9. Using Records

Exam Objectives

1. Create and use records

9.1 Purpose of Records

As they say, necessity is the mother of invention. You saw how the wide spread usage of the type safe enum pattern led to the inclusion of enums in Java. It is the same story with records. Most Java applications deal with transporting data objects back and forth between various application layers. Applications routinely use **relational databases** (RDBMS) to store data in tables. This data needs to be converted into regular Java objects so that other parts of the application can use them in an OOP way.

For example, you may have student data in a STUDENT table. But in Java application, you may have a Student class with appropriate fields for its attributes such as ID, Name, Address, and so on. You may either use plain **JDBC** to fetch the rows and then convert the raw data into collection of Student objects yourself or let an **Object Relational Mapping** (ORM) tool such as **Hibernate** do it for you, but in either case, there is a need to have a Student class with fields corresponding to the table columns. Most of the time, there is no business logic in the Student class, which is also why it is also called **Plain Old Java Object** (POJO). It exists just to keep the data of a Student **"record"** (in database parlance, each row of a table is also called a record). Moreover, the Student objects are not even meant to be **mutated**(meaning, once a Student object is created, there is no need to change its values). New Student objects are created as and when required and transported between layers. These objects are also known as **Data Transfer Objects** (DTO) and may have dozens or (even hundreds in some cases) of fields! Can you imagine coding and maintaining a class with a hundred fields? Think about its constructor and all the accessor methods! It is a nightmare and that's why there are tools to automate creation of such classes.

Well, Java 17 has made it a little bit easier to code such data heavy classes through something called **Record classes**. So, basically, Java records is a new way, which requires less typing, of writing Java classes that are meant to capture data elements, have no business logic, and whose objects are meant to be immutable. Let's see how. Here is the code for a `Student` class that I would normally to write prior to Java 17:

```
public class Student {
    private int id;
    private String name;
    private String address;

    public Student(int id, String name, String address){
        this.id = id;
        this.name = name;
        this.address = address;
    }
    public int getId() {
        return id;
    }
    public String getName() {
        return name;
    }
```

```
    public String getAddress() {
        return address;
    }
}
```

Observe that all I have in this class is a set of fields, a constructor that initializes those fields, and getter methods for those fields. There are no methods to set or modify the fields of a Student object because I don't want anyone to modify Student objects once they are created. I want Student objects to be **immutable**. The following is how I would do it in Java 16:

```
public record Student(int id, String name, String address){ };
```

That's it! In fact, with this one line, I actually get a few more things than what I get with the previous code. I will talk more about that later but the point to note here is that records are meant to provide a short and simple yet OO compatible way to capture state and make it immutable. Everything else about records follows from this requirement.

There is no difference in how you use a record because a record is, after all, nothing more than an immutable class. For example, here is how I would instantiate a Student and access its field:

```
Student s  = new Student(1, "Adam Smith", "123 Main Street");
String name = s.name(); //observe that it is not getName() but just name()
```

9.2 Creating and using a record

9.2.1 Record basics

Besides the access modifier and the record keyword, primarily, a record definition has two parts - **record header** and **record body**. Record header is where you specify the fields that you want in your record and record body is where you may optionally specify constructors and methods of that record.

So, in the Student record defined earlier, `(int id, String name, String address)` is the record header and `{ }` is the record body.

The parameters specified in a record header are called **record components**. Thus, id, name, and address, are the components of the Student record. Record components completely describe the state that a record is meant to keep and based on this state information, the compiler derives and generates its "API" using a set of rules. The API of a record is nothing but a constructor, accessor methods, the **equals** method, the **toString** method, and the **hashCode** method. Furthermore, the compiler also uses this information while compiling code that uses **record patterns**.

