By Manoj Debnath | November 17, 2022

The **Sealed** class is a recent introduction (JDK17) into the arsenal of Java. Due to this addition, another keyword was added to the set of keywords in the Java programming language. In fact, several new keywords were introduced to support the **Sealed** class: **sealed** and **non-sealed**.

These classes augment the classic concept of inheritance, where it is possible to make only a specific set of subclasses extend the parent class. So, in essence, **Final** classes are not allowed to inherit. **Non-final** classes, meanwhile, are inheritable by any subclass. The **Sealed** class works in-between, by specifying only some classes that are permitted to inherit. This programming tutorial introduces the uses and concepts behind the **Sealed** class in Java with code examples and use cases.

Read: Top Online Courses to Learn Java

What is Inheritance in Java?

As a refresher, inheritance is an object-oriented feature in which an inherited class extends the feature or functionality of its parent class. This enhances reusability. The subclass inherits the quality (*fields*, *methods*, and *nested classes*) of the parent class and can add its own qualities through polymorphism and function overloading. Unlike other object-oriented programming languages, which support multiple inheritance, Java strictly supports single inheritance. A subclass can extend only one parent class (single parent-child relationship between super and subclass). Here is an example of inheritance in Java:

```
class Bird {
    String greet;
    Bird(){
        greet="?!";
```

```
    }
    void saySomething(){
        System.out.println(greet);
    }
}

class Duck extends Bird{
    Duck(){
        greet="Quack, Quack";
    }
    void saySomething(){
        System.out.println(greet);
    }
}
```

Note, however, there is no such restriction when developers use *interfaces*. Java allows a class to implement multiple interfaces, as demonstrated in this example code:

```
class Bird {
    String greet;
    Bird(){
        greet="?!";
    }
    void saySomething(){
        System.out.println(greet);
    }
}

interface CanFly{
    default void shoo() {
        System.out.println("fly...");
    }
}

interface CannotFly{
    default void shoo() {
        System.out.println("run...");
    }
}

class Ostrich extends Bird implements CannotFly{
    Ostrich(){
        greet="boom";
    }
    void saySomething(){
        System.out.println(greet);
        shoo();
```

```
    }
}
>/pre>
```

Read: Java Tools to Increase Productivity

What is the Final Class in Java?

If programmers want to restrict a class from inheritance or make it absolutely uninheritable, we can simply begin the class definition with the **final** keyword. The purpose of this keyword is to prevent the class from being subclassed. The class, therefore, becomes unmodifiable and unextendible. Here is some example code showing how to use the **final** keyword in Java:

```
final class A {
    void func(){
        System.out.println("final class");
    }
}

class B extends A {} // this is not allowed
```

What is the Abstract Class in Java?

If developers want to make sure that no object can be created without extending the class, we can declare a class with the keyword **abstract**. Although an abstract can have all the features of a regular class, the use of the **abstract** keyword makes it special. To create an object of this class, programmers need to extend it with a **non-abstract** class, and only then are we allowed to create an instance of it. In this sense, the interface actually behaves like a pure **Abstract** class in Java. Here is an example:

```
abstract class Shape{
    void show(){
        System.out.print("Shape");
    }
}

class Box extends Shape{}

//...

Shape s = new Shape(); // error! Cannot create object
Shape s = new Box(); // OK
```

What is the Sealed Class in Java?

As you can see, before the introduction of **Sealed** classes, inheritance was an all or nothing kind of thing in Java. There was no provision for middle of the road, meaning – what if we want to permit some of the classes to inherit with restrictions that other classes will not be able to inherit. This type of restrictive, or selective, inheritance is possible with the **Sealed** class. It actually consists of two extreme classes: the **Final** class, which prevents inheritance entirely, and the **Abstract** class, which forces inheritance. The **Sealed** class enables developers to precisely specify which subclasses are permitted to extend the super class.

As there are **Sealed** classes, so, too, are there *Sealed interfaces*. Both give greater control over inheritance. This is particularly useful in designing class libraries.

Programmers can declare a **Sealed** class with the keyword **sealed**. Then we provide the *class name* and use the **permit** clause to specify allowable subclasses. Note that both the keywords, **sealed** and **permits**, are context-sensitive and have a special meaning in relation to class or interface declaration; they have no meaning outside this in Java.

A **Sealed** class in Java is declared as follows:

```
public sealed class A permits B, C {
        //...
}
```

In this code example, the class **A** is inheritable – or permitted to be inherited – by class **B** and **C**; no other class can inherit it.

```
public final class B extends A { }
public final class C extends A { }
public final class D extends A { } // Error! D cannot extend A
```

Note that the permitted class is declared **final**. This means that the permitted subclasses are not further inheritable. However, we can use other clauses, such as **non-sealed** or **sealed** with the subclasses, apart from the **final** keyword. In other words, a subclass of a sealed class must be declared as **final**, **sealed**, or **non-sealed**.

Now, if we want class **B** to be further extendable by class **D**, we may declare **D** as follows:

```
public sealed class B extends A permits D {}
public final class D extends B {} // now it's OK.
```

Now, suppose we want class **A** to be extended by **B** and **C** and we also want class **D** to extend class **B**, but we do not want class **D** to be declared as **final**, **non-sealed**, or **sealed**, then we may design the class as follows:

```
Public sealed class A permits B,C{}
public non-sealed class B extends A { }

public class D extends A { } // OK
```

Some key requirements for **Sealed** classes include:

- The permitted subclass must be accessible by the **Sealed** class.
- A **Sealed** class and subclasses must be in the same named module, although they can be in different packages.
- In case of an unnamed module, **Sealed** classes and subclasses must be in the same package.

Read: Polymorphism in Java

Sealed Interfaces in Java

Sealed interfaces are declared in much the same way as **Sealed** classes. Here, the **permit** clause is used to specify which class or interface is allowed to implement or extend the **Sealed** interface. For example, we can declare a **Sealed** interface as follows (here, the **Sealed** interface permits class **A** and **B** for implementation):

```
public sealed interface SealedInterface permits A, B {}
```

A class that implements a **Sealed** interface must be declared as **final**, **sealed**, or **non-sealed**:

```
public non-sealed class A implements SealedInterface{}
public final class B implements SealedInterface{}
```

If an interface extends a **Sealed** interface, it must be declared as either **Sealed** or **non-sealed**. For example:

```
public non-sealed interface AI extends SealedInterface{}
```

Final Thoughts on Sealed Classes in Java

The introduction of **Sealed** classes and interfaces in Java imbibes some flexibility into the feature of inheritance in the Java programming language. Although they may be used under special circumstances, **Sealed** classes may be in the class design of API libraries. The point is that the provision of flexibility is there in the language; a programmer can use them as per their program's requirements. The best thing about this feature is that it brings some form of flexibility in the use of inheritance, where one has to dwindle between **Final** classes that completely restrict inheritance, or go for **Abstract** classes where full inheritance is a must.

Read more Java programming and software development tips.