Java 15 Features

**Date 21/04/2023**

# Overview of Java 15 (Sept 15, 2020)

1. JEP 360 - Sealed Classes
2. JEP 384 - Records
3. JEP 339 - Edwards-Curve Digital Signature Algorithm (EdDSA)
4. JEP 371 - Hidden Classes

# JEP 360 - Sealed Classes

Java 15 introduces a sealed classes as preview feature which provides a fine-grained control over inheritance. Currently, Java provides no fine-grained control over the inheritance. Access modifiers such as public, protected, private, as well as the default package-private, provide very coarse-grained control. To end that, the goal of sealed classes is to allow individual classes to declare which types may be used as sub-types. This also applies to interfaces and determining which types can implement them. "Sealed Classes" introduced as a preview feature by **JDK Enhancement Proposal 360** offer developers of a Java class or interface the possibility to restrict which other classes and interfaces can extend or implement them.

A sealed class structure is defined as follows,

* The sealed keyword marks a sealed class.
* After the keyword permits, you list the allowed subclasses.
* A subclass of a sealed class must be either sealed, final, or non-sealed. In the first case, you must again define the allowed subclasses with permits. The last case means that the subclass is again open to inheritance - just like any regular class.
* Sealed classes help in creating a finite and determinable hierarchy of classes in inheritance.

Here is an example,

public sealed class Shape permits Circle, Square, Rectangle, WeirdShape { ... }

public final class Circle extends Shape { ... }

public final class Square extends Shape { ... }

public sealed class Rectangle extends Shape permits TransparentRectangle, FilledRectangle { ... }

public final class TransparentRectangle extends Rectangle { ... }

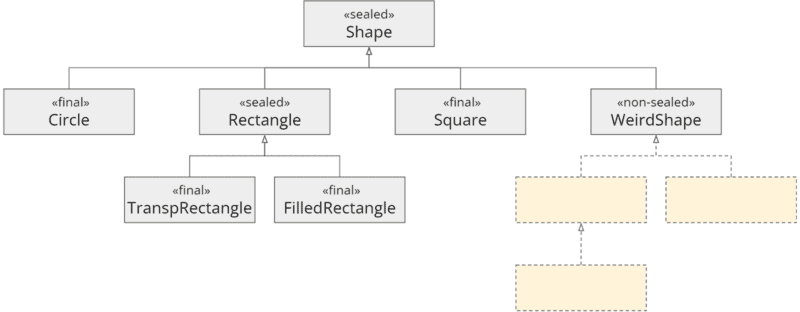
public final class FilledRectangle extends Rectangle { ... }

public non-sealed class WeirdShape extends Shape { ... }

**Here’s how each of the modifiers impact inheritance,**

* A permitted subclass that’s declared final cannot be extended further.
* A permitted subclass that’s declared sealed can be extended further but only by classes that are permitted by the subclass.
* A permitted subclass may be declared non-sealed can be extended further by any class. The superclass cannot restrict the subclasses further down this class hierarchy.

The following class diagram shows the class hierarchy implemented in the sample code. The orange rectangles demonstrate that the hierarchy is extensible only under WeirdShape.



public class \_01\_SealedClass\_Demo1 {

public static void main(String[] args) {

Person manager1 = new Manager(101, "John");

System.***out***.println(*getId*(manager1));

System.***out***.println(manager1.getName());

Manager manager2 = new Manager(101, "John");

System.***out***.println(manager2.getManagerId());

System.***out***.println(manager2.getName());

Person employee1 = new Employee(201, "Alex");

System.***out***.println(*getId*(employee1));

System.***out***.println(employee1.getName());

Employee employee2 = new Employee(201, "Alex");

System.***out***.println(employee2.getEmployeeId());

System.***out***.println(employee2.getEmployeeName());

}

private static int getId(Person person) {

if (person instanceof Employee emp) {

return emp.getEmployeeId();

} else if (person instanceof Manager mgr) {

return mgr.getManagerId();

}

return -1;

