the instances of managed beans.
- Some injectors used in practice are the following:

- **Google Guice:** Guice is a lightweight dependency injection framework that uses Java configuration, implements both types of injection (constructor and setter injection) and maintains managed instances with a Map<Class<?>, Object>.
- **Spring Injector:** It is not quite lightweight as Guice, but it is fully integrated with the whole Spring ecosystem. It implements all types of injection; it provides an XML style of configuration and a Java style. Differently from Guice, the
- container is a Map<String, Object>, that associates each managed instance with a label. It also supports auto discovery of beans through classpath scanning.

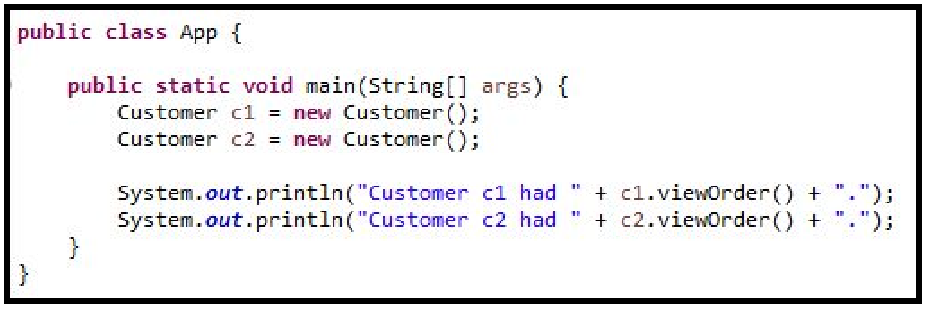
- Example of the dependency injection principle:
  Suppose we have the following classes - Customer and Menu



```java
public class Menu {

    String order;

    public Menu(String order) {
        this.order = order;
    }

    public String getOrder() {
        return this.order;
    }
}
```

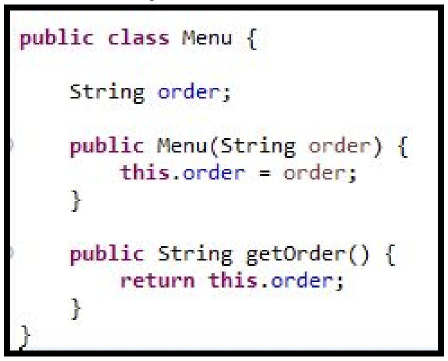


```java
public class Customer {

    Menu menu = new Menu("burger");

    public String viewOrder() {
        return this.menu.getOrder();
    }
}
```

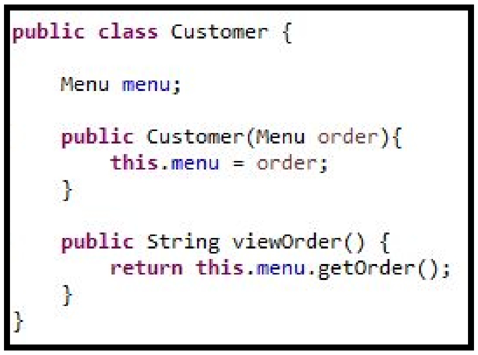


```java
public class App {

    public static void main(String[] args) {
        Customer c1 = new Customer();
        Customer c2 = new Customer();

        System.out.println("Customer c1 had " + c1.viewOrder() + ".");
        System.out.println("Customer c2 had " + c2.viewOrder() + ".");
    }
}
```

