Records

What are records?

- encapsulated classes, but without boilerplate code
- the encapsulation is secured
- constructor, getters, `toString()`, `equals()` and `hashCode()` are generated
- records cannot have explicit instance fields
- records can have static fields and methods
- records can have instance methods

```
// the old way of creating encapsulated class

public final class Student {

  // 1. declare private final fields
  private final String firstName;
  private final String lastName;
  private final int id;

  // 2. define the constructor
  public Student(String firstName, String lastName, int id) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.id = id;
  }
```

```
// 3. define getters
public int getId() {
  return id;
}
public String getFirstName() {
  return firstName;
}
public String getLastName() {
  return lastName;
}

// 4. override toString() method
@Override
public String toString() {
  return "Student{" +
    "firstName='" + firstName + '\'' +
    ", lastName='" + lastName + '\'' +
    ", id=" + id + '}';
}
```

```
  // 5. override equals() method
  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Student student = (Student) o;
    return id == student.id && Objects.equals(firstName, student.firstName) &&
       Objects.equals(lastName, student.lastName);
    }

  // 6. override hashCode() method
  @Override
  public int hashCode() {
  return Objects.hash(firstName, lastName, id);
  }
}
```

```
// all this can be replaced by only ONE line:

public record Student (String firstName, String lastName, int id);

var theStudent = new Student("Luka", "Popov", 1);
System.out.println(theStudent.firstName());
  => Luka
System.out.println(theStudent.lastName());
  => Popov
System.out.println(theStudent.id());
  => 1

// NOTE: the getter is not like getFirstName(), but firstName() !!
```

```
// toString() is nicely implemented
System.out.println(theStudent);
  => Student[firstName=Luka, lastName=Popov, id=1]



// equals() is implemented as we would expect
var anotherStudent = new Student("Luka", "Popov", 1);
System.out.println(theStudent == anotherStudent);
  => false
System.out.println(theStudent.equals(anotherStudent));
  => true
```

```
// we can override auto-generated constructor
// this is called "canonical constructor"


public record Student (String firstName, String lastName, int id) {
  public Student (String firstName, String lastName, int id) {
     if (id < 10 || id > 1_000_000) throw new IllegalArgumentException();
     this.firstName = firstName;
     this.lastName = lastName;
     this.id = id;
   }
}
```

```
// there is simpler way => "compact constructor"


public record Student (String firstName, String lastName, int id) {
  public Student {
     if (id < 10 || id > 1_000_000) throw new IllegalArgumentException();
  }
}


// notice the syntax: no ()
// instance fields don't need to be explicitly initialized
```

```
// compact constructor could contain any business logic, e.g.


public record Student (String firstName, String lastName, int id) {
  public Student {
    if (id < 10 || id > 1_000_000) throw new IllegalArgumentException();
    firstName = fistName.substring(0,1).toUpperCase
                + firstName.substring(1).toLowerCase;
    lastName = lastName.substring(0,1).toUpperCase
                + lastName.substring(1).toLowerCase;
  }
}
// this "normalizes" the name (luka -> Luka, lUKa -> Luka, etc.)
```