}

}

abstract sealed class Person permits Employee, Manager {

String name;

String getName() {

return name;

}

}

final class Employee extends Person {

String name;

int id;

Employee(int id, String name) {

this.id = id;

this.name = name;

}

public int getEmployeeId() {

return id;

}

public String getEmployeeName() {

return name;

}

}

non-sealed class Manager extends Person {

int id;

Manager(int id, String name) {

this.id = id;

this.name = name;

}

public int getManagerId() {

return id;

}

}

# JEP 384 - Records

Records were introduced as a preview feature in Java 14, in an effort to reduce boilerplate code when writing POJO based data carrier classes. Java 14 introduces a new class type record as preview feature to facilitate creation of immutable data objects. Java 15 enhances record type further. It is still a preview feature.

Now, with Java 15, Records get their second preview. While there aren’t any major changes(just some minor additions), still there are a few major clarifications and restrictions.

Some fine-tuning has been done for **Java 15 by JDK Enhancement Proposal 384**,

* The implicitly declared fields corresponding to the record components of a record class are final and you can no longer change a record's fields using reflection.
* You can combine records with sealed interfaces.
* You can define "local records" within methods.
* Prior to Java 15, one could declare native methods in records(though it wasn’t a good idea). Now the JEP explicitly prohibits against declaring native methods in records.

## Changing Fields of a Record via Reflection

In Java 14, it was possible to change the final fields of a record via reflection. The following example shows how you could change the x value of the Point record p shown above,

**Field X = Point.class.getDeclaredField("x");**

**X.setAccessible(true);**

**X.set(p, newX);**

In Java 15, this attempt results in an **IllegalAccessException**.

public class \_01\_Records\_Enhancement\_Demo1 {

record Circle(int x, int y) {

}

public static void main(String[] args) {

Circle c = new Circle(3, 5);

int x = c.x();

int y = c.y();

System.***out***.println(x);

System.***out***.println(y);

int newX = 5;

Field X = null;

try {

X = Circle.class.getDeclaredField("x");

X.setAccessible(true);

X.set(c, newX);

} catch (IllegalArgumentException | IllegalAccessException e) {

e.printStackTrace();

} catch (NoSuchFieldException | SecurityException e) {

e.printStackTrace();

}

}

}

## Records and Sealed Interfaces

Records can implement sealed interfaces, which were also added as a preview feature in Java 15. Accordingly, sealed interfaces may also list records in their "permits" list.

public class \_02\_Records\_Enhancement\_Demo2 {

public static void main(String[] args) {

Swift swift = new Swift(600000, "Baleno");

System.***out***.println(swift.getDetail());

Tata tata = new Tata(1000000);

System.***out***.println(tata.getDetail());

}

}

sealed interface Car permits Swift, Tata {

String getDetail();

}

record Swift(int price, String model) implements Car {

*@Override*

public String getDetail() {

return "The Swift model " + model + " car price is " + price;

}

}

record Tata(int price) implements Car {

*@Override*

public String getDetail() {

return "The Tata car price is " + price;

}

}

## Local Records

Records may now also be defined within methods and are then only visible within this method. These local records are helpful when you want to store intermediate results with multiple related variables.

Records can also be defined within methods to store intermediate values. Unlike local classes, a local record is implicitly static. This means they cannot access variables and instance members of the enclosing methods which is actually great since it prevents capturing of values by the record.

Local records are a great boon for Java developers who would earlier have to create helper records.

public class \_03\_Records\_Enhancement\_Demo3 {

public static void main(String[] args) {

Merchant alex = new Merchant("Alex");

Merchant john = new Merchant("John");

Merchant blake = new Merchant("Blake");

Merchant erica = new Merchant("Erica");

List<Merchant> merchantsList = List.*of*(alex, john, blake, erica);

