Primary key

A primary key uniquely identifies a record from a given table. A single primary key uses only one column of a given table to uniquely identify a record.

// insert database snapshot image right here

A single primary key

We have seen we declared the `id` field of the class Season as its primary key. Our goal has been achieved by placing the `@Id` annotation on the field called `id` – of course, you could use another name if you wish.

```
package racing.model.domain;

import javax.persistence.*;

@Entity
public class Season implements Serializable {

    @Id
    private Integer id;

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

}
```

The `@Id` annotation specifies which field or property will uniquely identity our entity from a database perspective. The field or property on which has been placed the `@Id` annotation is called primary key field or property, depending upon the placement of the `@Id` annotation on a field or property, respectively. In the `Season` class, the `id` field has been used as primary key.

Auto generated primary key

If you just place the `@Id` annotation on a field or property, you must explicitly set up its value because the `@Id` annotation itself is not enough to provide automatically generated values. If you desire automatically generated values for your primary key field or property, you need to use the `@GeneratedValue` annotation which is used for this purpose. Being so, let's review our `Season` class and add the `@GeneratedValue` annotation to its primary key field as follows

```
    @Id
    @GeneratedValue
    private Integer id;
```

By default, the `@GeneratedValue` annotation picks up the best strategy according to the target database to automatically generate a primary key value for a given entity. Additionally, it has an

attribute called `strategy` which provides more control

Composite keys

So far, we have seen single primary keys, but if you deal with legacy systems, you will commonly make use of composite keys. Unlike a single primary key, it uses more than one column in order to define a primary key, also called composite primary key. In addition, Java persistence provides built-in support for composite primary key. However, one requirement to make use of composite key is to declare a class, sometimes called composite key class, which holds the fields or properties used to map your composite key. To start our discussion, let's review our class `Driver`

```
import javax.persistence.*;

@Entity
public class Driver implements Serializable {

    @Id
    private Integer id;

    @Embedded
    private Name name;

}
```

The `Driver` class declares an `id` field, which holds a single primary key, and an embedded field of the class Name called `name`, which contains only two fields, `firstName` and `lastName`. Now, imagine we have a legacy system which uses `firstName` and `lastName` as primary keys. Since our `Name` class declare both fields, we can use it as a primary key class and so we can discard our field `id` and re-write our class as follows

```
import javax.persistence.*;

@Entity
public class Driver implements Serializable {

    @EmbeddedId
    private Name name;

}
```

In short, you can think of an `@EmbeddedId` field as a special `@Embedded` one whose fields of its embeddable class also play the role of primary key. Now, because the fields of our embeddable class called `Name` will be used as database identifiers, we need to replace the `@Embedded` annotation by the `@EmbeddedId`. However, each composite key class needs to override both `equals` and `hashCode` methods, apart from implementing the `Serializable` interface. Being so, let's add `equals` and `hashCode` methods to our class `Name`

```
import javax.persistence.*;
```

```
@Entity
public class Driver implements Serializable {

    @EmbeddedId
    private Name name;

    @Embeddable
    public static class Name implements Serializable {

        private String firstName;
        private String lastName;

        @Override
        public boolean equals(Object o) {
            if(!(o instanceof Name))
                return false;

            Name other = (Name) o;
            return new EqualsBuilder()
                    .append(firstName, other.firstName)
                    .append(lastName, other.lastName)
                    .isEquals();
        }

        @Override
        public int hashCode() {
            return new HashCodeBuilder()
                    .append(firstName)
                    .append(lastName)
                    .hashcode();
        }

    }
}
```

As you can see, the `Name` class keeps its basic skeleton, except that we add `equals` and `hashCode` methods. In both methods, we use `EqualsBuilder` and `HashCodeBuilder` classes from Jakarta commons library to help us to implement our methods. Also notice a kind of programming style known as method chaining. Although it can sound strange at a first glance, it can avoid a lot of unnecessary object reference such as shown below

```
        public boolean equals(Object o) {
            if(!(o instanceof Name))
                return false;

            Name other = (Name) o;
            EqualsBuilder equalsBuilder = new EqualsBuilder();
            equalsBuilder.append(firstName, other.firstName);
            equalsBuilder.append(lastName, other.lastName);

            return equalsBuilder.isEquals();
        }
```

In short, you can think of an `@EmbeddedId` field as a particular `@Embedded` one whose fields of

its embeddable class also play the role of database identifiers as illustrated below

