@Entity

You can think of an entity as a object whose state need to be persisted to some kind of persistence mechanism. In the context of the Java persistence API, relational databases are used to persist the state of your objects. Usually, an entity class exposes its persistent state through JavaBeans-style properties, also known as getters and setters methods. But you are free to expose its persistent state through fields if you want. Later on, we will see how we define the persistent state of an entity class.

@Entity Minimum requirement

In order to perform its job, you need to supply some details about the class whose instances you want to persist its state.

In the same way you, before buying a new software, need to know its minimum requirements to fulfil its needs, you need to know the minimum requirements to define an entity class as well. To mark a class as an entity class, you need to annotate it with the `@Entity` annotation on class declaration. To begin our journey, let's define a single version of the `Season` entity class

```
import javax.persistence.*;

@Entity
public class Season implements Serializable {}
```

For now, we omitted the class body. The `@Entity` annotation, placed on class declaration, mark it as an entity class. Annotate a class with the `@Entity` means its state should be persisted to the database. The interface `Serializable`, although not required, may be necessary if some instance of your class needs to be serialized. The `@Entity` annotation has an attribute called `name` which is defaulted to the unqualified name of the entity class, in our case, `Season`. Thus, when we refer to the entity name, we are referencing the `name` attribute of the `@Entity` annotation. Any entity class will, by default, be mapped to a table with the same name as the entity name. But if the default mapped table name does not meet your needs, you can annotate the class with the `@Table` annotation and specify its desired value by using its `name` attribute as follows

```
import javax.persistence.*;

@Entity
@Table(name="SEASONS")
public class Season implements Serializable {}
```

Now, instead of being mapped to a table called `Season`, the entity name, we override its default behavior by using the `@Table` annotation whose `name` attribute has been set to `SEASONS` – of course, you can use another name as well. Another useful scenario where the `@Table` annotation comes in handy is when you have a legacy system whose database uses a case sensitive approach in order to recognize its tables.

Java Persistence and data object names

`SQL-92` standard states that data object names such as tables and columns are case-insensitive unless delimited by double quotes. Because of this, some caution must be taken into account when mapping your entity class. Notice, for instance, the following class

```
@Entity
public class Group implements Serializable {}
```

Because a class named `Group` will, by default, be mapped to a table named `Group`, which is a SQL reserved keyword, you need to override its default table name by using delimited identifiers – surrounded by double quotes - as follows

```
@Entity
@Table("\"GROUP\"")
public class Group implements Serializable {}
```

Although `SQL-92` is to be a standard, implementations slightly vary from vendor to vendor. Hence, whatever the vendor, be consistent when defining data object names - once you can deal with portability issues between vendors - by adopting good practices as follows

Use either all uppercase or all lowercase when naming data objects such as tables and columns once some databases settings are case-sensitive. Anyway, follow some convention and stick to them.

Avoid using SQL reserved keywords as data object names, even if delimited by double quotes, because it tends to make your code unreadable. If needed, prefer to use other more descriptive name instead. The same rule apply to special characters.

If possible, prefix column names with the table name so that you do not need to worry about ambiguous column names when querying for such objects involving complex joins, making your code more readable.

The persistence state of your entity

To do its job, Java persistence access the state of your entity either using its fields or properties; that is, it reads from and writes to the database either accessing its fields or properties. Whether to access either fields or properties, it will depend on the placement of mapping-related annotations. Basically, mapping-related annotations refer to the annotations which are responsible for specifying database identifiers or primary key, mapping attributes to columns and so on. As we read this chapter, we will see basic mapping-related annotations.

Field access strategy

If you place all of your mapping-related annotations on the fields, Java persistence will access the state of your entity by using its fields. This kind of access is called field access strategy. If you use field access strategy, your fields must be declared as `private`, `protected` or defaulted to the package visibility so that if you want to provide client access to the state of your entity, you need to add some public method, usually getters and setters, to your entity class as a way of manipulating your data.