List<Sale> salesList = List.*of*(new Sale(alex, LocalDate.*of*(2023, *Month*.***APRIL***, 17), 11034.20),

new Sale(john, LocalDate.*of*(2023, *Month*.***APRIL***, 18), 8234.22),

new Sale(blake, LocalDate.*of*(2023, *Month*.***APRIL***, 19), 11000.47),

new Sale(erica, LocalDate.*of*(2023, *Month*.***APRIL***, 20), 19654.20));

\_03\_Records\_Enhancement\_Demo3 demo = new \_03\_Records\_Enhancement\_Demo3();

List<Merchant> topMerchant = demo.findTopMerchant(salesList, merchantsList, 2023, *Month*.***APRIL***);

topMerchant.stream().forEach(m -> System.***out***.println(m.name()));

}

record Merchant(String name) {

}

record Sale(Merchant merchant, LocalDate date, double value) {

}

List<Merchant> findTopMerchant(List<Sale> sales, List<Merchant> merchants, int year, *Month* month) {

record MerchantSales(Merchant merchant, double sales) {

}

return merchants.stream()

.map(merchant -> new MerchantSales(merchant, this.computeSales(sales, merchant, year, month)))

.sorted((m1, m2) -> Double.*compare*(m2.sales(), m1.sales())).map(MerchantSales::merchant)

.collect(Collectors.*toList*());

}

double computeSales(List<Sale> sales, Merchant merchant, int year, *Month* month) {

return sales.stream().filter(s -> s.merchant().name().equals(merchant.name()) && s.date().getYear() == year

&& s.date().getMonth() == month).mapToDouble(s -> s.value()).sum();

}

}

# JEP 339 - Edwards Curve Digital Signature Algorithm (EdDSA)

EdDSA is a modern signature method that is faster than previous signature methods, such as DSA and ECDSA while maintaining the same security strength. EdDSA is supported by many crypto libraries such as OpenSSL and BoringSSL. Many users already use EdDSA certificates. Edwards-Curve Digital Signature Algorithm, EdDSA is an advanced elliptic curve scheme and is better than existing signature schemes in the JDK. It has improved security and performance as compared to other signature schemes. It is supported by popular crypto libraries like OpenSSL, BoringSSL etc. EdDSA will only be implemented in java 15 only in the SunEC provider.

public class \_01\_EdDSA\_Algorithm\_Demo1 {

public static void main(String[] args) throws NoSuchAlgorithmException, SignatureException, InvalidKeyException {

String message = "Happy Learning!";

KeyPairGenerator kpgenerator = KeyPairGenerator.*getInstance*("Ed25519");

KeyPair kpair = kpgenerator.generateKeyPair();

Signature sign = Signature.*getInstance*("Ed25519");

sign.initSign(kpair.getPrivate());

sign.update(message.getBytes(StandardCharsets.***UTF\_8***));

byte[] signature = sign.sign();

System.***out***.println("Signature => " + Base64.*getEncoder*().encodeToString(signature));

}

}

# JEP 371 - Hidden Classes

Application frameworks such as Java EE and Spring generate numerous classes dynamically at runtime. In particular, they create proxies for application classes to add features such as access control, caching, transaction management, and JPA lazy loading.

The existing ClassLoader.defineClass() and Lookup.defineClass() APIs generate bytecode indistinguishable from the bytecode that results from compiling static application classes.

Thus, the dynamically generated classes are discoverable by all other classes in the class loader hierarchy and exist as long as the class loader in which they were generated.

That is typically undesirable. On the one hand, those classes are usually considered framework-specific implementation details that should remain hidden from the rest of the application. On the other hand, they are often only needed for a particular time, unnecessarily increasing the application's memory requirements after they have been used.

In Java 15, JDK Enhancement Proposal 371 has introduced "hidden classes" into the JDK.

Hidden classes are defined via the MethodHandles.Lookup.defineHiddenClass() method and cannot be used by other classes - neither directly nor via reflection.

# References

<https://www.happycoders.eu/java/java-15-features/>

<https://www.tutorialspoint.com/java15/index.htm>

<https://www.baeldung.com/java-15-new>

<https://www.digitalocean.com/community/tutorials/java-15-features>