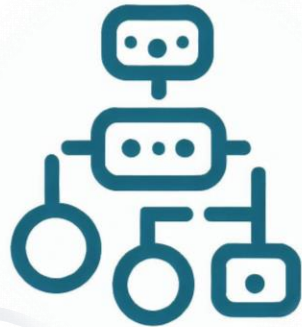


# Models Inheritance and Customization



SoftUni Team  
Technical Trainers



**SoftUni**



Software University

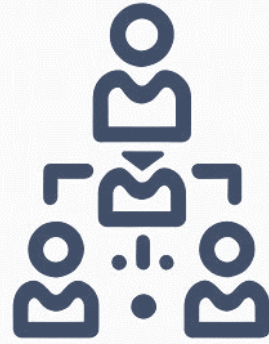
<https://softuni.bg>

[sli.do](https://sli.do)

**#python-db**

1. Model Inheritance
  - Multi-table Inheritance
  - Abstract Base Classes
  - Proxy Models
2. Model Methods
  - Built-in Methods
  - Custom Methods
3. Custom Fields
  - Custom Field Built-in Methods





# Model Inheritance

# Inheritance of Models

- **Model inheritance** allows you to create a **new model** based on an **existing** one
- The **new model (child)**
  - has **all the fields and methods** of the **original model (parent)**
  - can also **define its own additional fields and methods**



# Types of Model Inheritance

- There are **three types of model inheritance** in Django
  - **Multi-table Inheritance**
    - Both parent and child models generate database tables
  - **Abstract Base Classes**
    - The abstract model (parent) does **not** generate a database table
  - **Proxy Models**
    - The proxy model (child) does **not** generate a database table



- **Multi-table inheritance** creates
  - a **separate** database **table** for **each model** in the **inheritance chain**
- **Each table** includes
  - **fields** from **all** the **parent** models in the **hierarchy**
- Django **automatically** generates
  - a **OneToOneField** field for the **relationship** in the **child's** model to its **parent**

```
from django.db import models

class ParentModel(models.Model):
    parent_field = models.CharField(max_length=50)
```

A field that a child will inherit  
from its parent

```
class ChildModel(ParentModel):
    child_field = models.IntegerField()
```

Own field that only child has



- You are given an empty **ORM project skeleton** (you can download it from [here](#)) needed to create a **Zoo Management System**
- First, in the **main\_app** create 4 models called "**Animal**", "**Mammal**", "**Bird**", and "**Reptile**"
- A full description of the problem can be found in the Lab document [here](#)

# Solution: Zoo Animals

```
class Animal(models.Model):
    name = models.CharField(max_length=100)
    species = models.CharField(max_length=100)
    birth_date = models.DateField()
    sound = models.CharField(max_length=100)

class Mammal(Animal):
    fur_color = models.CharField(max_length=50)

class Bird(Animal):
    wing_span = models.DecimalField(max_digits=5, decimal_places=2)

class Reptile(Animal):
    scale_type = models.CharField(max_length=50)
```

- **Abstract models**
  - are **base classes**
  - allow **other models** to **inherit** fields and **methods** from them
  - do **not** create their **own** database **tables**
  - act as **templates** for **other models** to **reuse** common fields and **behavior**

```
from django.db import models
```

```
class AbstractBaseModel(models.Model):  
    common_field = models.CharField(max_length=100)
```

```
class Meta:
```

```
    abstract = True
```

A field that a child will inherit  
from its abstract parent

No database table will be created

```
class ChildModel(AbstractBaseModel):  
    additional_field = models.IntegerField()
```

Own field that only child has

# Class Meta

- Use the **inner class Meta**
  - to insert **metadata** into the model
- Adding **Meta** inner class is **optional**

```
class PersonBaseClass(models.Model):  
    ...  
    age = models.IntegerField()  
  
    class Meta:  
        abstract = True
```

Turns the model into an  
Abstract Base Class

Meta option

\*Note: **Meta options** will be the subject of an article in the next presentation