Property access strategy

As fields, if you place all of your mapping-related annotations on the properties, Java persistence will access the state of your entity by using its properties. This kind of access is called property access strategy. Property access strategy requires your properties must follow a JavaBean-style property. A JavaBean-style property is a pair of accessor methods, also known as getters and setters, whose method signature follows the pattern

```
<PROPERTY_TYPE> get<PROPERTY_NAME>()
void set<PROPERTY_NAME>(<PROPERTY_TYPE> <ref>)
```

where `<PROPERTY_TYPE>` should by replaced by the type of the property, `<PROPERTY_NAME>` by replacing its first character to lower case unless the first few characters are upper case as illustrated bellow

JavaBean-style property signature	Property name

`Integer getId()`
`void setId(Integer id)`
	`id`

`String getSSN()`
`void setSSN(Integer ssn)`
	`SSN`

For boolean properties, `is<PROPERTY_NAME>` can be used instead of the `set<PROPERTY_NAME>` method. Additionally, when placing mapping-related annotations on the properties, they must go on the getter method – it can not be applied to a setter one. Java persistence also requires `public` or `protected` visibility when placing mapping-related annotations on the properties.
In short, by following the convention over configuration, when placing mapping-related annotations

on the fields or properties, be consistent; that is, place either on the fields or properties. Otherwise, you will get an unpredictable behavior.

Access strategy and primary key

Because each entity needs, at least, one primary or composite key, the kind of access strategy can also be determined by the field or property which holds the database identifier or primary key. If a field is used to hold the database identifier, field access strategy will be used. Likewise, if a property is used to hold the database identifier, property access strategy will be used instead. A single primary key will be covered next. Detailed overview of primary and composite keys will be seen in the next chapter.

Primary key

To be consistent with database identity, an entity class must define a primary key. The simplest case is to define a single primary key by using the `@Id` annotation. It can be placed either on the field or on the JavaBean-style property you want to use as database identity or primary key. If you place the `@Id` annotation on some field, Java persistence will access the state of your entity by using its fields unless they be annotated with the `@Transient` annotation. Any property will be discarded by the Java persistence when accessing the state of your entity. The following types can be used as primary key:

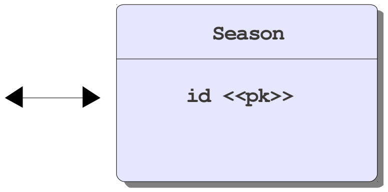
- Any Java primitive (`int`, `long`, `double`, `float` etc), primitive wrappers (`Integer`, `Long`, `Double`, `Float` etc)
- `java.lang.String`
- `java.util.Date`, `java.sql.Date`
- `java.math.BigDecimal`, `java.math.BigInteger` (both just supported as of version 2.x)

If you use other than these data types, your application will not be portable. If possible, avoid using floating point numbers as primary key. Back to our `Season` entity class, let's add a field called `id` to our entity class and annotate it with the `@Id` annotation.

```
import java.persistence.*;

@Entity
public class Season implements Serializable {

    @Id
    private Integer id;

}
```

We place the @Id annotation on the field called id which means it will be used to hold the database identifier or primary key. Java persistence requires that if you place mapping-related annotations, such as @Id, on the fields, they must be declared as private, protected or defaulted to the package visibility. The previous code is enough for generating the following

```
@Entity
public class Season implements Serializable {

  @Id
  private Integer id;


}
```

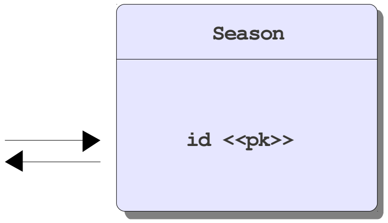


Because we place the @Id annotation on the field called id – you could use another name if you want -, Java persistence will access our entity by using its fields unless they be annotated with the @Transient annotation. This way, Java persistence will read from and write to the database by using its fields. Any property will be discarded by the Java persistence when accessing the state of your entity. By default, fields are mapped to columns of the same name so that the field called id will be mapped to a column called id. Because you can not provide public access to the fields of your entity class, you can encapsulate its fields, by using JavaBean-style properties. So, if you want to encapsulate your id field by using a JavaBean-style property - unless specified otherwise, from now on, we will refer to a JavaBean-style property as property -, you can rewrite your Season entity class as

```
import javax.persistence.*;

@Entity
public class Season implements Serializable {

    private Integer id;

    @Id
    public Integer getId() { return this.id; }
    public void setId(Integer id) { this.id = id; }

}
```

