Records in Java 14

Passing immutable data between objects is one of the most common, yet tedious tasks in many Java applications. Traditionally, this required the creation of a class filled with boilerplate code—constructors, getters, toString(), and the crucial hashCode() and equals() methods—just to encapsulate simple data. This approach was verbose and error-prone, often leading to trivial mistakes and muddled intentions in otherwise straightforward code.

With the release of Java 14, records offer a streamlined, built-in solution to this problem. By eliminating the need for manual boilerplate, including the automatic generation of hashCode() and equals(), records allow developers to focus more on business logic and less on repetitive coding tasks. All while ensuring immutability and consistent behavior in collections.

What is a Record?

A record is a special kind of class in Java that is designed to serve as a transparent, immutable data carrier. When you define a record, Java automatically generates:

- A constructor
- Accessor methods (getters)
- equals(), hashCode(), and toString() methods

These methods are all based on the fields declared in the record.

```
public record Point(int x, int y) {}
```

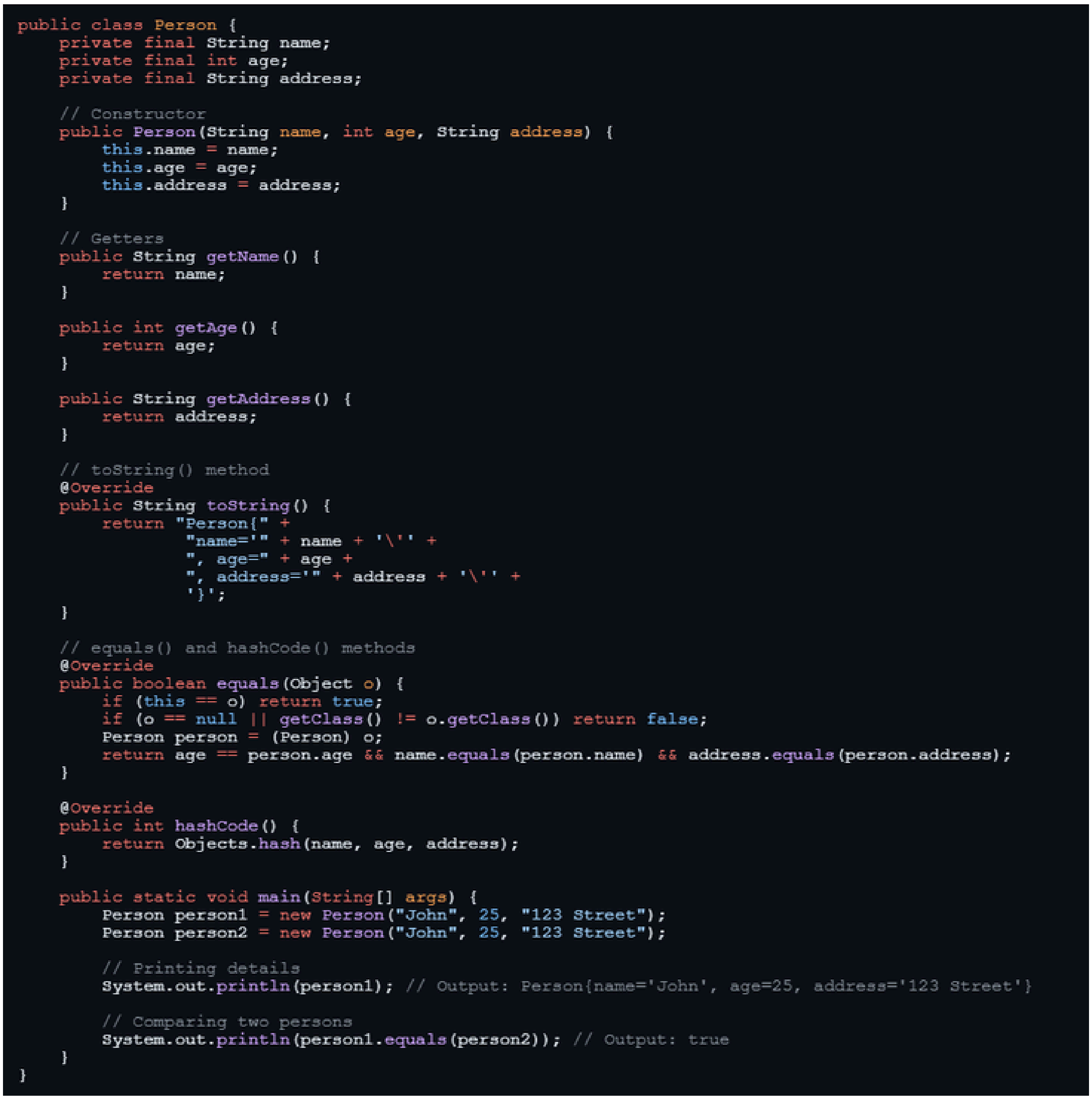
This Point record has two fields, x and y. With this single line of code, Java will automatically generate:

A constructor Point(int x, int y)

Getter methods x() and y()

toString(), equals(), and hashCode() based on x and y.