# Problem: Zoo Employees

- In the `main_app` create an **additional model** called **"Employee"**
  - It is a **base class** for any **type of employee** in the zoo
  - It is **NOT** meant to **create a database table** on its own
- Then, **create 2 more models: "ZooKeeper" and "Veterinarian"**
  - They are **types of employees**
- A full description of the problem can be found in the Lab document [here](#)

# Solution: Zoo Employees

```
class Employee(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    phone_number = models.CharField(max_length=10)

    class Meta:
        abstract = True

class ZooKeeper(Employee):
    # Add the predefined choices in the SPECIALITIES variable
    specialty = models.CharField(max_length=10, choices=SPECIALITIES)
    managed_animals = models.ManyToManyField('Animal')

class Veterinarian(Employee):
    license_number = models.CharField(max_length=10)
```

- **Proxy models** allow you to create a **new model**
  - that **behaves exactly like** an existing **model**
  - with some **customizations** added
- The **proxy model** uses the **same** database **table** as the **original model**
- Useful when adding
  - **extra methods, managers, or custom behavior** to an **existing model** **without** modifying the **original model**



```
from django.db import models

class OriginalModel(models.Model):
    ...
    field = models.CharField(max_length=50)
```

Original model fields

```
class ProxyModel(OriginalModel):
    ...
    class Meta:
        proxy = True
```

Some extra methods

No new table will be created

# Problem: Animal Display System

- In the `main_app` create one **additional model** called **"ZooDisplayAnimal"**
  - It inherits from the **"Animal"** model but does **NOT** have its **own database table**
  - Its primary purpose is to **extend** the **"Animal"** model **behavior**
  - Currently, it is **NOT needed to add additional logic** to the model

# Solution: Animal Display System

```
class ZooDisplayAnimal(Animal):  
    class Meta:  
        proxy = True
```





# Model Methods

Built-in Methods, Custom Methods

# Model Methods

- **Model methods** are functions defined **within** a Django model
- They allow you to **perform**
  - operations on model instances
  - other tasks related to the model
- Types of **model methods**
  - **Built-in** methods
  - **Custom** methods



- **Built-in Methods** are standard methods provided by
  - Django's **models.Model** class
- Main **built-in methods**
  - **save()**
    - Called when **saving** an instance to the database
  - **clean()**
    - Used for data validation **before** saving

More at: <https://docs.djangoproject.com/en/5.0/topics/db/models/#model-methods>

# Overriding Predefined Model Methods

- **Override** built-in methods to add
  - **custom behavior** or **validation** to a model

```
from django.db import models
```

```
class MyModel(models.Model):
```

```
    field = models.CharField(max_length=100)
```

```
    def save(self, *args, **kwargs):
```

```
        ...
```

```
        super().save(*args, **kwargs) # Call the original save method
```

```
    def clean(self):
```

```
        ...
```

Custom logic before saving

Custom validation logic

- In the "ZooKeeper" model add a **custom validation** logic **before** each zookeeper object is **saved**
  - Create a **validation** to ensure that the object is **checked against the given list of valid choices ("SPECIALITIES")**
  - If the **specialty** is **not** a valid choice, a **ValidationError** should be raised with the message: **"Specialty must be a valid choice."**



# Solution: Zookeeper's Specialty

```
from django.core.exceptions import ValidationError

class ZooKeeper(Employee):
    ...

    def clean(self):

        choices = [choice[0] for choice in self.SPECIALITIES]
        if self.specialty not in choices:
            raise ValidationError(
                "Specialty must be a valid choice."
            )
```

- **Custom Model Methods**
  - **Additional** methods defined in a model
  - Performing **specific tasks** or **calculations related** to the **model**
  - Acting on a particular **model instance**
  - Keeping **business logic** in **one place**

```
from django.db import models

class MyModel(models.Model):
    field = models.CharField(max_length=100)

    def custom_method(self):
        ...
```