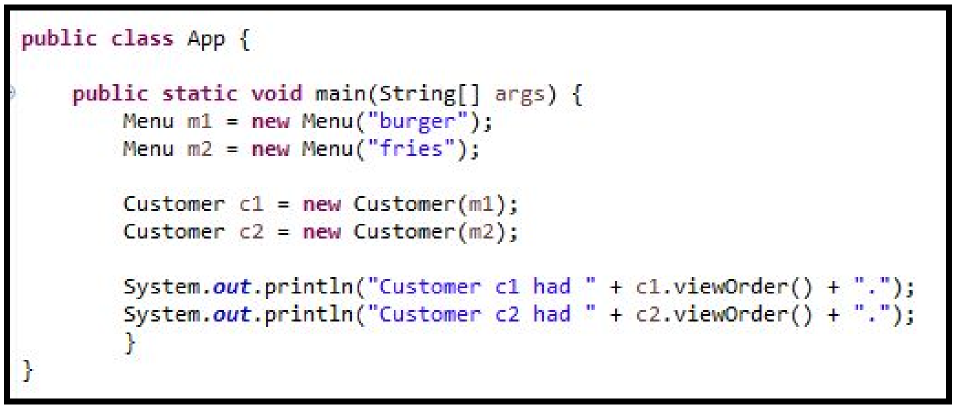
A problem arises when a customer doesn't want to only order burgers.
Furthermore, it's not good practice to create an instance of another class inside a class.
A better way to do it is like this:



```java
public class Menu {

    String order;

    public Menu(String order) {
        this.order = order;
    }

    public String getOrder() {
        return this.order;
    }
}
```



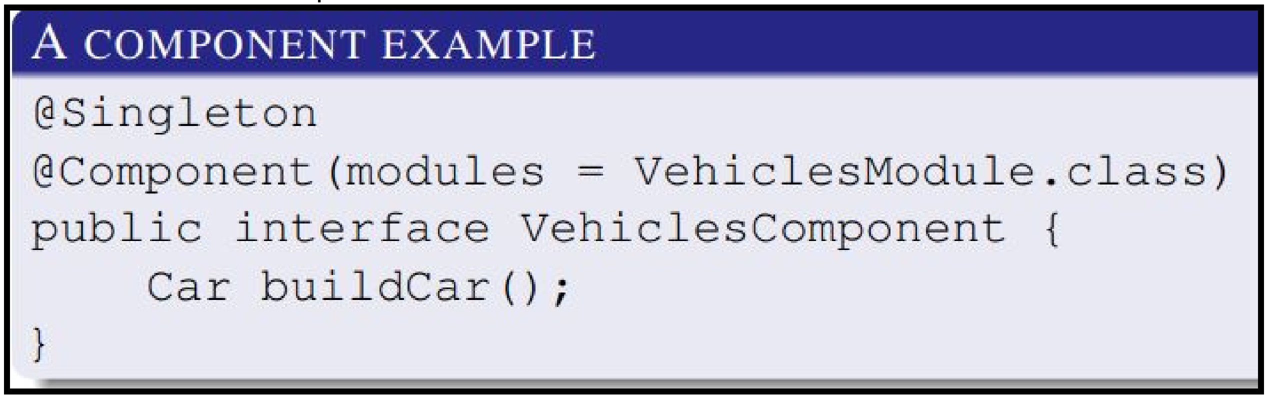
```java
public class Customer {

    Menu menu;

    public Customer(Menu order){
        this.menu = order;
    }

    public String viewOrder() {
        return this.menu.getOrder();
    }
}
```



```
public class App {

    public static void main(String[] args) {
        Menu m1 = new Menu("burger");
        Menu m2 = new Menu("fries");

        Customer c1 = new Customer(m1);
        Customer c2 = new Customer(m2);

        System.out.println("Customer c1 had " + c1.viewOrder() + ".");
        System.out.println("Customer c2 had " + c2.viewOrder() + ".");
        }
}
```