Record fields

As per the Java language specification, for each record component, a record class has a field with the same name as the record component and the same type as the declared type of the record component. This field, which is declared implicitly, is known as a **component field**. Furthermore, for each record component, a record class has a method with the same name as the record component and an empty formal parameter list. This method, which is declared explicitly or implicitly, is known as an

accessor method. The fundamental concept behind the rules that govern how a record is defined and interpreted is that the state of a record is completely defined by the information specified by its components. Therefore, the following restrictions are placed on a record:

1. A record is not allowed to define any **instance field** explicitly. It may have static fields though because static fields do not constitute the state of an object.

2. A record implicitly **extends java.lang.Record** class but it is not allowed to have an **extends clause**. The reason is that if it were allowed to extend another class, it would inherit the instance fields of that class and its state would depend on that class as well.

3. A record is allowed to **implement interfaces** and may inherit default methods from that interface.

4. A record is implicitly **final**, which means no other class can extend from it. For the same reason, you cannot make a record **abstract**, **sealed**, or **unsealed**.

9.2.2 Record constructors

Recall that a normal class may have one or more constructors and if it has no explicit constructor, the compiler provides a "default" constructor, which doesn't really do anything other than invoking `super()`. However, records are immutable and since there is no way to set the values of its component fields once the constructor has finished execution, the rules for the constructors of records are different from the rules for the constructors of regular classes.

Canonical Constructor

Every record must have exactly one **canonical constructor**. If you don't provide it in your record explicitly, the compiler will provide it. Unlike the default constructor of a regular class, the canonical constructor **must** initialize all of the component fields of the record using the parameters passed to it. For example, the canonical constructor of the `Student` record looks like this:

```
public Student(int id, String name, String address) {
   this.id = id;
   this.name = name;
   this.address = address;
}
```

It has no throws clause and its access modifier is the same as that of the record itself. In a way, the canonical constructor is the default constructor for records.

It is often required to check the input parameters for their validity or to alter them (or in technical jargon, to "normalize" them) in some way before copying them to the record fields. It is possible to do that by defining the canonical constructor explicitly. For example:

```
public record Student(int id, String name, String address){
   public Student(int id, String name, String address) {
      if(id < 1) throw new RuntimeException("Invalid ID"); //check validity of input
```

```
        this.id = id;
        if(name.trim().length() == 0) name = "DUMMY"; //fix input data
        this.name = name;
        this.address = address;
    }
}
```

The above is how you would write a constructor for a regular class and it is a perfectly valid way for records as well. But to make it easier, a new shorter way of writing the canonical constructor, called the "compact constructor" has been introduced. It is written as follows:

```
public Student { //No formal parameter list
    if(id < 1) throw new RuntimeException("Invalid ID");
    if(name.length < 0) name = "";
    //what about address?
}
```

There are a two points worth noting about the above compact constructor:

1. There is no formal parameter list in the above definition. As you can see from the regular constructor, the list of formal parameters is not any different from the one present in the record header. So, the compiler just copies it from there instead of making the developer repeat the same information in the constructor.

2. We are not assigning the value of the variables to the record fields. There is no `this.id = id;` or other similar statements anywhere! Actually, you are not allowed to set or modify the component fields explicitly at all anywhere in the compact constructor. Component fields are are final and they are set only once at the end of the compact constructor by code inserted by the compiler. The compiler automatically inserts the assignment statements of the form `this.<fieldname> = <fieldname> ;` automatically at the end of the compact constructor.

There are many rules associated with canonical constructors. You can read about them in the JLS if you have time. I am summarizing the ones that are relevant for the exam below:

	Long form Canonical Constructor	Compact Canonical Constructor
Example	`public record Student(int id, String name){ public Student(int id, String name){ this.id = id; this.name = name; } }`	`public record Student(int id, String name){ public Student{ } }`
Number of constructors	**Exactly one** - Either long-form or compact. Not both.	Same
Access modifier	Cannot be more restrictive than the record.	Same
Formal parameter list	**Required** - The order, the names, and the types must match with the record header.	No formal parameter list. Local variables of the same names and types as the ones present in the header are automatically defined in the constructor. They are initialized with the arguments passed to the constructor.
Initialization of component fields	Must be done explicitly	Cannot be done explicitly. Compiler assigns the values of the automatically defined local variables to component fields at the end of the constructor.
Call to another constructor	Not allowed. Explicit calls to `this(...)` or `super(...)` are not allowed.	Same
Exceptions	Throws clause is not allowed. Body may throw unchecked exceptions.	Same
Modify input arguments	Allowed	Allowed

Non-canonical constructors

A record is allowed to have any number of non-canonical constructors but the first line of such constructors must be a call to another constructor of that record using the `this(...)` syntax (also called an **"alternate constructor"**). Note that this requirement is different from regular classes where the first line of a constructor must be a call to either another constructor using the `this(...)` syntax or a super class's constructor using the `super(...)`syntax.

An implication of this requirement is that by the time the control reaches the second line of non-canonical constructor's body, the component fields of this record are already set. Since the component fields are final, you can't modify them anymore. Here is an example:

```
public record Student(int id, String name, String address){
   public Student(int whatever, String name) throws Exception{
```

```
        this(checkId(whatever), name.trim(), "DUMMY"); //first line must be an explicit
    call to another constructor of this record

        //component fields of this record are already set at this point
        //this.name = "Dummy"; //can't do this now

        name = "Dummy"; //valid but name refers to the constructor parameter and
    updating it has no impact on the component field
    }
    private static int checkId(int value){
      if(value<0) throw new IllegalArgumentException("Bad Id");
      return value;
    }
}
```

Observe that I haven't provided a canonical constructor in the above record and so, the compiler will provide it automatically, which is why the call to `this(...)` is valid. The above code highlights the following points regarding a non-canonical constructor:

1. Names of the formal parameters are not important. I have named the formal constructor parameter as `whatever` to show that the name of the parameter in a non-canonical constructor doesn't have to be the same as the name of the component field.
2. Throws clause is allowed.
3. The first line must be call to another constructor of the same record.
4. Although we cannot modify component fields after the call to the alternate constructor, it is possible to validate input arguments or to supply normalized values of input arguments to the alternate constructor if the modification is done on the first line itself as illustrated by the calls to the `checkId` method and `name.trim()`.

9.2.3 Record methods

Accessor methods

The compiler provides accessor methods for all of the component fields automatically. The name and the return type of an accessor method are the same as the name and type of the field. So, for example, you can access the id and the name fields of the Student record defined above as follows:

```
Student s = new Student(1, "Bob", "123 Main");
int id = s.id();
String name = s.name();
```

You are allowed to define an accessor method explicitly. It must be public and must not have a throws clause. Of course, since a record is immutable, there is no method to modify a component field.

It is important to note that the accessor methods do not follow the **JavaBeans** naming convention for accessor methods. As per the JavaBeans naming conventions, the accessor method for `name` should have been **getName** but here, it is just **name**.

Utility methods

The compiler automatically generates a few instance methods in addition to the accessor methods. These are:

1. `public final boolean equals(Object o)` : It returns true if and only if all of the component fields of this record are equal to the same fields of the other record as per their equals method.

2. `public final int hashCode()` : It returns a hash code value derived from the hash code values of each component field of the record.

3. `public final String toString()` : It returns a string derived from the name of the record class and the names and string representations of every component field of the record. For example, `s.toString()` will return `Student[id=1, name=Bob, address=123 Main]`. This method is useful when you want to print the contents of a record in a log file.

You are free to provide your own implementations of these methods explicitly in the record.

Other methods

You are allowed to define other instance and static methods in a record just like you do in normal classes.