Custom model method

Custom logic

# Problem: Animal Display System Logic

- It is time to add logic to the **"ZooDisplayAnimal"** model
  - It is designed to create a **customized** view of animal data exclusively for visitors
- Your task is to **implement two custom methods** **"display\_info"**, and **"is\_endangered"**
- A full description of the problem can be found in the Lab document [here](#)

# Solution: Animal Display System Logic

```
class ZooDisplayAnimal(Animal):  
    class Meta:  
        proxy = True  
  
    def display_info(self):  
        return f"Meet {self.name}! Species: {self.species},  
born {self.birth_date}." \  
            f" It makes a noise like '{self.sound}'."
```

# Solution: Animal Display System Logic

```
class ZooDisplayAnimal(Animal):  
    ...  
  
    SPECIES_AT_RISK = ["Cross River Gorilla", "Orangutan",  
"Green Turtle"]  
  
    def is_endangered(self):  
        danger = self.species in self.SPECIES_AT_RISK  
        return f"{self.species} is at risk!" if danger \  
            else f"{self.species} is not at risk."
```

- **Custom model properties** allow you to
  - define **new attributes** for a **model** that are
    - **not stored** in the **database**
    - **calculated** or **derived** from **existing** model fields
- They are similar to **regular model fields**
  - but **do not correspond** to **database columns**
  - defined as **Python class properties**



# Custom Model Properties

- To create a **custom model property**
  - use the **@property** decorator in Python
- The decorator allows you to **define** a **method** that
  - acts as a **property**
  - does **not** require a database column

```
class Employee(models.Model):  
    birth_date = models.DateField()  
    ...  
  
    @property  
    def age(self):  
        ... # Returns the  
calculated age
```



# Problem: Animal's Age

- In the "**Animal**" model implement **one property** that **calculates and returns the age** of an animal based on its birth date
- The age is **dynamically** calculated each time, ensuring that it remains accurate over time



# Solution: Animal's Age

```
class Animal(models.Model):  
    ...  
  
    @property  
    def age(self):  
        today = date.today()  
        age = today.year - self.birth_date.year - (  
            (today.month, today.day) <  
            (self.birth_date.month, self.birth_date.day))  
        return age
```



**Custom Fields**

# Custom Fields

- Django allows you to create **custom fields** by **subclassing**
  - **`django.db.models.Field`**
    - or one of the **existing field classes**
      - **`models.CharField`, `models.IntegerField`**, etc.
- **Custom fields** can be **helpful** when using
  - **custom data type**
  - **validation**
  - **serialization** for your model fields



# Custom Fields Built-in Methods

- Django provides **several built-in** custom **field methods** that you can **override** to
  - **customize** the **behavior** of the custom model field
- Some of the most useful **built-in custom field methods**
  - **from\_db\_value()**
    - Converts the field's **value** as retrieved **from** the **database** into its **Python representation**
  - **to\_python()**
    - Converts the field's **value** from the **serialized format** (usually as a string) into its **Python representation**



# Custom Fields Built-in Methods

- `get_prep_value()`
  - Prepares the field's value **before saving** it to the database
- `validate()`
  - Performs **custom validation** on the field's value
- `deconstruct()`
  - Used when serializing the **field** to store its constructor **arguments as a tuple**, allowing Django to **recreate** the **field** when migrating or serializing models



**\*Note:** you do **not** need to override all of these methods for every custom field

# Custom Field Example

```
from django.db import models
```

```
class CustomField(models.Field):
```

Overriding some built-in methods

```
    def to_python(self, value):
```

```
        # Custom data conversion logic
```

```
        ...
```

```
    def get_prep_value(self, value):
```

```
        # Custom value preparation for database storage
```

```
        ...
```

```
class MyModel(models.Model):
```

```
    custom_field = CustomField()
```

An instance of the custom field

# Custom Phone Field - Example

```
from django.db import models
```

```
class PhoneNumberField(models.CharField):
```

```
    def __init__(self, *args, **kwargs):
```

```
        kwargs['max_length'] = 15
```