Now, we place the @Id annotation on the property called id which encapsulates a field of the same name – of course, you could encapsulate a field of another name. If you place the @Id annotation on some property, Java persistence will access the state of your entity by using its properties unless they be annotated with the @Transient annotation. Any field will be discarded by the Java persistence when accessing the state of your entity. Java persistence also requires public or protected visibility when placing mapping-related annotations on the properties. Like fields, properties are mapped to columns of the same name. Thus, the property called id will be mapped to a column called id. The previous code generates the following

```
@Entity
public class Season implements Serializable {

  private Integer id;

  @Id
  public Integer getId() { return this.id; }
  public void setId(Integer id) { this.id = id; }

}
```



Because we place the `@Id` annotation on the property called `id`, Java persistence will access our entity by using its properties. This way, Java persistence will read from and write to the database by using its properties. One advantage when encapsulating your field is to place some business logic in the getters and setters. For instance, you can validate its input so that the setter method accepts only positive integer value as follows

```
    @Id
    public Integer getId() { return this.id; }
    public void setId(Integer id) {
        if(id <= 0) {
            throw new IllegalStateException("Id property must be a positive");
        }

        this.id = id;
    }
```

Furthermore, draw your attention when placing mapping-related annotations on properties because the order in which Java persistence will populate your setters when reading from the database can deal with unpredictable behavior once some setters may access others properties that have not been initialized yet. Keep this in mind.

Adding fields and properties

We have seen a single version of the `Season` entity class. It contains only one property called `id` which is responsible for storing the database identity or primary key. Within our domain model, for each season is assigned a year. Hence, let's add a property called `year` to the `Season` class as follows

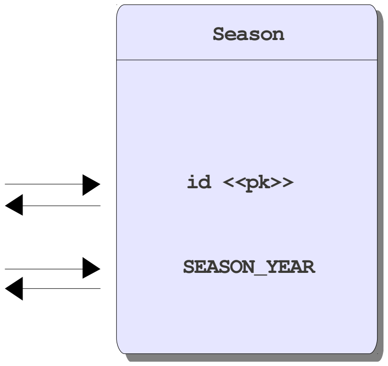
```
import javax.persistence.*;

@Entity
public class Season implements Serializable {

    private Integer id;
    private Integer year;

    @Id
    public Integer getId() { return this.id; }
    public void setId(Integer id) { this.id = id; }
    public Integer getYear() { return this.year = year; }
    public void setYear(Integer year) { this.year = year; }

}
```

The previous code uses property access strategy because we place the @Id annotation on the property called id. But if we attempt to export to some database – usually, Java persistence providers support this feature –, something goes wrong. This occurs because the year property has been mapped to a column called year which is a reserved SQL keyword. In addition to the @Table annotation, you can use the @Column to override its default column name by using its name attribute as follows

```
    @Column(name="SEASON_YEAR")
    public Integer getYear() { return this.year; }
```

Now, the year property will be mapped to a column called SEASON_YEAR. As shown before, we could delimit the reserved SQL keyword by using double quotes, but again, it is not recommended you use SQL reserved keywords as data object names, even if delimited by double quotes. The updated Season class will generate the following

```
@Entity
public class Season implements Serializable {

  private Integer id;

  private Integer year;

  @Id
  public Integer getId() { return this.id; }
  public void setId(Integer id) { this.id = id; }

  @Column(name= "SEASON_YEAR" )
  public Integer getYear() { return this.year; }
  public void setYear(Integer year) { this.year = year; }

}
```



Immutability

If you have a legacy system and do not own privileges to modify records, you can define an attribute of your entity class as immutable so that Java persistence neither inserts nor updates its state. To define an attribute of your entity class as immutable, you can use the attribute insertable, updatable, or both, of the @Column annotation. The insertable attribute mark a data

Nullable attributes

Useful when exporting your mapping, you can make use of the nullable attribute of the @Column annotation if you want to mark an attribute of your entity class as nullable. By default, its value is `true`, which means an attribute of your entity class can accept `null` values. However, if you do not desire a `null` value, you can set its value to `false` as follows

```
    @Column(name="SEASON_YEAR", nullable=false)
    public Integer getYear() { return this.year; }
```

Additionally, as of version 2.0, Java persistence API provides seamless integration with JSR-303 bean validation API. So, if you want, you can also make use of the `@NotNull` annotation of the JSR-303 bean validation API as an alternative to the `nullable` attribute as described bellow

```
    @javax.validation.constraints.NotNull
    @Column(name="SEASON_YEAR")
    public Integer getYear() { return this.year; }
```

Both declarations are equivalent except that, if you use the `@NotNull` annotation, your scope goes beyond the Java persistence once you can use JSR-303 bean validation outside your Java persistence application.