<u>**Dagger:**</u>

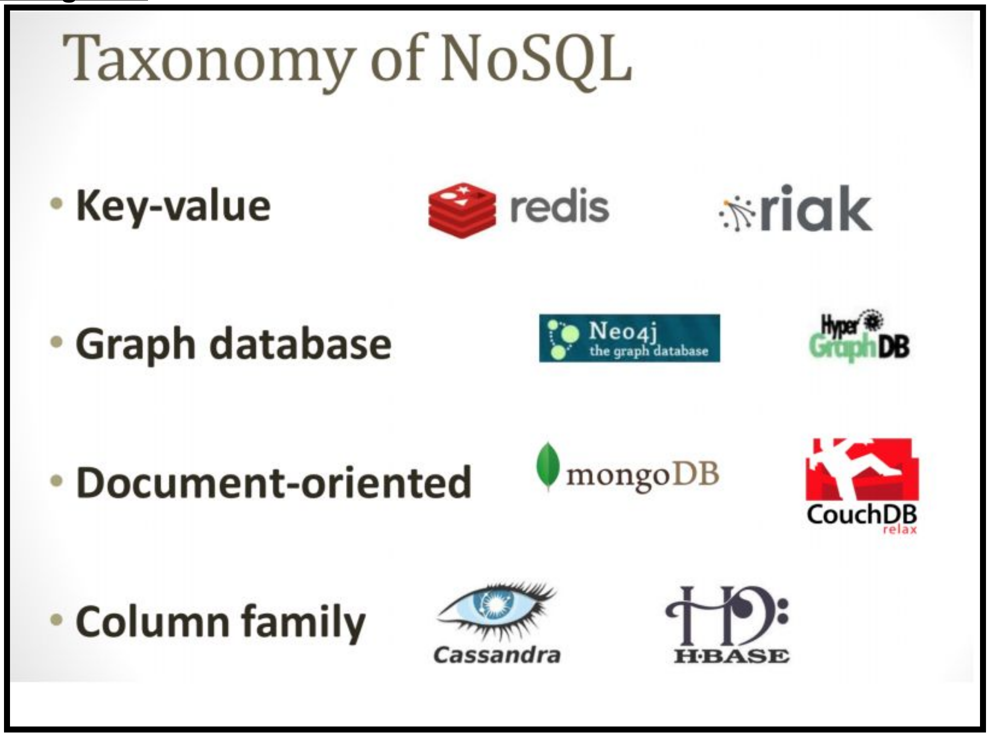
- Dagger-2 is a fast and lightweight dependency injection framework.
- It is implemented through an external component which provides instances of objects (or dependencies) needed by other objects.
- In particular, the injection happens at run-time or at compile-time.
- Run-time DI is usually based on reflection which is simpler to use but slower at run-time. An example of a run-time DI framework is Spring.
- Compile-time DI, on the other hand, is based on code generation.
  This means that all the heavy-weight operations are performed during compilation.
  Compile-time DI adds complexity but generally performs faster.
  Dagger 2 falls into this category.
- **Components** are essentially the glue that holds everything together.
- They are a way of telling Dagger what dependencies should be bundled together and made available to a given instance so they can be used.
- They provide a way for a class to request dependencies being injected through their @Inject annotation.
- E.g.
  Moving on, we're going to create our component interface. This is the class that will generate Car instances, injecting dependencies provided by VehiclesModule.
  Simply put, we need a method signature that returns a Car and we need to mark the class with the @Component annotation. Notice how we pass our module class as an argument to the @Component annotation. If we didn't do that, Dagger wouldn't know how to build the car's dependencies.



A COMPONENT EXAMPLE

```
@Singleton
@Component(modules = VehiclesModule.class)
public interface VehiclesComponent {
    Car buildCar();
}
```

