You can find lots of articles and videos explaining how to write maintainable Java code. All of that advice is also valid for your persistence layer. In the end, it's still Java code, isn't it?

But that advice isn't enough.

You can follow general Java best practices and still build a messy persistence layer. JPA and Hibernate heavily rely on *String* references, which quickly get unmaintainable, and the managed lifecycle introduces invisible functionalities and requirements.

So, you need to take it one step further and not only follow best practices for Java. You also need to apply specific best practices for a persistence layer using JPA or Hibernate. But don't worry. That's much easier than you might think.

Use constants for query and parameter names

Let's start with one of the most annoying parts of JPA and Hibernate: Magic *Strings* that reference entity attributes, queries, bind parameters, entity graphs, …

The best and only thing you can do to avoid these problems is to introduce *String* constants for at least each element you reference in your business code. If you want to take it one step further, you could even introduce constants for all entity attributes.

```
@Entity
@Table(name = "purchaseOrder")
@NamedQuery(name = Order.QUERY_ALL, query = "SELECT o FROM Order o")
public class Order {

    public static final String QUERY_ALL = "query.Order.all";

    public static final String ATTRIBUTE_ID = "id";
    public static final String ATTRIBUTE_ORDER_NUMBER = "orderNumber";

    ...
}
```

You can then use these constants in your code instead of magic *Strings*.

```
List<Order> orders = em.createNamedQuery(Order.QUERY_ALL).getResultList();
```

By doing that, you get a few things back that you lost in the first place:

- You can use your IDE to find all places in your code that call a specific query or use any other named element.
- It gets much easier to find and reuse existing queries that already fetch the required information.
- If you need to rename an attribute or any other named element, you simply change the value of the *String* constant.

Use the JPA Metamodel with your JPA APIs

If you're working with some of JPA's APIs, like the Criteria API or the Entity Graph API, you should prefer JPA's Metamodel classes over *String* constants. The metamodel consists of static classes that your persistence provider, e.g., Hibernate, can generate at build time. You can then use them in your code to reference entity attributes in a type-safe way.

Hibernate generates the metamodel classes, if you add a dependency to the *hibernate-jpamodelgen* artifact to your maven *pom.xml*.

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-jpamodelgen</artifactId>
</dependency>
```

You can then use the *Order_* class with most of JPA's APIs. I use it in the following example to select the orderNumber and customer attributes of the *Order* entity.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Tuple> cq = cb.createTupleQuery();
Root<Order> root = cq.from(Order.class);
cq.multiselect(root.get(Order_.ORDER_NUMBER), root.get(Order_.CUSTOMER));
```

Use field-based access

Another way to improve the readability and usability of your entities is to use field-based access.

It's one of the 2 access strategies supported by JPA and Hibernate. You use it by annotating your entity attributes with the mapping annotations. That puts all mapping information at the top of the class, and you can get a quick overview of all mapped attributes.

```
@Entity
@Table(name = "purchaseOrder")
public class Order {

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

    private String orderNumber;

    @OneToMany(mappedBy = "order", fetch = FetchType.LAZY)
    private Set<OrderItem> items = new HashSet<OrderItem>();

    @ManyToOne(fetch = FetchType.LAZY)
    private Customer customer;

    ...
}
```

In addition to that, it gives you the freedom to implement the getter and setter methods in any way you want. By using field-based access, you tell Hibernate to use reflection to set or get the value of an attribute. That enables you to implement the getter and setter methods in a way that makes your entity easy to use.

You could, for example, introduce a utility method that manages a bidirectional association. It’s easy to forget to update both ends of an association, so why not offer a method for it? It’s a general best practice for to-many associations.

```
@Entity
@Table(name = "purchaseOrder")
public class Order {
    public Set<OrderItem> getItems() {
        return this.items;
    }

    public void addItem(OrderItem item) {
        item.setOrder(this);
        this.items.add(item);
    }

    public void removeItem(OrderItem item) {
        item.setOrder(null);
        this.items.remove(item);
    }
}
```

Use named bind parameters

Named bind parameters are an easy and effective way to improve the readability of the code that executes a query. That’s especially the case if you combine it with my first recommendation and create a *String* constant for it.

```
@Entity
@Table(name = "purchaseOrder")
@NamedQuery(name = Order.QUERY_BY_CUSTOMER,
    query = "SELECT o FROM Order o WHERE o.customer = :"
                   + Order.PARAM_CUSTOMER)
public class Order {


    public static final String QUERY_BY_CUSTOMER = "query.Order.byCustomer";


    public static final String PARAM_CUSTOMER = "customer";


    ...
}
```

By using a named bind parameter in your query, you make the code that calls the query easier to understand. Everyone can immediately see which value it sets for which bind parameter.

```
TypedQuery<Order> q = em.createNamedQuery(Order.QUERY_BY_CUSTOMER,
Order.class);
q.setParameter(Order.PARAM_CUSTOMER, "Thorben Janssen");
List<Order> orders = q.getResultList();
```

Use Hibernate's *QueryHints* and *GraphSemantic* class to define a query hint

You can use query hints to provide additional information about a query and to activate or deactivate certain features of the *EntityManager*. You can use it to mark a query as read-only, activate Hibernate's query cache, set an SQL comment, reference an entity graph that shall be applied to a query and much more. I summarized the most interesting ones in 11 JPA and Hibernate query hints every developer should know.

The easiest and most readable way to set a query hint is to use Hibernate's *org.hibernate.annotations.QueryHints* and *org.hibernate.graph.GraphSemantic* classes with String constants for most query hints. Using that class, you can rewrite the previous example to use the *GraphSemantic.FETCH* constant instead of *javax.persistence.fetchgraph*.

```
Order newOrder = em.find(Order.class, 1L,
        Collections.singletonMap(GraphSemantic.FETCH.getJpaHintName(), graph));
```

Get query results as Tuples

The last recommendation that I want to give in this article is to use the *Tuple* interface to handle query results that contain multiple objects. You can see an example of such a query in the following code snippet.

```
List<Tuple> results = em.createQuery("SELECT " + Order.ATTRIBUTE_ID +
        " as " + Order.ATTRIBUTE_ID + ", " + Order.ATTRIBUTE_ORDER_NUMBER
        + " as "+Order.ATTRIBUTE_ORDER_NUMBER+" FROM Order o",
        Tuple.class).getResultList();

for (Tuple r : results) {
    log.info("ID: "+r.get(Order.ATTRIBUTE_ID));
    log.info("Order Number: "+r.get(Order.ATTRIBUTE_ORDER_NUMBER));
}
```

That query selects the *id* and the *orderNumber* of an *Order* entity as scalar values. If you don't want to use a DTO projection, you can either process each record as an *Object[]* or as a *Tuple* interface.

The *Tuple* interface is the better option. It provides methods to access each element of a record in the resultset by index or by alias. You can even automatically cast each of them to the correct type.