Discarding attributes

We have seen all of mapped attributes will be persisted unless they are marked with the `@Transient` annotation. Being so, let's introduce the class `Driver` which has a calculated property called `name` as follows

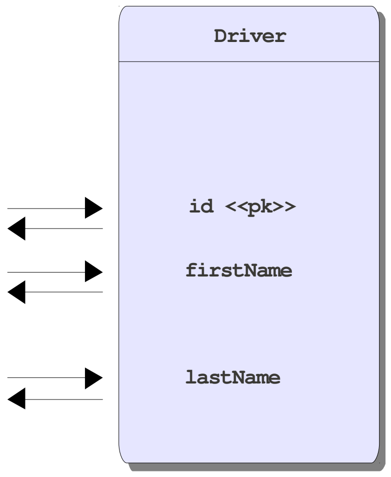
```
import javax.persistence.*;

@Entity
public class Driver implements Serializable {

    private Integer id;

    private String firstName;
    private String lastName;

    @Id
    public Integer getId() { return this.id; }
    public void setId(Integer id) { this.id = id; }

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

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

    public String getName() { return firstName + " " + lastName; }

}
```

Both properties `firstName` and `lastName` will be mapped to columns of the same name since we do not override their default column names by using the `@Column` annotation. However, we have a calculated property called `name` which depends on both `firstName` and `lastName` properties. Thus, we do not need to map the `name` property because it can be resolved by using the `firstName` and `lastName` properties. So, we place the `@Transient` annotation on the `name` property in order to avoid to be persisted as follows

```
    @Transient
    public String getName() { return firstName + " " + lastName; }
```

Now, the resulting mapping for the class `Driver` is described as follows

```
@Entity
public class Driver implements Serializable {

  private Integer id;

  private String firstName;
  private String lastName;

  @Id
  public Integer getId() { return this.id; }
  public void setId(Integer id) { this.id = id; }

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

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

  @Transient
  public String getName() {
    return this.firstName +   " " + this.lastName;
  }

}
```



Because we place the `@Transient` annotation on the property called `name`, Java persistence will ignore it.

Creating, Updating, Reading and Deleting

EntityManager

The `EntityManager` interface API supplies all of persistence-related operations such as creating, reading, updating and deleting (the so-called acronym CRUD). Outside a JEE environment, we use the `EntityManagerFactory` interface to create `EntityManager` instances. To obtain an `EntityManagerFactory`, we rely on the static method `createEntityManagerFactory` of class `Persistence` which takes as parameter the persistence unit name so that the process can be described as follows

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("racingPU");
EntityManager entityManager  = factory.createEntityManager();
```

However, creating an `EntityManagerFactory` is a heavyweight process because it needs to scan for annotated entity classes, load all of persistence-related information such as connection settings, properties and so on. Such information are grouped by a persistence unit. A persistence unit is declared in a file called `persistence.xml` whose root element is `persistence`. A sample persistence unit enough for running a Java SE application is declared as follows

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
             xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                                 http://java.sun.com/xml/ns/persistence/persistence_1_0.
xsd">
    <persistence-unit name="racingPU">
        <provider>oracle.toplink.essentials.PersistenceProvider</provider>
        <class>racing.model.domain.Season</class>
        <properties>
            <property name="toplink.jdbc.driver" value="org.h2.Driver"/>
            <property name="toplink.jdbc.url" value="jdbc:h2:mem:racing"/>
            <property name="toplink.jdbc.user" value="root"/>
            <property name="toplink.jdbc.password" value="root"/>
            <property name="toplink.ddl-generation" value="drop-and-create-tables"/>
        </properties>
    </persistence-unit>
</persistence>
```