<u>MongoDB:</u>



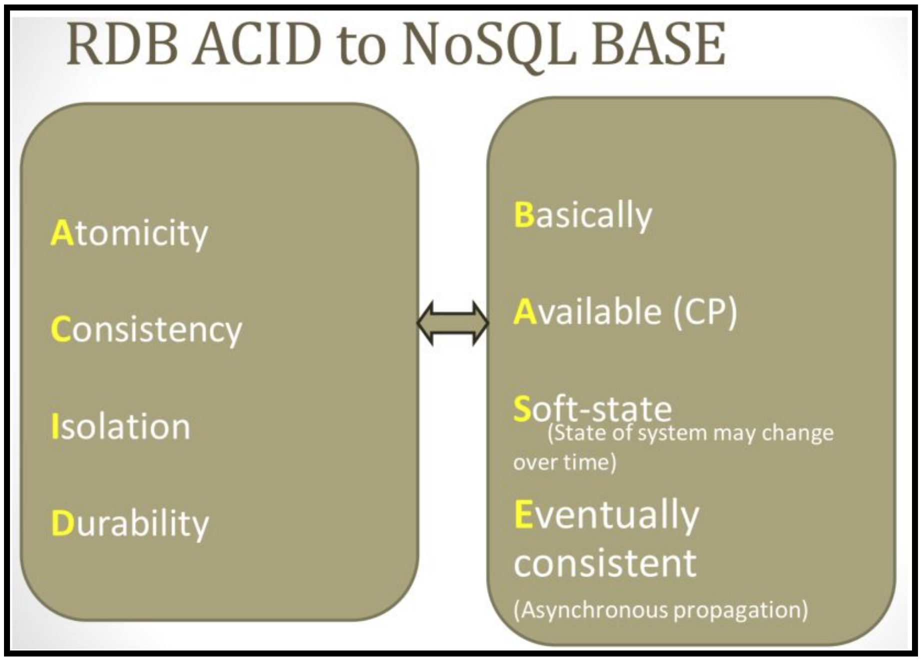
Taxonomy of NoSQL

- Key-value
- Graph database
- Document-oriented
- Column family

redis
riak
Neo4j the graph database
HyperGraphDB
mongoDB
CouchDB relax
Cassandra
HBASE

- **Theory of NoSQL - CAP:**
- Given many nodes where nodes contain replicas of partitions of the data, we want:
    - **C**onsistency:
        - All replicas contain the same version of the data.
        - The client always has the same view of the data no matter the node.
    - **A**vailability:
        - System remains operational on failing nodes.
        - All clients can always read and write.
    - **P**artition Tolerance:
        - Multiple entry points.
        - System remains operational on system split (communication malfunction).
        - System works well across physical network partitions.
- **CAP Theorem:** Satisfying all 3 at the same time is impossible.
- **How Does NoSQL Vary From RDBMS:**
- Looser schema definition.
- Applications are written to deal with specific documents/data.
  Applications are aware of the schema definition as opposed to the data.
- Designed to handle distributed, large databases.
- Trade offs:
    - No strong support for ad hoc queries but designed for speed and growth of databases.
      An **ad hoc query** does not reside in the computer or the database manager but is dynamically created depending on the needs of the data user. In SQL, an ad hoc query is a loosely typed command/query whose value depends upon some variable.

- Relaxation of ACID properties.
  I.e.



- **Benefits of NoSQL:**
  1. **Elastic scaling:**
     - RDBMS scales up while NoSQL scales out.
       I.e. RDBMS uses a bigger server while NoSQL uses multiple servers.
     - NoSQL distributes data across multiple hosts.
  2. **Big Data:**
     - NoSQL is designed for large amounts of data.
     - If there is a huge increase in data, DBMS's capacity and constraints of data volume are at its limits.
  3. **DBA Specialists:**
     - RDBMS requires highly trained specialists to monitor databases.
     - NoSQL requires less management, automatic repair and simpler data models.
  4. **Flexible Data Models:**
     - Changing a schema in RDBMS has to be carefully managed.
     - NoSQL databases are more relaxed in the structure of data.
     - Database schema changes do not have to be managed as one complicated change unit.
     - The application is already written to address an amorphous schema.
  5. **Economics:**
     - RDBMS relies on expensive proprietary servers to manage data.
     - NoSQL relies on clusters of cheap commodity servers to manage the data and transaction volumes.
     - Cost per gigabyte or transaction per second for NoSQL can be lower than the cost for RDBMS.