Defining a max-length

```
        super().__init__(*args, **kwargs)
```

```
    def get_prep_value(self, value):
```

Preparing value for saving in DB

```
        if value is None:
```

```
            return value
```

Filtering only the digits to be saved in DB

```
        return ''.join(filter(str.isdigit, value))
```

```
class Employee(models.Model):
```

```
    ...
```

A default value can be set

```
    phone_number = PhoneNumberField(default='111-111-111')
```

Using the custom phone field

Saved value will be '11111111'



# Problem: Veterinarian Availability

- In the "**Veterinarian**" model implement a new field called "**availability**" with a custom model field type called "**BooleanChoiceField**"
  - It should **behave like a Boolean field** but has **custom choices** and a **default value**
- A full description of the problem can be found in the Lab document [here](#)

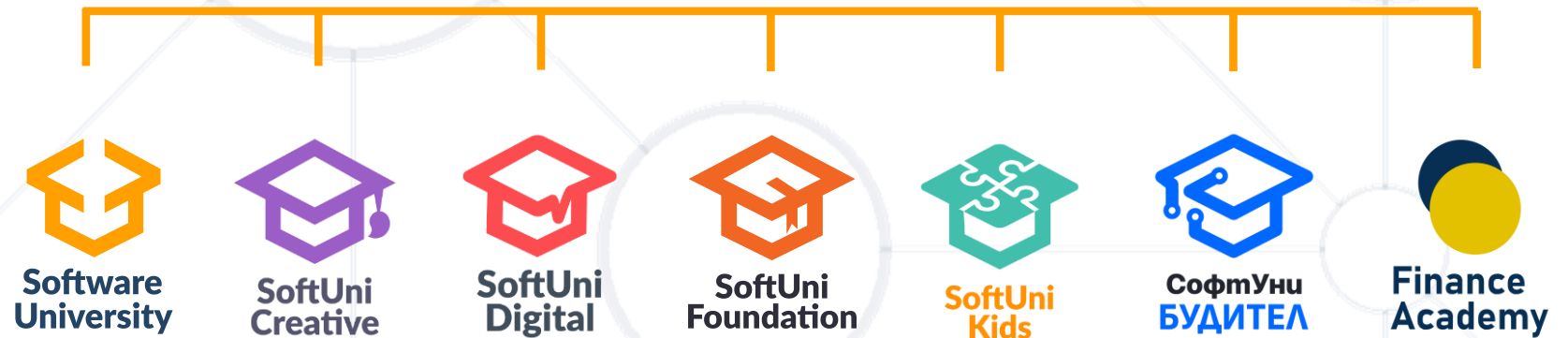
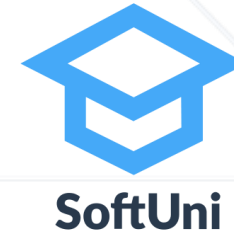
# Solution: Veterinarian Availability

```
class BooleanChoiceField(models.BooleanField):  
    def __init__(self, *args, **kwargs):  
        kwargs['choices'] = ((True, 'Available'),  
                             (False, 'Not Available'))  
        kwargs['default'] = True  
        super().__init__(*args, **kwargs)  
  
class Veterinarian(Employee):  
    ...  
    availability = BooleanChoiceField()
```

- Model **inheritance**
  - **Multi-table** Inheritance
  - **Abstract Base Classes**
  - **Proxy Models**
- Model **Methods**
  - **Built-in** Methods, **Custom** Methods
- **Custom** Fields
  - Custom Field **Built-in** Methods



# Questions?



# SoftUni Diamond Partners



**SUPER  
HOSTING  
.BG**



**VIVACOM**



**INDEAVR**  
Serving the high achievers



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [softuni.org](http://softuni.org)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)



- This course (slides, examples, demos, exercises, homework, documents, videos, and other assets) is **copyrighted content**
- Unauthorized copy, reproduction, or use is illegal
- © SoftUni – <https://softuni.org>
- © Software University – <https://softuni.bg>

