Abstract Methods

> A method with the keyword `abstract` in its declaration is known as an **abstract method**.

Rules to be Followed

- In contrast to a concrete/normal Java method an **abstract method** does not have a body/definition i.e. it only has a declaration or method signature inside an *abstract class or an interface* (more on these later).
- An **abstract method** can be declared inside an *abstract class* or an *interface* only.
- In other words, it can be said that to contain any **abstract method** in its implementation a class has to be declared as an *abstract class* because *non-abstract classes* **cannot** have abstract methods.
- An *abstract method* **cannot** be declared *private* as it has to be implemented in some other class.

Declaration

Now moving on to the *syntax* part, syntactically, the generalized declaration of an abstract method is as follows:

```
public abstract void methodName(parameter(s));
```

An abstract method's declaration has:

1. An *access identifier*
2. The keyword `abstract`
3. A `return` type
4. A name of the method
5. The parameter(s) to be passed
6. A semicolon(;) to end the declaration

At this point, one may raise a question about the definition or the body of an abstract method i.e. *"Where do we implement the body of an abstract method?"*

Well, the upcoming topics will address the above question.

Abstract Class

> An **abstract class** is a class which is declared using the keyword `abstract`.

Rules to be Followed

- An abstract class ***cannot*** be instantiated i.e. one cannot create an object of an *abstract class*.
- An *abstract class* can have the declaration of *abstract method(s)* (as an abstract method's body cannot be implemented in an abstract class) but it is not compulsory to have any.
- Non-abstract/normal methods can be implemented in an **abstract class**.
- To use an *abstract class* it needs to be **inherited from**.
- The class which *inherits* from the *abstract class* **must** implement all the *abstract methods* declared in the *parent abstract class*.
- An abstract class can have everything else as same as a normal Java class has i.e. constructor, `static` variables and methods.

Declaration

Talking about the syntax, the *declaration* of an `abstract` class in Java is as follows:

```
abstract class ClassName {

  // Implementation here

}
```

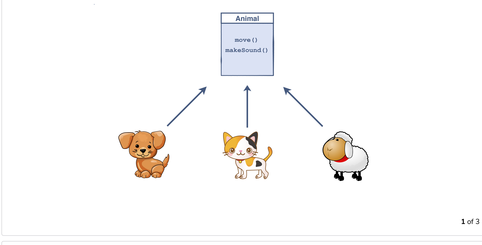
Implementation

Abstraction has already been discussed in the previous lesson. Abstract classes are used to achieve abstraction in Java.

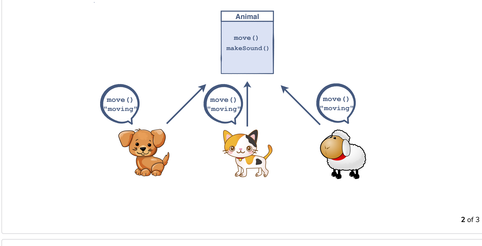
Consider modeling an Animal kingdom using Java having:

- A base `abstract` class named `Animal`
- A child class named `Dog`
- A child class named `Cat`
- A child class named `Sheep`

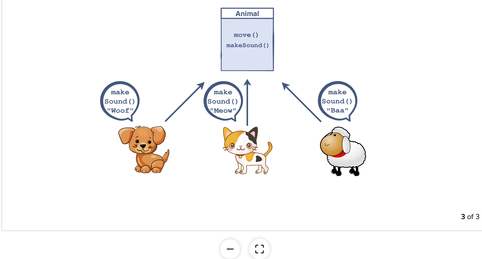
All of these animals make different sounds:



1 of 3



2 of 3



3 of 3

In the above example, one can observe that in the `Animal` class all the common traits of the animals should be implemented. All the other type-specific traits should be implemented inside the respective *child* classes. The `abstract` classes provide exactly the same functionality to the programmer. Let's implement this example below:

```
abstract class Animal {

  public abstract void makeSound();

  public void move() {
    System.out.println(getClass().getSimpleName()+" is moving");
    //getClass().getSimpleName() is an inbuilt functionality of Java
    //to get the class name from which the method is being called
  }

}

class Dog extends Animal {

  @Override
  public void makeSound() {
  System.out.println("Woof Woof...");
  }

}

class Cat extends Animal {

  @Override
  public void makeSound() {
  System.out.println("Meow Meow...");
  }

}
```

Run Save Reset

From the example above, we can observe just how beneficial an abstract class can be:

- All the animals can move and this is a common trait so the `move()` method is implemented in the `Animal` class and all the child classes can use this without any implementation inside themselves.
- All the animals make different sounds and because of that an `abstract` method is declared in the `Animal` class so that all the child classes **must** `@Override` this method in their own respective ways.