9.2.4 Records and Generics

I haven't discussed generics yet, but just be aware that records can be generic. So, for example, you can define a Student record as `record Student<T> (int id, String name, T data);` and use it as `Student<Double> s = new Student<> (1, "Bob", 1.0);` You cannot, however, define an component field accessor method with generics though. So, `public T data(){ return data; }` will not compile. You must rely on the accessor method generated by the compiler in this case. So, `Double d = s.data();` will work as expected.

9.2.5 Records and serialization

I will discuss serialization in detail in the I/O chapter but for now, remember that records are **serializable**. The mechanism, though, is slightly different from regular classes. The serialized form of a record object is a sequence of values derived from the record component fields. During deserialization, individual record components are deserialized and then the canonical constructor of the record is invoked. The `serialVersionUID` of a record class, which is `0L` by default unless explicitly declared, is not checked during deserialization.

Note that deserialization of a regular class does not require the constructor of the class to be invoked and the serialVersionUID of the serialized object must match with that of the class. The process of serialization and deserialzation of records cannot be customized by providing writeObject, readObject, readObjectNoData, writeExternal, or readExternal methods.

9.2.6 Quiz

What is wrong with the following record definitions? **A.**

```
record JournalEntry(int crdAcctId, int dbtAcctId, double amt, String desc){
  private int dbid;
  public int getDbid(){ return dbid; }
  public void setDbid(int dbid){ this.dbid = dbid; }
}
```

B.

```
record Address(String value){ }
record Person(Address a){ }
record Student extends Person(String grade){
}
```

C.

```
interface Marker{ void mark(); }
public abstract record Data() implements Marker{ }
```

D.

```
record Account(int id){
   public static int classID;
   public Account{
     this.id = id;
   }
}
```

E.

```
record Account(int id){
   public Account(int acctId){
     this.id = acctId;
   }
   public Account(){
     this(0);
   }
}
```

F.

```
record Account(int id, String name){
   public Account(int id, String name){
     this.id = id;
   }
   public Account(String name){
     this(0, name);
   }
}
```

G.

```
record Account(int id, String name){
   public Account{
   }
   public Account(int id, String name){
     this.id = id;
     this.name = null;
   }
}
```

A. You are not allowed to define an instance field in a record explicitly. All instance fields are defined implicitly in the record header. You may define static fields, instance methods, and static methods.

B. A record always implicitly extends `java.lang.Record`. It cannot have an explicit extends clause. It is ok to have another record as a component field. So, `Person` is fine.

C. A record is always implicitly **final**. Therefore, it cannot be **abstract**, **sealed**, or **unsealed**. It may implement interfaces. It is okay to not have any component field in a record.

D. You cannot modify instance members of a record in the compact canonical constructor explicitly. So, `this.id = id;` will not compile. The `classID` field is fine.

E. While using the long form canonical constructor, the names of the formal parameters must match with names used in the record header. So, `acctId` is invalid. The regular non-canonical constructor is fine.

F. While using the long form canonical constructor, every component field of the record must be set explicitly. Here, `name` is left uninitialized.

G. You may have either the long form canonical constructor or the compact canonical constructor, not both. The statement `this.name = null;` is fine. Although it makes no sense, but it does initialize the `name` field.

9.3 Exercise

1. Write the following class as a record.

```
enum ProductType{
    FMCG, APPLIANCE, PHARMA
}
class Product{
    int productId;
    String name;
    ProductType pt;
    //constructor accessor methods for all of the above fields
```

```
}
```

2. Create a constructor in the above record that takes only one parameter for `productId`.

3. Modify the record definition such as `productId` cannot be set to less than 1. Do it with the compact constructor and then with the regular constructor.

4. Create a `PriceWrapper` class as follows: `class PriceWrapper{ double price; //constructor and methods not shown }`. Add a field of type `PriceWrapper` in the above record. Provide an instance method in the record to update the `price`. Create two instances of the `Product` record with the same values but change the `price` in one instance. Check the equality of the two records to see whether they are still considered equal.

5. Override the `equals` method in `Product` to include price comparison while determining whether two `Product` records are equal or not.