**Abstraction**

With abstraction, you can hide the internal workings of an object and only show the features the user needs to know about. Java provides two ways to implement abstraction: abstract classes and interfaces. With abstract classes, you can achieve partial abstraction, while interfaces make total (100%) abstraction possible.

**Abstract classes**

An abstract class is a superclass (parent class) that cannot be instantiated. You need to instantiate one of its child classes if you want to create a new object. Abstract classes can have both abstract and concrete methods. Abstract methods contain only the method signature, while concrete methods declare a method body as well. Abstract classes are defined with the abstract keyword.

In the example below, you can see an abstract class called Animal with two abstract and one concrete method.

abstract class Animal {

// abstract methods

abstract void move();

abstract void eat();

// concrete method

void label() {

System.out.println("Animal's data:");

}

}

Extend the Animal abstract class with two child classes: Bird and Fish. Both of them set up their own functionality for the move() and eat() abstract methods.

class Bird extends Animal {

void move() {

System.out.println("Moves by flying.");

}

void eat() {

System.out.println("Eats birdfood.");

}

}

class Fish extends Animal {

void move() {

System.out.println("Moves by swimming.");

}

void eat() {

System.out.println("Eats seafood.");

}

}

Now, test it with the TestBird and TestFish classes. Both call the one concrete (label()) and the two abstract (move() and eat()) methods.

class TestBird {

public static void main(String[] args) {

Animal myBird = new Bird();

myBird.label();

myBird.move();

myBird.eat();

}

}

class TestFish {

public static void main(String[] args) {

Animal myFish = new Fish();

myFish.label();

myFish.move();

myFish.eat();

}

}

In the console, the concrete method has been called from the Animal abstract class, while the two abstract methods have been called from Bird() and Fish(), respectively.

[Console output of TestBird]

Animal's data:

Moves by flying.

Eats birdfood.

[Console output of TestFish]

Animal's data:

Moves by swimming.

Eats seafood.

**Interfaces**

An interface is a 100% abstract class. It can have only static, final, and public fields and abstract methods. It’s frequently referred to as a blueprint of a class as well. Java interfaces allow us to implement multiple inheritance in our code, as a class can implement any number of interfaces. Classes can access an interface using the implements keyword.

In the example, define two interfaces, Animal and Bird. Animal has two abstract methods, while Bird has two static fields and an abstract method.

interface Animal {

public void eat();

public void sound();

}

interface Bird {

int numberOfLegs = 2;

String outerCovering = "feather";

public void fly();

}

The class Eagle implements both interfaces. It defines its own functionality for the three abstract methods. The eat() and sound() methods come from the Animal class, while fly() comes from Bird.

class Eagle implements Animal, Bird {

public void eat() {

System.out.println("Eats reptiles and amphibians.");

}

public void sound() {

System.out.println("Has a high-pitched whistling sound.");

}

public void fly() {

System.out.println("Flies up to 10,000 feet.");

}

}

In the TestEagle test class, instantiate a new Eagle object (called myEagle) and print out all the fields and methods to the console.

As static fields don’t belong to a specific object but to a whole class, you need to access them from the Bird interface instead of the myEagle object.

class TestEagle {

public static void main(String[] args) {

Eagle myEagle = new Eagle();

myEagle.eat();

myEagle.sound();

myEagle.fly();

System.out.println("Number of legs: " + Bird.numberOfLegs);

System.out.println("Outer covering: " + Bird.outerCovering);

}

}

The Java console returns all the information you wanted to access:

[Console output of TestEagle]

Eats reptiles and amphibians.

Has a high-pitched whistling sound.

Flies up to 10,000 feet.

Number of legs: 2

Outer covering: feather

**Encapsulation**

With encapsulation, you can protect the fields of a class. To do so, declare the fields as private and providing access to them with getter and setter methods.