Surrogate and business keys

Composite key made up of both surrogate and business keys

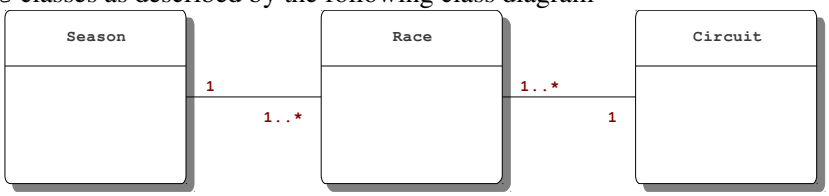
In order to publish each Team, a set of images is available for viewing.

```
import javax.persistence.*;

@Entity
public class TeamGallery implements Serializable {

    @EmbeddedId
    private TeamGalleryId id;

    @Column(insertable=false, updatable=false)
    private String fileName;

    @Embeddable
    public static class TeamGalleryId implements Serializable {

        private Integer teamId;
        private String fileName;

    }

}
```

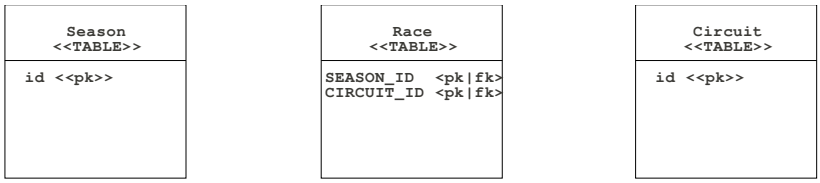
Composite key made up of surrogate keys

Specially found in legacy systems, composite primary key composed of surrogate keys

Getting back our domain model, the `Race` class has a `@ManyToOne` relationship with the `Season` and `Circuit` classes as described by the following class diagram



Strictly speaking, one season can have many races whereas one circuit can be assigned to many races. From the point of view of the database, the `Race` table provides foreign key references to the primary key of the `Season` and `Circuit` tables. In addition, let's assume for the purpose of our example that both foreign key columns have been also used as primary key of the `Race` table according to the database schema shown bellow



The `Race` table declares two foreign key columns, `SEASON_ID` and `CIRCUIT_ID`. The former references the primary key of the `Season` table whereas the later is a foreign key field to the primary key of the `Circuit` table. Additionally, both fields form the primary key of the `Race` table. As shown earlier, we have seen that a surrogate key is not meaningful to our application and so you do not want to end up with something like

```
public class Race implements Serializable {

    @Column(name="SEASON_ID", updatable=false, nullable=false)
    private Integer seasonId;
    @Column(name="CIRCUIT_ID", updatable=false, nullable=false)
    private Integer circuitId;

    private Date qualifyingDate;
    private Date raceDate;
    private Integer numberOfLaps;

}
```

Unlike our class diagram, which defines only state regarding our domain model, we introduced two foreign key fields, `seasonId` and `circuitId`, as attributes of our class. However, both fields have nothing to do with our domain model, so we need to get rid of them. On the other hand, `seasonId` and `circuitId` are used as primary key, and, as we have seen, a composite primary key requires a primary key class.

The next version of the `Race` class declare a static inner class called `RaceId`, used as primary key class of our entity class, which defines `seasonId` and `circuitId`.

```
import javax.persistence.*;

@Entity
public class Race implements Serializable {

    @EmbeddedId
    private RaceId id;

    private Date qualifyingDate;
    private Date raceDate;
    private Integer numberOfLaps;

    @Embeddable
    public static class RaceId implements Serializable {
```

```
        @Column(name="SEASON_ID", updatable=false, nullable=false)
        private Integer seasonId;
        @Column(name="CIRCUIT_ID", updatable=false, nullable=false)
        private Integer circuitId;

        public boolean equals(Object o) {
            if(!(o instanceof RaceId))
                return false;

            if((seasonId == null) && (circuitId == null))
                return false;

            RaceId other = (RaceId) o;
            return seasonId.equals(other.seasonId) && circuitId.equals(other.circuitId);
        }

        public int hashCode() {

        }

    }

}
```

If you compare the former and the later version of the `Race` class, you will see that the foreign and primary key fields of the former version have been replaced by a `@EmbeddedId` field of the `RaceId` class. Just like in `@Embeddable`, the `@EmbeddedId` requires an `@Embeddable` class, in our case, `RaceId`, which will be used as primary key class of the `Race` class. Additionally, all of the fields of the `@Embeddable` class will be used as primary key. As we have seen earlier, the `@EmbeddedId` annotation is a particular `@Embedded` annotation where the fields of its `@Embeddable` class function as primary key as illustrated bellow

// highligh about static inner class

The previous code declare an outer and a static inner class, called `Race` and `RaceId`, respectively. The `RaceId` class is declared as a static inner class because it can be instantiated regardless of its outer class. Static inner classes are useful when both outer and inner classes are logically grouped. In our case, the static inner class is the primary key class of the outer entity class. But if you wish, feel free to declare your `RaceId` as a non-static inner class; that is, as a stand-alone class.

