Java 21

Record Patterns

Record Patterns

- Review “*type patterns*”
- Review “*pattern matching*”
- Review records
- Record Patterns

Type Patterns

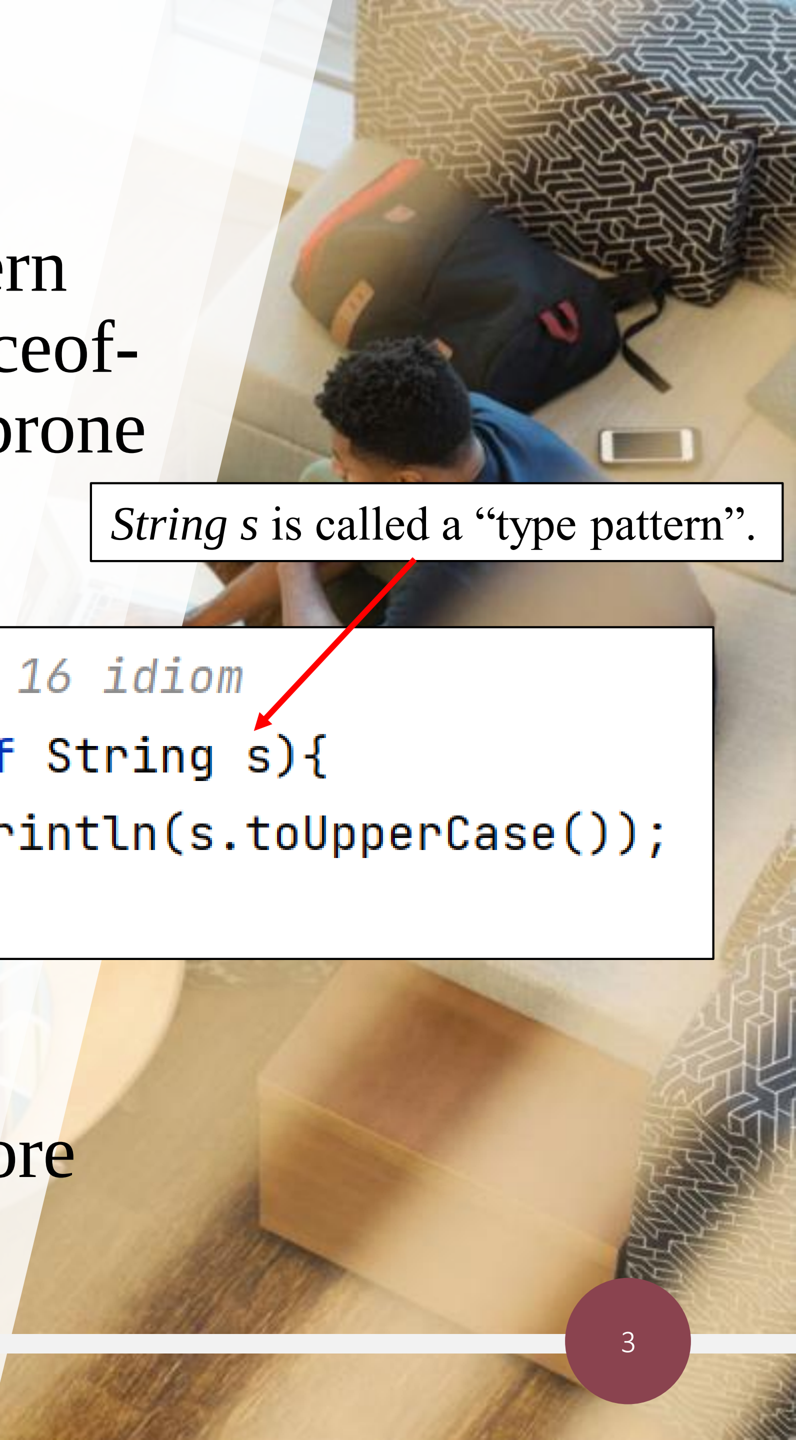
- In Java 16, *instanceof* was extended to take a type pattern and perform pattern matching. This simplified the instanceof-and-cast idiom, resulting in more concise and less error-prone code.

String s is called a “type pattern”.

```
// old pre-Java 16 instanceof-and-cast idiom
if(obj instanceof String){
    String s = (String)obj;
    System.out.println(s.toUpperCase());
}
```

```
// new post-Java 16 idiom
if(obj instanceof String s){
    System.out.println(s.toUpperCase());
}
```

- As there is no casting with *type patterns*, the style is more declarative.