The Animal class below is fully encapsulated. It has three private fields and each of them has its own set of getter and setter methods.

class Animal {

private String name;

private double averageWeight;

private int numberOfLegs;

// Getter methods

public String getName() {

return name;

}

public double getAverageWeight() {

return averageWeight;

}

public int getNumberOfLegs() {

return numberOfLegs;

}

// Setter methods

public void setName(String name) {

this.name = name;

}

public void setAverageWeight(double averageWeight) {

this.averageWeight = averageWeight;

}

public void setNumberOfLegs(int numberOfLegs) {

this.numberOfLegs = numberOfLegs;

}

}

The TestAnimal class first sets a value for each field with the setter methods, then prints out the values using the getter methods.

public class TestAnimal {

public static void main(String[] args) {

Animal myAnimal = new Animal();

myAnimal.setName("Eagle");

myAnimal.setAverageWeight(1.5);

myAnimal.setNumberOfLegs(2);

System.out.println("Name: " + myAnimal.getName());

System.out.println("Average weight: " + myAnimal.getAverageWeight() + "kg");

System.out.println("Number of legs: " + myAnimal.getNumberOfLegs());

}

}

As you can see below, the Java console returns properly all the values you set with the setter methods:

[Console output of TestAnimal]

Name: Eagle

Average weight: 1.5kg

Number of legs: 2

**Inheritance**

Inheritance allows us to extend a class with child classes that inherit the fields and methods of the parent class. It’s an excellent way to achieve code reusability. In Java, we need to use the extends keyword to create a child class.

In the example, the Eagle class extends the Bird parent class. It inherits all of its fields and methods, plus defines two extra fields that belong only to Eagle.

class Bird {

public String reproduction = "egg";

public String outerCovering = "feather";

public void flyUp() {

System.out.println("Flying up...");

}

public void flyDown() {

System.out.println("Flying down...");

}

}

class Eagle extends Bird {

public String name = "eagle";

public int lifespan = 15;

}

The TestEagle class instantiates a new Eagle object and prints out all the information (both the inherited fields and methods and the two extra fields defined in the Eagle class).

class TestEagle {

public static void main(String[] args) {

Eagle myEagle = new Eagle();

System.out.println("Name: " + myEagle.name); System.out.println("Reproduction: " + myEagle.reproduction);

System.out.println("Outer covering: " + myEagle.outerCovering);

System.out.println("Lifespan: " + myEagle.lifespan);

myEagle.flyUp();

myEagle.flyDown();

}

}

You can see the console output below:

[Console output of TestEagle]

Reproduction: another egg

Outer covering: feather

Lifespan: 15

Flying up...

Flying down...

**Polymorphism**

Polymorphism makes it possible to use the same entity in different forms. In Java, this means that you can declare several methods with the same name until they are different in certain characteristics. Java provides us with two ways to implement polymorphism: method overloading and method overriding.

**Static polymorphism**

Method overloading means that you can have several methods with the same name within a class. However, the number, names, or types of their parameters need to be different.

For example, the Bird() class below has three fly() methods. The first one doesn’t have any parameters, the second one has one parameter (height), and the third one has two parameters (name and height).

class Bird {

public void fly() {

System.out.println("The bird is flying.");

}

public void fly(int height) {

System.out.println("The bird is flying " + height + " feet high.");

}

public void fly(String name, int height) {

System.out.println("The " + name + " is flying " + height + " feet high.");

}

}

The test class instantiates a new Bird object and calls the fly() method three times. Firstly, without parameters, secondly, with one integer parameter for height, and thirdly, with two parameters for name and height.

class TestBird {

public static void main(String[] args) {

Bird myBird = new Bird();

myBird.fly();

myBird.fly(10000);

myBird.fly("eagle", 10000);

}

}

In the console, we can see that Java could have differentiated the three polymorphic fly() methods:

[Console output of TestBird]

The bird is flying.

The bird is flying 10000 feet high.

The eagle is flying 10000 feet high.

**Dynamic polymorphism**

By using the method overriding feature of Java, you can override the methods of a parent class from its child class.

The Bird class extends the Animal class in the example below. Both have an eat() method. By default, Bird inherits its parent’s eat() method. However, as it also defines its own eat() method, Java will override the original method and call eat() from the child class.

class Animal {

public void eat() {

System.out.println("This animal eats insects.");

}

}

class Bird extends Animal {

public void eat() {

System.out.println("This bird eats seeds.");

}

}

The TestBird class first instantiates a new Animal object and calls its eat() method. Then, it also creates a Bird object and calls the polymorphic eat() method again.

class TestBird {

public static void main(String[] args) {

Animal myAnimal = new Animal();

myAnimal.eat();

Bird myBird = new Bird();

myBird.eat();

}

}