Now Lets Implement a program without using records, create Person class with name, age, address attributes and compare the objects and display the details of the person



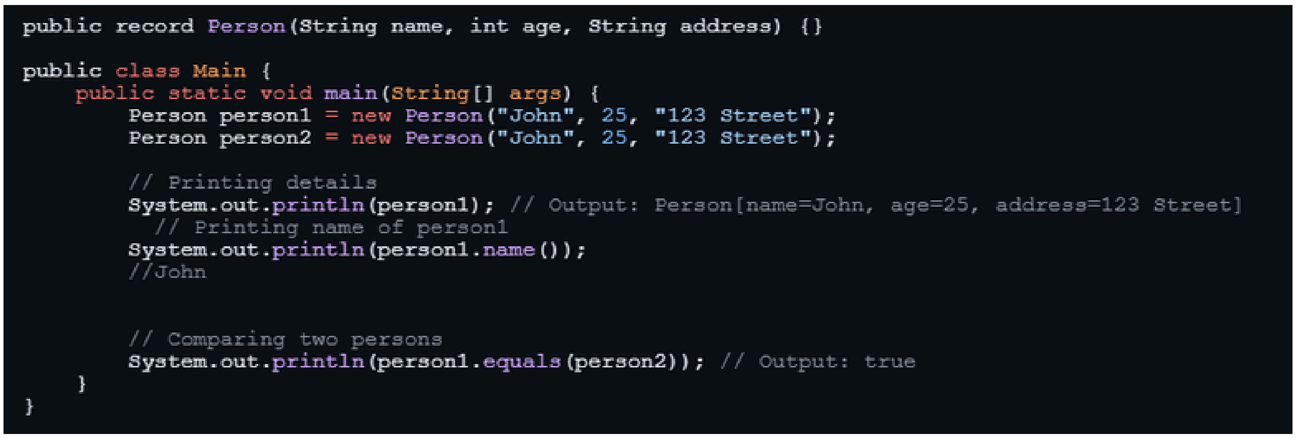
```java
public class Person {
    private final String name;
    private final int age;
    private final String address;

    // Constructor
    public Person(String name, int age, String address) {
        this.name = name;
        this.age = age;
        this.address = address;
    }

    // Getters
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public String getAddress() {
        return address;
    }

    // toString() method
    @Override
    public String toString() {
        return "Person{" +
               "name='" + name + '\'' +
               ", age=" + age +
               ", address='" + address + '\'' +
               '}';
    }

    // equals() and hashCode() methods
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return age == person.age && name.equals(person.name) && address.equals(person.address);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age, address);
    }

    public static void main(String[] args) {
        Person person1 = new Person("John", 25, "123 Street");
        Person person2 = new Person("John", 25, "123 Street");

        // Printing details
        System.out.println(person1); // Output: Person{name='John', age=25, address='123 Street'}

        // Comparing two persons
        System.out.println(person1.equals(person2)); // Output: true
    }
}
```

While this accomplishes our goal, there are two problems with it:

- There's a lot of boilerplate code
- We obscure the purpose of our class: to represent a person with a name and address

In the first case, we have to repeat the same tedious process for each data class, monotonously creating a new field for each piece of data; creating equals, hashCode, and toString methods; and creating a constructor that accepts each field. While IDEs can automatically generate many of these classes, they fail to automatically update our classes when we add a new field.

we can replace our repetitious data classes with records. Records are immutable data classes that require only the type and name of fields. The equals, hashCode, and toString methods, as well as the private, final fields and public constructor, are generated by the Java compiler.



```java
public record Person(String name, int age, String address) {}

public class Main {
    public static void main(String[] args) {
        Person person1 = new Person("John", 25, "123 Street");
        Person person2 = new Person("John", 25, "123 Street");

        // Printing details
        System.out.println(person1); // Output: Person[name=John, age=25, address=123 Street]
          // Printing name of person1
        System.out.println(person1.name());
        //John


        // Comparing two persons
        System.out.println(person1.equals(person2)); // Output: true
    }
}
```

Explanation of Java Records

- **Reduced Boilerplate**: The Person record automatically provides the constructor, getters, toString(), equals(), and hashCode() without any manual implementation.
- **Immutability**: Records are inherently immutable, so once a record object is created, its fields cannot be modified.
- **Compact and Readable**: The code is concise, easy to read, and reduces the chance of errors, especially when fields are added or modified.

Real-World Use Cases

- DTOs (Data Transfer Objects): Records are an excellent choice for DTOs, which are used to transfer data between layers in an application.
- Immutable Data Objects: Use records for configurations, settings, or other immutable data models where the data should not change after it is initialized.
- Response Payloads: Records can be used to model REST API responses in a compact and immutable way.

-------------Happy Coding-----------------