- **Disadvantages of NoSQL:**
    1. **Support:**
    - RDBMS vendors provide a high level of support to clients and they have a stellar reputation.
    - NoSQL are open source projects with startups supporting them. Their reputation is not yet established.
    2. **Maturity:**
    - RDBMS are mature products, meaning that they are stable and dependable. However, this also means that they are old and no longer cutting edge or interesting.
    - NoSQL are still implementing their basic feature set.
    3. **Administration:**
    - The role of a RDBMS administrator is well defined.
    - For NoSQL, no administrator is necessary but NoSQL still requires effort to maintain.
    4. **Lack of Expertise:**
    - The whole workforce for RDBMS developers is trained and seasoned.
    - They are still recruiting developers for NoSQL.
    5. **Analytics and Business Intelligence:**
    - RDBMS is designed to address this niche.
    - NoSQL is designed to meet the needs of an Web 2.0 application, not designed for ad hoc queries. However, tools are being developed to address this need.
- **MongoDB:**
- Developed by 10gen in 2007.
- It is a document-oriented NoSQL database.
- It is a hash-based (dynamic) and schema-less database.
  That means there's no data definition language (DDL).
  In practice, this means that you can store hashes with any keys and values that you choose.
  Keys are a basic data type but in reality they are stored as strings.
  Document identifiers (_id) will be created for each document. The field name is reserved by the system.
  MongoDB does not need any pre-defined data schema.
  Every document in a collection could have different data. This addresses NULL data fields.
- Tracks the schema and mapping.
- Uses BSON format.
  BSON is based on JSON, but the B stands for Binary.
  It is a binary-encoded serialization of JSON-like documents.
  Zero or more key/value pairs are stored as a single entity.
  Each entry consists of a field name, data type and value.
  Large elements in a BSON document are prefixed with a length field to facilitate scanning.
- Written in C++.
- Supports APIs (drivers) in many languages, such as:
    - JavaScript
    - Python
    - Ruby
    - Perl
    - Java
    - C++

- It uses **secondary indexes**. A secondary index is a way to efficiently access records in the primary database by means of some piece of information other than the usual primary key.
- Query language via API
- Atomic writes and fully consistent reads, if the system is configured that way.
- Master-slave replication with automated failover (replica sets).
- Built in horizontal scaling via automated range-based partitioning of data.
- No joins or transactions.
- Hierarchical Objects:
    - A MongoDB instance may have zero or more databases.
    - A database may have zero or more collections.
    - A collection may have zero or more documents.
    - A document may have zero or more fields.

      Note: MongoDB indexes function much like their RDBMS counterparts.
- RDBMS Concepts to NoSQL:

RDBMS	**MongoDB**
Database	Database
Table, View	Collection
Row	Document (BSON)
Column	Field
Index	Index
Join	Embedded document
Foreign key	Reference
Partition	Shard

- JSON Format:
    - Data is in name/value pairs.
    - A name/value pair consists of a field name followed by a colon followed by a value.

      E.g. “name”: “ABC”
    - Data is separated by commas.

      E.g. “name”: “ABC”, “age”: 15
    - Curly brackets, { }, hold objects.

      E.g. {“name”: “ABC”, “age”: 15}
    - An array is stored in square brackets, [ ].

