**Abstraction in Java**

**Overview**

Abstraction in Java is a process of hiding the implementation details from the user and showing only the functionality to the user. It can be achieved by using abstract classes, methods, and interfaces. An abstract class is a class that cannot be instantiated on its own and is meant to be inherited by concrete classes. An abstract method is a method declared without an implementation. Interfaces, on the other hand, are collections of abstract methods.

**Introduction**

Abstraction is a key concept in OOP and in general as well. Think about real world objects, they are made by combining raw materials like electrons, protons, and atoms, which we don't see due to the abstraction that nature exposes to make the objects more understandable. Similarly, in computer science, abstraction is used to hide the complexities of hardware and machine code from the programmer.

This is achieved by using higher-level programming languages like Java, which are easier to use than low-level languages like assembly.

**Abstraction in Java**

Abstraction in Java refers to hiding the implementation details of a code and exposing only the necessary information to the user. It provides the ability to simplify complex systems by ignoring irrelevant details and reducing complexity. Java provides many in-built abstractions and few tools to create our own.

**Abstraction in Java can be achieved using the following tools it provides :**

* Abstract classes
* Interfaces

Let’s model the characters and objects from the very famous Angry Birds game to understand these concepts in detail.

**Example**

Basically, the gameplay involves shooting birds into pigs and structures in order to defeat them and retrieve their stolen eggs.

class Bird {

String name;

int size;

int strength;

protected void attack() {

*// empty method to be overridden by specific bird types*

}

public String getName(){

return name;

}

}

class Hal extends Bird {

@Override

public void attack() {

*// implementation of boomerang attack for Hal object*

}

}

class Chuck extends Bird {

@Override

public void attack() {

*// implementation of speed-up attack for Chuck object*

}

}

*// instantiate Chuck and Hal objects*

Bird chuck = new Chuck();

Bird hal = new Hal();

*// call attack method on objects*

chuck.attack(); *// Chuck speeds up*

hal.attack(); *// hal spins and boomerangs*

The code models the characters and objects in the game Angry Birds. It defines a Bird class with properties such as name, size, and strength, and a method attack() to be overridden by specific bird types. The classes Hal and Chuck extend the Bird class and provide the implementation for the attack() method. The reference type for the Chuck and Hal objects is Bird, but polymorphism ensures that the right attack() method is called at runtime.

However, there are some problems with the above model. What if I do the following :

Bird someBird = new Bird();

someBird.attack();

**Which character does this instantiated object define?** The answer is **none**.

The Bird object does not define a character. It can be attacked but the attack() method will not do anything because the implementation of the attack is in the sub-classes. Bird is just a model of real birds and cannot be instantiated. To stop creating Bird objects and make the model clearer, it can be made into an abstract class using the abstract keyword.

.

**What is an Abstract Class?**

In Java, an abstract class is a class that cannot be instantiated on its own and is meant to be subclassed. It provides a common base for subclasses to inherit method and field definitions, but allows for subclasses to provide their own implementations.

We can mark the Bird class abstract and the attack method as abstract to understand it better:

class abstract Bird {

String name;

int size;

int strength;

*// other properties go here*

protected abstract void attack();

public String getName(){

return name;

}

}

Now if we create object of Bird class:

Bird bird = new Bird();

The compiler would complain and the code would not compile.

**What is an Abstract Method?**

An abstract method is a method that is declared but not defined in a class. It acts as a placeholder for methods that must be implemented in subclasses.

It is used to enforce a certain level of consistency across related classes and ensure that subclasses implement specific behaviors. By declaring a method as abstract, you specify that the method must be implemented in a subclass, but you do not provide an implementation for the method in the superclass.

protected abstract void attack();

**Interface**

Interfaces are probably the most powerful tools to achieve abstraction in java.

Do not confuse it with the GUI interface or the dictionary or generic use of the work interface meaning API. It’s a java keyword called the **interface**.

Let’s see how interfaces solve the above-mentioned problems.

Interfaces are completely pure abstract classes. As a result of this, the classes implementing an interface are forced to give the implementation of all the methods and hence the JVM is not confused about which implementation to pick at runtime.

Since it’s just like an abstract class, it cannot be instantiated. And since it actually doesn’t provide an implementation and it cannot be instantiated it can’t have a constructor either. So let’s define the Hittable interface for the GameObjects that can be hit:

public interface Hittable {

int calculateDamage(int strength, double velocity, double angle);

}

**Why do classes have to implement interfaces?**

Duh, since all methods are abstract they don’t have any body. So by themselves they just define an API or method signatures however some class has to supply its implementation for the defined methods or API.

**Which classes should implement the interface?**

In our example any game object that is-a Hittable entity or class can implement this interface. Meaning, that any class that can take up the role that the interface is defining can implement it. To implement an interface use the keyword implementsfollowed by the interface name like so :

public abstract class Bird extends GraphicObject implements Hittable {

int calculateDamage(int strength, double velocity, double angle){

*// provide the logic to calculate damage for a bird*

}

}

Similarly the Block class can also implement the same interface :

public abstract class Block extends GraphicObject implements Hittable {

int calculateDamage(int strength, double velocity, double angle) {

*// provide the logic to calculate damage for a bird*

}

}

The compiler forces you to provide a body for the calculateDamage method as soon as the class implements the Hittable interface.

Let’s look at the Hittable interface definition a bit more closely.

public interface Hittable {

int calculateDamage(int strength, double velocity, double angle);

}

An interface is really just the exact definition of the API or method signatures that we require to accomplish certain tasks with the behaviour documented for each API or method.

Moreover, since interface is a pure abstract class one can program against the interface type as well and make full use of polymorphism like so :

Bird chuck = new Chuck();

Hittable hittableBlock = new WoodenBlock();

int chuckStrength = chuck.getStrength();

int velocity= chuck.getVelocity();

int angle = chuck.getAngle();

hittableBlock.calculateDamage(chuckStrength, velocity, angle);

We call the calculateDamage on the Hittable reference type which updates strength in the WoodenBlock by reducing it based on the input.

**Other Important Properties of an Interface**

All methods in an interface are by default **public** and you don’t have to explicitly write the public keyword either. This is because all classes have to implement all methods and if programming is done against the interface type the method call always has to be accessible at runtime.

Apart from abstract methods the interface body can **contain constants** as well. Although it’s **not a good practice** to put constants in your interface, that’s a story for another time.

One can also extend an interface with another interface. Like so :

public interface Collidable extends Hittable {

int displace(double incomingVelocity, double incomingAngle);

}

This works exactly like extending classes. The Collidable interface will get all the methods from the Hittable interface.

**Conclusion**

* Abstraction is a concept in object-oriented programming (OOP) that refers to the ability to focus on essential features of an object and ignoring the non-essential details.
* Abstraction in Java is achieved through the use of abstract classes and interfaces.
* An abstract class is a class that cannot be instantiated and serves as a blueprint for other classes. Abstract classes can have both abstract methods and concrete methods.
* An abstract method is a method declared without an implementation in an abstract class and must be overridden by subclass to provide a body for the method.
* We use interfaces to enforce behavior for multiple classes, while abstract classes only allow inheritance within a single hierarchy.
* An interface is a blueprint for classes that defines a set of methods that a class must implement.
* Interfaces cannot be instantiated on their own and must be implemented by a class. Also, classes can implement multiple interfaces.