First of all, the root element `persistence` requires its `version` attribute because there is a one-to-one correspondence between its value and the Java persistence version used by your application. As you can see, a `persistence-unit` element group a set of persistence-related attributes: a provider responsible for supplying `EntityManager` instances, entity classes, and vendor-specific properties. The `provider` element specifies which class is responsible for supplying EntityManager instances. Each persistence unit can define its own provider. Although you do not need to list your entity classes since Java Persistence will scan for annotated entity classes, you should explicitly define your entity classes to ensure portability among Java SE applications by using the `class` element. Additionally, the `properties` element declare both standard and vendor-specific properties used by a persistence unit. If some property is not recognized by the persistence provider, it will be ignored. Here, we are just using vendor-specific properties. In fact, just as of version 2.0, it was defined the following standard properties for Java SE applications

Property	Description
`javax.persistence.jdbc.driver`	fully-qualified name of the driver class

javax.persistence.jdbc.url	url of the database to which to connect
javax.persistence.jdbc.user	database user
javax.persistence.password	database user's password

So, if you code an application in compliance with the Java persistence version 2.0, you can re-write your `persistence.xml` file as follows

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
             xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                                 http://java.sun.com/xml/ns/persistence/persistence_2_0.
xsd">
```

The persistence.xml version 2.0 designed for Java SE applications differs from its 1.0 counterpart in that you must update the version attribute of the persistence element to reflect the version used. It is up to you whether to replace or not the vendor-specific properties driver, url, user and password by its equivalent standardized ones.

Putting our code into action

Now we have seen some core concepts addressing the Java persistence, it is time to run a sample application. To keep our code as simple as possible, we rely on the H2 database, an open-source SQL database written in Java which supports in-memory database, ideal for running our application, available in a single jar file.
Because creating an `EntityManagerFactory` is a heavyweight process, we can define a helper class which supplies a singleton object as follows

```
package racing.utils.persistence;

import javax.persistence.*;

public class EntityManagerFactoryHelper {

    private static final EntityManagerFactory singleton;

    static {
        try {
            singleton = Persistence.createEntityManagerFactory();
        } catch (Throwable e) {
            throw new ExceptionInInitializerError(e);
        }
    }

    public static EntityManagerFactory getEntityManagerFactory() {
        return singleton;
    }
}
```

You can use only one EntityManagerFactory per application once it is thread-safe. We initialize our singleton EntityManagerFactory object in a static block because it will be executed as soon as our helper class be loaded. The ExceptionInInitializerError thrown in the catch block reports an unexpected exception.

Scala

```
import javax.persistence._;

@Entity
class Season {

    @Id
    var id:Int = _;

    @Column(name="SEASON_YEAR")
    var year:Int = _;

}
```

Scala, by default, does not generate JavaBean-style properties when compiled into .class files. If you wish, Scala can generate JavaBeans-style properties by annotating each field declaration with the @BeanProperty annotation – for boolean properties, if you prefer is<PROPERTY_NAME> to use set<PROPERTY_NAME>, place @BooleanBeanProperty instead - as follows

```
    @Id
    @BeanProperty var id:Int = _;

    @Column(name="SEASON_YEAR")
    @BeanProperty var year:Int = _;
```

However, the @BeanProperty annotation is available as of version 2.8 of Scala. Being so, if you use an earlier version of Scala, you must explicitly declare JavaBean-style properties as follows

```
import javax.persistence._;

@Entity
class Season {

    var id:Int = _;
    var year:Int = _;

    @Id
    def getId():Int = { return this.id; }
    def setId(id:Int):Unit = { this.id = id; }

    @Column(name="SEASON_YEAR")
    def getYear():Int = { return this.year; }
    def setYear(year:Int):Unit = { this.year = year; }

}
```

Additionally, unless you declare JavaBean-style properties, Java persistence will make use of field access strategy to access your entity. Since we map our entity class, let's create our helper class, responsible for supplying our heavyweight `EntityManagerFactory` object.

```
object EntityManagerFactoryHelper {

    val entityManagerFactory:EntityManagerFactory = 
Persistence.createEntityManagerFactory();

    def getEntityManagerFactory():EntityManagerFactory = {
        return entityManagerFactory;
    }

}
```

In short, the `object` keyword is the Scala way of declaring static fields and methods. Basically, you can think of our Scala helper as its Java counterpart. The field which holds an instance of type `EntityManagerFactory` is declared as immutable by using the `val` keyword. In addition, we add a read-only property, following the JavaBean-style pattern, which encapsulates our field.

Groovy

Groovy language arise as a lightweight version of Java, discarding many of the boilerplate code needed by Java applications.