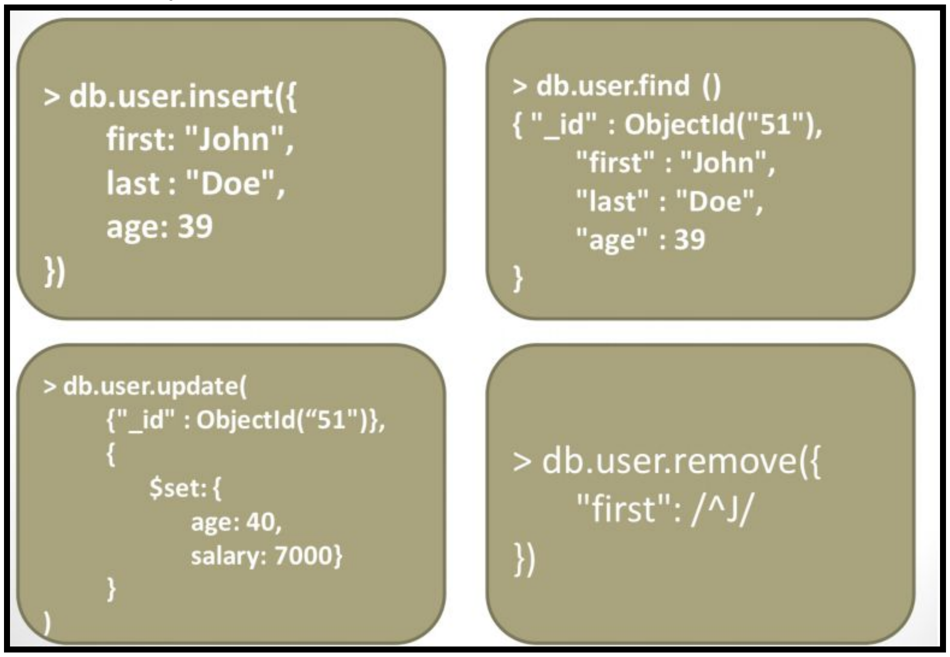
      E.g. [{“name”: “ABC”, “age”: 15}, {“name”: “DEF”, “age”: 45}]

- CRUD Operations:
    - CRUD stands for **C**reate **R**ead **U**pdate **D**elete
    - Create:
        - Syntax: **db.collection.insert(<document>)**
          E.g.
          db.parts.insert({type: “screwdriver”, quantity: 15})
          Note: We can omit the _id field, as we did above, to have MongoDB generate a unique key. We can also fill in the _id field ourselves, as shown below.
          db.parts.insert({_id: 10, type: “screwdriver”, quantity: 15})
        - Syntax: **db.collection.save(<document>)**
          Will update an existing record or create a new record.
        - Syntax: **db.collection.update(<query>, <update>, {upsert:true})**
          Will update 1 or more records in a collection that satisfies the query.
    - Read:
        - Syntax: **db.collection.find(<query>, <projection>)**
        - Syntax: **db.collection.findOne(<query>, <projection>)**
        - Similar to the SELECT command in SQL.
        - Can modify the query to impose limits, skips and sort orders.
          E.g.
          db.parts.find({parts: “hammer”}).limit(5)
        - Can specify to return the top number of records from the result set.
    - Update:
        - Syntax: **db.collection.update(<query>, <update>, <options>)**
    - Delete:
        - Syntax: **db.collection.remove()**
        - Deletes all records in the given collection.
        - Syntax: **db.collection.remove(<query>, <justOne>)**
        - Deletes all records that satisfies the query.
        - The justOne specifies to delete only 1 record that satisfies the query.
        - E.g.
          db.parts.remove({type: /^h/}) removes all parts starting with h.
    - Query Operators:

Name	**Description**
$eq	Matches values that are equal to a specified value
$gt, $gte	Matches values that are greater than/greater than or equal to a specified value
$lt, $lte	Matches values that are less than/less than or equal to a specified value
$ne	Matches values that are not equal to a specified value
$in	Matches any of the values specified in an array
$nin	Matches none of the values specified in an array
$or	Join query clauses with a logic OR
$and	Join query clauses with a logic AND

$not	Inverts the effect of a query expression
$nor	Join query clauses with a logic NOR
$exists	Matches documents that have a specified field

- CRUD Examples



```
> db.user.insert({
      first: "John",
      last : "Doe",
      age: 39
})
```

```
> db.user.find ()
{ "_id" : ObjectId("51"),
      "first" : "John",
      "last" : "Doe",
      "age" : 39
}
```

```
> db.user.update(
      {"_id" : ObjectId("51")},
      {
          $set: {
              age: 40,
              salary: 7000}
      }
)
```

```
> db.user.remove({
      "first": /^J/
})
```