Scala

The Scala language does not have a static reserved keyword such as in Java. As a consequence, we can not declare a static inner class in Scala. However, Scala allows you declare more than one public class in a single `.scala` file, which can logically group `Race` and `RaceId` classes as follows

```
package racing.domain.model;

import javax.persistence._;
import java.io.Serializable;
import java.util.Date;
```

```
@Entity
class Race implements Serializable {

    @EmbeddedId
    @BeanProperty var id:RaceId = _;

    @BeanProperty var qualifyingDate:Date = _;
    @BeanProperty var raceDate:Date = _;
    @BeanProperty var numberOfLaps:Integer = _;

}

@Embeddable
class RaceId implements Serializable {

    @BeanProperty var seasonId:Integer = _;
    @BeanProperty var circuitId:Integer = _;

}
```

If we declare `Race` and `RaceId` classes in a single `.scala` file such as `Race.scala`, for example, both classes will reside in the `racing.domain.model` package. This is possible because we declared our package statement at the top of the file, which is applied to all classes of our file. In addition, Scala does not enforce a directory structure like in Java. Being so, you do not have to put our previous `.scala` file into a `racing/domain/model` directory.

Groovy

```
import javax.persistence.*;

@Entity
class Race implements Serializable {

    @EmbeddedId
    RaceId id;

    // does Groovy support static inner class ???

}
```

Like Scala, you can also declare more than one public class in a single `.groovy` class. So, if you wish, you can also declare your `Race` and `RaceId` classes as follows

```
import javax.persistence.*;

@Entity
class Race implements Serializable {

    @EmbeddedId
    RaceId id;

}
```

```
@Embeddable
class RaceId implements Serializable {

    Integer seasonId;
    Integer circuitId;

}
```

Composite key made up of business keys

In the `Driver` class, we have used an embedded field of the class `Name`, which declares two fields, `firstName` and `lastName`. Suppose now we prefer to use `firstName` and `lastName` as such to use an embedded field. In addition, we want to use both fields as primary key as follows

```
@Entity
public class Driver implements Serializable {

    @Id
    private String firstName;
    @Id
    private String lastName;

}
```

The previous code shows we are using single fields as primary key instead of an embedded id field used earlier. In addition, we have seen a must requirement in order to use a composite primary key is to define a primary key class, which can be fulfilled by our `Name` class. However, if you desire single fields as primary key, you must supply a hint to the `Driver` entity class so that it can identify which class should be used as its primary key class. Such a hint is the `@IdClass` annotation, placed on the class declaration. The `@IdClass` annotation requires you provide the class you want to use as primary key class as its value as shown bellow

```
import javax.persistence.*;

@Entity
@IdClass(Name.class)
public class Driver implements Serializable {

    @Id
    private String firstName;
    @Id
    private String lastName;

}
```

We place the `@IdClass` annotation, which takes as its value our primary key class, `Name.class`, on the class declaration. But when you desire to use single fields as primary key, it is required a one-to-one correspondence between the names of our primary key fields of the `Driver` class and those

of the primary key class; that is, if, in the class `Driver`, we define `firstName` and `lastName` as primary key fields, the names of the fields declared in the primary key class must also be `firstName` and `lastName`. Otherwise, your application will not work as expected.

Scala

The Scala language itself does not differ so much from its Java counterpart. The only difference is because when we refer to a class in Java, for instance, `Name.class`, in Scala, we use the `classOf` method of the implicit imported `Predef` object and so our Scala version of the `Driver` class becomes

```
import javax.persistence._;

@Entity
@IdClass(classOf[Name])
public class Driver implements Serializable {

    @Id
    @BeanProperty String firstName;

    @Id
    @BeanProperty String lastName;

}
```

Notice how we refer to the value of the `@IdClass` annotation: `classOf[Name]` without the `.class` suffix. Additionally, we do not need to import `scala.Predef` object because it is implicitly imported by Scala.

Groovy

In Groovy, we use its standard syntax

```
import javax.persistence.*;

@Entity
@IdClass(Name.class)
class Driver implements Serializable {

    @Id
    String firstName;

    @Id
    String lastName;

}
```

The `Driver` class, written in Groovy, does not differ so much from its Java counterpart, except that we rely on default public visibility for classes and fields. Moreover, Groovy compiler generates JavaBean-style getters and setters for our fields, `firstName` and `lastName`.

XML