```
import javax.persistence.*;

@Entity
class Season implements Serializable {

    @Id
    Integer id;

    @Column("SEASON_YEAR")
    Integer year;

}
```

As you should know, a single field declaration in Groovy encapsulates a JavaBean-style property so that when compiled into a .class file, getters and setters will be automatically generated. In addition, we do not need to import `java.io.Serializable` because the package `java.io`, among other commonly used packages for creating Java applications, is automatically imported by Groovy. But a question arise: when compiled into a .class file, will Java mapping-related annotations be placed on the fields or properties ? In other words, will Java persistence access the state of our entity by using its fields or properties ? like Scala, mapping-related annotations will be placed on the fields when compiled into .class file. Hence, Java persistence will access the state of our entity by using field access strategy.

Although, in Groovy, you can use the same style of String concatenation used in Java, we stick to the Groovy syntax, a cleaner and simple approach. To give you an overview about how it works, let's re-write our calculated read-only property `name` from the class `Driver` by using Groovy syntax as follows

```
import javax.persistence.*;

class Driver implements Serializable {

    @Id
    Integer id;

    String firstName;
    String lastName;

    String getName() {
        return "${firstName} ${lastName}";
    }

}
```

In the jargon of Groovy, any String can be surrounded by either double or single quotes. If you surround your String by double quotes, any expression delimited by `${` and `}` will be evaluated so both properties `firstName` and `lastName` will be resolved. However, if you use single quotes, the String will be evaluated as a pure literal. For instance, if you had delimited your String as `'${firstName} ${lastName}'`, you will get as result *`${firstName} ${lastName}`*. Keep this in mind.

```
import javax.persistence.*;

class EntityManagerFactoryHelper {

    private static final EntityManagerFactory entityManagerFactory;

    static {
        entityManagerFactory = Persistence.createEntityManagerFactory("racingPU");
    }

    static EntityManagerFactory getEntityManagerFactory() {
        return entityManagerFactory;
    }

}
```

Our helper responsible for creating a singleton instance of EntityManagerFactory follows a syntax similar to its Java counterpart except that we supply a read-only static property. When we supply our getters or setters, or both, Groovy does not generate getters and setters for us.

Mapping using XML

Mixing both strategies

But if you need flexibility, as of version 2.0, you can rely on the @Access annotation. You can define its default behavior by putting the @Access annotation on it.

We defined the default property access strategy by placing @Access(AccessType.PROPERTY) on class declaration. Notice that when applied to a class, the @Access annotation defines the default access strategy of that class. It does not affect the behavior of others classes. The @Access annotation accepts two values: either AccessType.PROPERTY or AccessType.FIELD.

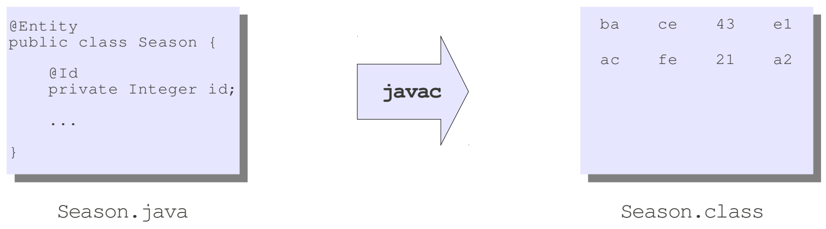
If you need to access both field and property strategy, you can override the default access strategy by placing @Access(AccessType.PROPERTY) on the properties for which you want property access strategy or on the fields for which you want field access strategy. One requirement when mixing both strategies is to define its default access strategy by placing @Access on class declaration. If possible, because of consistency issues, prefer to stick either to field or property annotation within one entity class.
Back to our

Design issues

To keep our discussion about the requirements needed to mark a class as a entity class, it must

- have a public or protected no-arg constructor.
- not be marked as `final`. Additionally, any persistent attribute can not be marked as `final` as well.
- be a first-level class; that is, it can not be an interface or enum.

We will cover in detail the first two items in order to give you some insight about how the Java persistence works under the hood. As you should know, any Java source file is compiled into a standardized and portable bytecode file, known by its .class extension. The process can be described as follows



The Java language itself does not provide built-in support for generating .class files at runtime. However, because .class files follow a standardized format – for further information on .class file format, please refer to the Java virtual machine specification -, some libraries can generate and load .class files at runtime.

Although Java persistence does not impose any design pattern for building an implementation, providers usually make use of entity subclasses to monitor your entity. The next code shows you a dummy implementation

