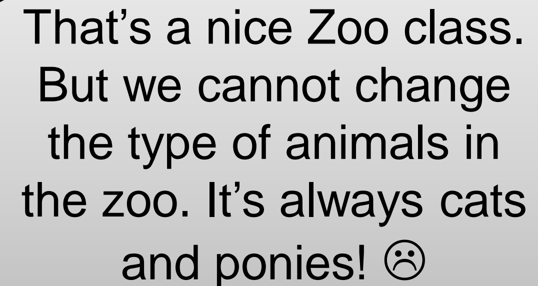
The Factory Method Design Pattern

A zoo with small and big animals

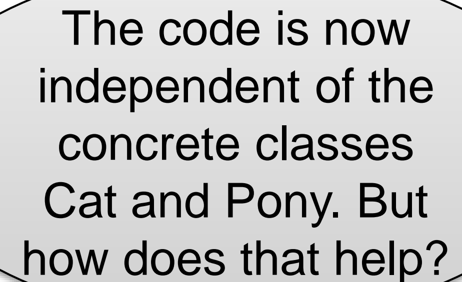
```
public class Zoo {
    private ArrayList<Animal> animals=new ArrayList<>();

    public void fillZoo(int n) {
        Random randomNumberGenerator=new Random();
        for(int i=0;i<n;i++) {
            boolean randomB=randomNumberGenerator.nextBoolean();
            if(randomB)
                animals.add(new Cat());
            else
                animals.add(new Pony());
        }
    }

    public int getYearlyCosts() {
        int sum=0;
        for(Animal animal : animals) {
            sum += animal.getYearlyCosts();
        }
        return sum;
    }
}
```



Using factory methods to create objects

- In this code, we have moved the creation of the objects into *factory methods* (=methods that create objects)

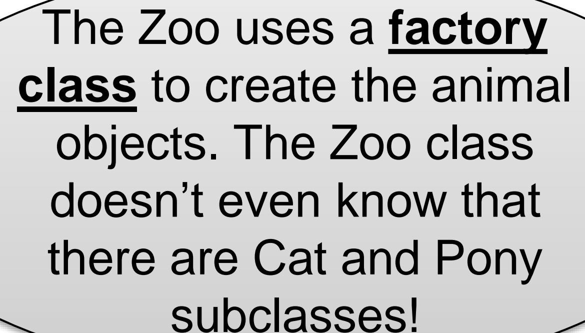
```
public class Zoo {
    private Animal createSmallAnimal() {
        return new Cat();
    }

    private Animal createBigAnimal() {
        return new Pony();
    }

    public void fillZoo(int n) {
        Random randomNumberGenerator=new Random();
        for(int i=0;i<n;i++) {
            boolean randomB=randomNumberGenerator.nextBoolean();
            if(randomB)
                animals.add(createSmallAnimal());
            else
                animals.add(createBigAnimal());
        }
    }
```



Using a factory class

- We can no move the object creation out of the Zoo class:

```
public class AnimalFactory {
    public Animal createSmallAnimal() {
        return new Cat();
    }
    public Animal createBigAnimal() {
        return new Pony();
    }
}

public class Zoo {
    private ArrayList<Animal> animals=new ArrayList<>();

    public void fillZoo(int n) {
        AnimalFactory af=new AnimalFactory();
        Random randomNumberGenerator=new Random();
        for(int i=0; i<n; i++) {
            boolean randomB=randomNumberGenerator.nextBoolean();
            if(randomB)
                animals.add(af.createBigAnimal());
            else
                animals.add(af.createSmallAnimal());
        }
    }
    ...
```



Using an abstract factory class

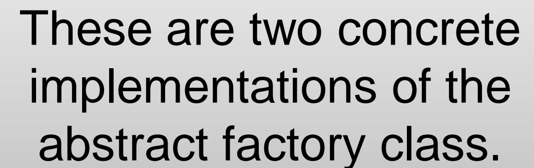
- Let's turn the factory into an abstract class. In this way, the code is independent of the concrete sub-classes of the Animal class:

```
public abstract class AbstractAnimalFactory {
    public abstract Animal createSmallAnimal();
    public abstract Animal createBigAnimal();
}

public class Zoo {
    private ArrayList<Animal> animals=new ArrayList<>();

    public void fillZoo(AbstractAnimalFactory af, int n) {
        Random randomNumberGenerator=new Random();
        for(int i=0; i<n; i++) {
            boolean randomB=randomNumberGenerator.nextBoolean();
            if(randomB)
                animals.add(af.createBigAnimal());
            else
                animals.add(af.createSmallAnimal());
        }
    }
    ...
```

The AbstractFactory only declares the factory methods. The implementation is somewhere else.

Sub-classing the abstract factory

```
public class PetFactory extends AbstractAnimalFactory {
    @Override
    public Animal createSmallAnimal() {
        return new Cat();
    }
    @Override
    public Animal createBigAnimal() {
        return new Pony();
    }
}
```




```
public class WildlifeFactory extends AbstractAnimalFactory {
    @Override
    public Animal createSmallAnimal() {
        return new Fox();
    }
    @Override
    public Animal createBigAnimal() {
        return new Elephant();
    }
}
```

We can now create zoos with different animals without changing the code of the Zoo class:

```
Zoo z1 = new Zoo(); z1.fillZoo(new PetFactory(),10);
Zoo z2 = new Zoo(); z2.fillZoo(new WildlifeFactory(),5);
```

Factory methods with parameters

- We could even have "intelligent" factories. Instead of three methods:

```
public abstract class AbstractAnimalFactory {
    public abstract Animal createSmallAnimal();
    public abstract Animal createBigAnimal();
    public abstract Animal createVeryBigAnimal();
}
```

we could have something more general:

```
public abstract class AbstractAnimalFactory {
    public abstract Animal createAnimal(String size, String color);
}
```

- And in the implementation in the PetFactory class:

```
@Override
public Animal createAnimal(String size, String color) {
    if(size.equals("very small") && color.equals("grey"))
        return new Mouse();
    else if(size.equals("small"))
        return new Cat();
    else if ...
}
```

Factory: Summary

- Factories are useful when you want to create objects without knowing (or without wanting to know) the exact class:

```
animals.add(af.createSmallAnimal());
```

- By using a factory, our Zoo class has become independent from the concrete sub-classes of the Animal class
  - The Zoo class does not even know which sub-classes exist
- Other people can now use our Zoo class with new animals that they defined themselves
  - You only have to make a new factory for your animals. You don’t need to change the Zoo class