1. Model Inheritance

1.1. Multi-table inheritance: both parent and child models generate DB tables

```
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_colour = 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,
      )
```

1.2. Abstract Base Classes: the abstract model(parent) does not generate a DB table.

Acts as templates for other models to reuse common fields and behaviour

Using the inner class Meta

```
class ZooKeeper(Employee):
      ZOOKEEPER_CHOICES = (
            ('Mammals', 'Mammals'),
            ('Birds', 'Birds'),
            ('Reptiles', 'Reptiles'),
            ('Others', 'Others'),
      )
      speciality = models.CharField(
            max_length=10,
            choices=ZOOKEEPER_CHOICES,
      )
      managed_animals = models.ManyToManyField(
            to=Animal,
      )


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

1.3. Proxy Models: the proxy model(child) does not generate DB table
- The proxy model allows you to create a new model that behaves like an existing model with some customizations added
- The proxy model uses the same DB table as the original model
- Useful when adding extra methods, managers, or custom behaviour to existing model without modifying the original model

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

2. Model Methods
2.1. Built-in Methods
save() - called when saving an instance to the DB
clean() - used for data validation before saving
get_absolute_url()
__str__

```
                  def clean(self):
                        super().clean()
                        if self.speciality not in self.Specialities:
                              raise ValidationError("Speciality must be a valid choice")

                     def clean(self):
                        super().clean()

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

2.2. Custom Methods

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

      def __extra_info(self):
            extra_info = ''
            if hasattr(self, 'mammal'):
                  extra_info = f'Its fur color is {self.mammal.fur_colour}'
            elif hasattr(self, 'bird'):
                  extra_info = f'Its wingspan is {self.bird.wing_span} cm.'
            elif hasattr(self, 'reptile'):
                  extra_info = f'Its scale type is {self.reptile.scale_type}.'
            return extra_info

      def display_info(self):
            return f"Meet {self.name}! It's {self.species} and it's born {self.birth_date}. It makes a noise like '{self.sound}'!{self.__extra_info()} "

      def is_endangered(self):
            return True if self.species in ['Cross River Gorilla', 'Orangutan', 'Green Turtle'] else False
```

***

```
      @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
```

3. Custom Fields

- Custom Field Built-in Methods

from_db_value() - converts the field's value as retreived from the DB into its Python representation

to_python() - converts the field's value from the serialized format(usually as str)into its Python representation

get_repr_value() - prepares the field's value before saving to the DB

validate() - performs custom validation on the field's value

deconstruct() - used when serializing the field to store its constructor argument as tuple, allowing Django to recreate the field when migrating or serializing models