```
public class SeasonAsProxy extends Season implements Serializable {

    private boolean modified = false;

    private StringBuilder query = new StringBuilder("UPDATE SEASONS SET ");

    @Override
    public void setYear(Integer year) {
        super.setYear(year);

        this.modified = true;
        query.append("SEASON_YEAR = ")
             .append(year);
    }

    public boolean isModified() {
        return this.modified;
    }

    public String getQueryAsString() {
        return query.toString();
    }

}
```

First of all, it does not reflect any Java persistence provider. The purpose of the previous code is to give you some insight about how generated proxies work behind the scenes. Because each proxy inherits from an entity class in order to add persistence functionality, it must not be marked as `final` (`final` keyword at class-level does not allow inheritance). Likewise, persistent attributes can not be marked as `final` once they need to be overriden as well. Finally, `public` or `protected` visibility needed by persistent properties is because a subclass inherits all of the `public` and `protected` properties of its parent, no matter what package the subclass is in.

If you see the previous code, our proxy has an encapsulated field called `modified` used as a flag in order to determine whether our entity has been modified. Thus, if we attempt, for example, to find a Season entity whose primary key is `7` and set up its `year` property as follows

```
Season season = entityManager.find(Season.class, 7);
season.setYear(2014);
```

Do not worry about the `EntityManager` object at this time. For now, you just need to know its API is used to perform persistence-related operations over entities. Here, we are using its `find` method which takes as parameter the entity class and a primary key. Its return is a managed entity, if any. Otherwise, it returns `null`. You can think of a managed entity as a instance whose state is tracked by the Java persistence. If it finds any change, the state of our entity is transparently synchronized with the database. Because we changed the `year` property, Java persistence will, under the hood, issue the following SQL command

```
UPDATE Season SET SEASON_YEAR = 2014 WHERE ID = 7
```

The previous model works fine if you model your entity classes by using property access strategy once proxies can intercept method calls as shown above. However, if you use field access strategy, the same rule is not valid (Java does not support interception at level of field). To bypass this constraint, it can record the loaded entity and performs a side-by-side field comparison before deciding whether to update or not our entity as follows

```
public class SeasonAsProxy extends Season implements Serializable {

    private Season loadedEntity;

    public Season getLoadedEntity() {
        return this.loadedEntity;
    }

}
```

Now, the Season entity returned by the `find` method was assigned to the field called `loadedEntity`. Doing so, Java persistence can compare the persistent fields of our loaded entity with those from its superclass in order to decide whether to update or not our entity (usually, making use of Java reflection).

No-arg public or protected constructor

A class can have different constructors. The number of constructors depends on your business logic and your needs. For instance, we could re-write the `Season` class as follows

```
import javax.persistence.*;

@Entity
public class Season implements Serializable {
    @Id
    private Integer id;

    @Column(name="SEASON_YEAR")
    private Integer year;

    public Season(Integer id) {
        this.id = id;
    }
    public Season(Integer id, Integer year) {
        this.id = id;
        this.year = year;
    }

}
```

We add two constructors to our Season class. One accepts two arguments: the id and year of the Season. The other accepts only its id. Now the Season class has one question: which constructor should the Java persistence call to instantiate a Season entity ? As you have seen, Java persistence providers create subclasses of entity classes in order to track its changes. However, for each new instantiated object, its constructor should call, either implicitly or explicitly, some constructor from its superclass as follows

```
public class SeasonAsProxy extends Season implements Serializable {

    public SeasonAsProxy() {
        // Which constructor from Season should I call ?
    }

    ...
}
```

For the sake of simplicity, we omitted the remaining code. Once we explicitly added constructors to our Season class, Java compiler does not include its default no-arg `public` constructor. In addition, if no explicit call to some constructor from its superclass is provided, Java will, by default, call its no-arg constructor, if any; otherwise, you will get an error. Because our entity class have different constructors, Java persistence can not guess what each constructor parameter means so that it requires a `public` or `protected` no-arg constructor to instantiate an entity. Hence, we have to add a new `public` or `protected` no-arg constructor to our Season class.