Abstraction - Abstract Classes and Interfaces

In Java, **abstraction** is a mechanism for hiding the implementation details of a class and showing only necessary information to the users of that class. It is a fundamental concept in object-oriented programming that allows us to represent complex real-world entities in a simplified manner. It does so by focusing on the essential features of an object while hiding the inner workings of how they are what they are. In simple terms, abstraction lets us create a generalized representation of an object, focusing only on the important characteristics and behaviours.

Consider a television as an example of abstraction. When you use a television, you interact with its basic functionalities like turning it on/off, changing channels, and adjusting the volume. However, you don't need to know the intricate details of how the television works internally, such as the circuitry, components, or manufacturing process. The TV presents an abstraction where you can utilize its functionalities without being concerned about the underlying complexities.

Abstraction is one of the four fundamental OOP concepts along with **inheritance**, **polymorphism**, and **encapsulation**. Abstraction is implemented in Java using **abstract classes** and **interfaces**.

Even with everything we've learned so far, the above description might feel a bit jargon-y. To put it in succinct and simpler language, have a look at this YouTube video:

Open in new tab
https://youtu.be/L1-zCdrx8Lk

In sum: abstraction enables us to make our code more modular, and separate out concerns that are shared across objects and methods. It sets us up to be able to handle more cases more elegantly, and cuts down on needless repetitions.

Let's get into the implementation.

InfoWarningTip
This lesson makes use of a method called `@Override`. We will go over this method in depth soon. For now, just try to understand the basics of what it does as we walk through some examples. If you don't understand everything about it, that's OK.

#Implementation using Abstract Class

An abstract class is a class that **cannot be instantiated**, and it is typically used as a base class for other classes. It can have both abstract and non-abstract methods. **Abstract methods are methods that do not have a body, and they need to be overridden by the subclass.**

#Example 1:

Here is an example of an abstract class called Shape, which is the base class for different types of shapes:

CopyC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
public abstract class Shape {
//non-abstract method
public void moveTo(int x, int y) {
// Implementation goes here, which would make this a non-abstract method.
}
//abstract method
public abstract void draw();
}
```

In this example, the Shape class has one non-abstract method `moveTo(int x, int y)` and one abstract method `draw()`. The `moveTo(int x, int y)` method has a body, and

it can be called on an object of the `Shape` class, while the `draw()` method does not have a body and it needs to be overridden by the subclasses.

Here is an example of a subclass of Shape called Circle:

```
public class Circle extends Shape {
@Override
public void draw() {
//implementation to draw a circle
System.out.println("Draw a circle");
}
}
```

In the example above, the Circle class extends the Shape class and overrides the `draw()` method to provide its own implementation of how to draw a circle. As you can see, the `Shape` class provides a general abstraction of a shape, while the `Circle` class provides a specific implementation of that abstraction.

InfoWarningTip

It's important to note that you **can't** create an instance of an abstract class, like `Shape shape = new Shape()`, but you can create a reference variable of the abstract class type and assign it an instance of a subclass, like `Shape shape = new Circle()`.

#Example 2:

Here is an example of an abstract class called `Vehicle`, which is the base class for different types of vehicles:

```
public abstract class Vehicle {
private int numberOfWheels;
private String fuelType;
public Vehicle(int numberOfWheels, String fuelType) {
this.numberOfWheels = numberOfWheels;
this.fuelType = fuelType;
}
public abstract void move();
public abstract void stop();
}
```

In this example, the `Vehicle` class has two attributes `numberOfWheels` and `fuelType`, and two abstract methods `move()` and `stop()`. This class is representing a general abstraction of a vehicle, and it provides a way to set number of wheels and fuel type

of a vehicle. The `move()` and `stop()` methods do not have a body and they need to be overridden by the subclasses to provide the specific implementation of how the vehicle moves and stops.

Here is an example of a subclass of `Vehicle` called `Car`:

```
class Car extends Vehicle {
public Car(int numberOfWheels, String fuelType) {
super(numberOfWheels, fuelType);
}
@Override
public void move() {
System.out.println("The car is moving on the road.");
}
@Override
public void stop() {
System.out.println("The car has stopped at a traffic
signal.");
}
}
```

In this example, the `Car` class extends the `Vehicle` class and overrides the `move()` and `stop()` methods to provide its own implementation of how a car moves and stops. Of course, it's calling the constructor of the base class `Vehicle` using the `super` keyword to set the `numberOfWheels` and `fuelType` attributes in order to set those variables. We will talk more about `super` soon.

Here is an example of another subclass of `Vehicle` called `Bicycle`:

```
class Bicycle extends Vehicle {
public Bicycle(int numberOfWheels, String fuelType) {
super(numberOfWheels, fuelType);
}
@Override
public void move() {
System.out.println("The bicycle is moving on the road.");
}
@Override
public void stop() {
System.out.println("The bicycle has stopped at a traffic
signal.");
}
}
```

In this example, the `Bicycle` class also extends the `Vehicle` class and overrides the `move()` and `stop()` methods to provide its own implementation of how a bicycle moves and stops.

All of these examples have their shared attributes contained in the parent class, but override and assign values where necessary. That's abstraction at work.

#Implementation using Interface

In Java, abstraction is also implemented using interfaces. An interface is like a template that contains method declarations (not including the body of the methods) and constants. A class that implements an interface must provide the implementations (body of the method) for all of the interface's methods.

InfoWarningTip
One reason to use an interface instead of an abstract class is that a class can implement any number of interfaces that it needs to, while it can only extend one abstract class. It's also easier to implement a single interface across many different kinds of classes, such as a `speak` method which is needed for, say, a `robot`, a `smartPhone`, a `human`, and a `parrot` class. These classes are all unlikely to share many other class variables, but a `speak` method could suit all of them quite well, and would generally operate in a similar manner.

#Example 1:

Here is an example of an interface called `Shape`, which defines the basic behaviour of different types of shapes:

```
public interface Shape {
void draw();
double getArea();
}
```

In this example, the `Shape` interface has two abstract methods `draw()` and `getArea()`. These methods do not have a body, and they need to be implemented by any class that implements the `Shape` interface.

Here is an example of a class called `Circle` that implements the `Shape` interface:

```
class Circle implements Shape {
private double radius;
public Circle(double radius) {
this.radius = radius;
}
@Override
public void draw() {
//implementation to draw a circle
System.out.println("Draw a circle");
}
@Override
public double getArea() {
return Math.PI * radius * radius;
}
}
```

In this example, the `Circle` class implements the `Shape` interface and provides an implementation for the `draw()` and `getArea()` methods. As the `Circle` class implements the `Shape` interface, it must provide an implementation for all of the methods defined in the interface.

Here is another example of a class called `Rectangle` that also implements the `Shape` interface:

```
class Rectangle implements Shape {
private double width;
private double height;
public Rectangle(double width, double height) {
this.width = width;
this.height = height;
}
@Override
public void draw() {
//implementation to draw a rectangle
System.out.println("Draw a rectangle");
}
@Override
public double getArea() {
return width * height;
}
}
```

In this example, the `Rectangle` class also implements the `Shape` interface and provides an implementation for the `draw()` and `getArea()` methods.

As you can see, the `Shape` interface provides a **general abstraction** of a shape, while the `Circle` and `Rectangle` classes provide specific implementations of that abstraction.

As described in the video at the top of the page, all of this behaviour is designed to ensure that our inputs and outputs are predictable. Imagine if you instantiated a shape that didn't have a `getArea()` method, but that area was required for another calculation further down the line. It would cause an error at the very least, or could cause the program to fail at worst. By implementing the principles of abstraction and ensuring that all similar objects have the traits they need to have, we can ensure that our code continues to play nicely across a vast array of objects, methods, and dynamic behaviours. What happens in the in-between steps may be different from shape to shape, but the input and output remain consistent in the ways that we need them to.

#Java 8 interface

Java 8 introduced the concept of default methods and static methods in interfaces, which allows for providing a common behaviour for different classes that implement the interface and also gives the flexibility to add new methods without affecting the existing implementation of the interface. We'll be using it, but it's good to know that previous versions of Java may not support this feature.

- Default methods provide a default implementation for a method in an interface, which can be overridden by the classes that implement the interface.
- Static methods are associated with the interface itself and not with any class that implements the interface.

Here's an example of an interface called `Operations` that defines a default method for adding two numbers and a static method for generating a random number:

```
public interface Operations {
default int add(int a, int b) {
```

```
return a + b;
}

static int getRandomNumber() {
return (int) (Math.random() * 100);
}
}
```

Here's an example of a class called `Calculator` that implements the `Operations` interface:

```
class Calculator implements Operations {
// no need to implement add method, as we are using default
method
}
```

You can use the default method and the static method as follows:

```
public class Main {

public static void main(String[] args) {
Calculator calculator = new Calculator();
int result = calculator.add(5, 7);
System.out.println("Result of add: " + result);//Result of
add: 12
int randomNumber = Operations.getRandomNumber();
System.out.println("Random number: " + randomNumber);
}
}
```

In this example, the class `Calculator` doesn't need to implement the default methods of the interfaces, they can use the default implementations provided by the interfaces. The static methods can be called directly on the interface name, without needing an instance of the class.

#Reasons for using Abstraction in Java

1. **Code Reusability:** Abstraction in Java allows for creating a flexible and reusable codebase by creating a common interface for different classes to implement. This makes it easier to create new classes that conform to the same interface, allowing for code reuse.
2. **Polymorphism:** Abstraction in Java allows for polymorphism, which means that a class can be used through its interface, rather than its concrete implementation.

This allows for creating a more flexible and maintainable codebase by programming to an interface, rather than a concrete implementation.

3. **Loose Coupling:** Abstraction in Java promotes loose coupling between classes, which means that classes can be developed and modified independently of each other. This makes the codebase more maintainable and easier to modify.
4. **Separation of Concerns:** Abstraction in Java provides a clear separation between the implementation details of a class and its behaviour, making the code more readable, maintainable and easier to understand.
5. **Extensibility:** Abstraction in Java allows for creating a hierarchy of classes, interfaces, and packages that provide a general abstraction of a problem domain and specific implementations of that abstraction, which makes the code more extensible and flexible.
6. **Flexibility:** Abstraction in Java allows for creating objects that can change their behaviour at runtime, which helps in creating more flexible and robust code.
7. **Modularity:** Abstraction in Java promotes a modular design and make it possible to change the implementation of a class without affecting the other parts of the system.
8. **Simplicity:** Abstraction in Java simplifies the complex system, by hiding the unnecessary details and exposing the important details we need to know.

#Summary

- **Abstraction** is a key concept in object-oriented programming, and it is used to hide the implementation details of an object and only shows its behaviour. This ensures that its input and output remain consistent.
- In Java, abstraction is implemented using **abstract classes** and **interfaces**.
- An abstract class is a class that **cannot** be instantiated and can contain **both abstract and non-abstract methods**.
- An interface is a completely abstract class containing only **abstract methods and constants**.
- A class that implements an interface must provide an implementation for all of the interface's methods.
- Using interfaces and abstract classes allows for a more flexible and maintainable codebase by programming to an interface rather than a concrete implementation.

- Abstraction allows for creating a flexible, modular and reusable codebase by creating a common interface for different classes to implement.

#Knowledge Check

Fill In The Blank
Before Java 8, an interface could only contain abstract and .

Fill In The Blank
Abstraction in Java is accomplished through the use of classes and

In your own words, what's the difference between an abstract class and an interface? Feel free to use outside resources to help you understand better, if necessary.
abstract classes provide a mix of both method implementation and method contracts, while interfaces focus solely on method contracts.

#Extra Resources

Per the last question above, you may want to watch this video to better understand the difference between an abstract class and an interface:

Open in new tab
https://youtu.be/YfeEFovG3G4

Here's a good video about why to use abstract classes:

Open in new tab

https://youtu.be/HvPlEJ3LHgE

Back
Unsubmit
Next