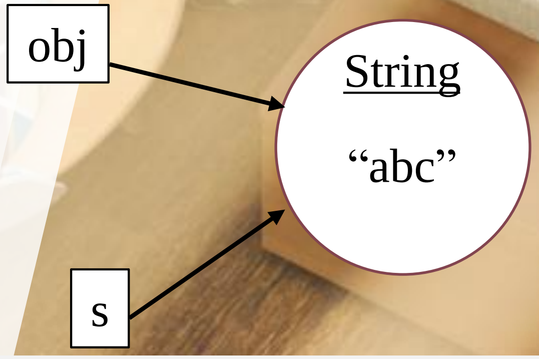


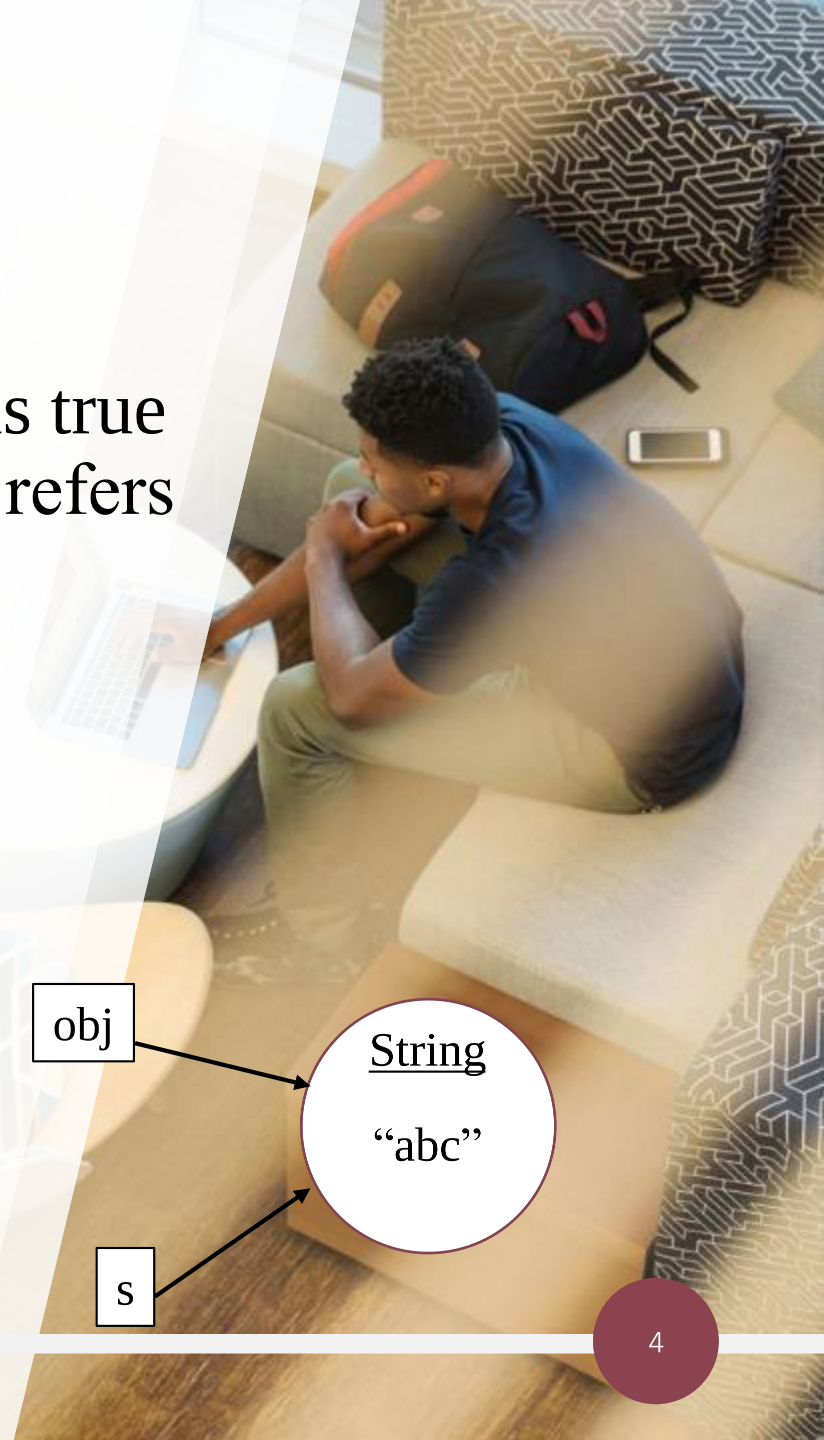
Pattern Matching

- *Pattern matching* is done at runtime.
- If the pattern matches, then the *instanceof* expression is true and the pattern variable '*s*' now refers to whatever '*obj*' refers to.

String s is called a "type pattern".

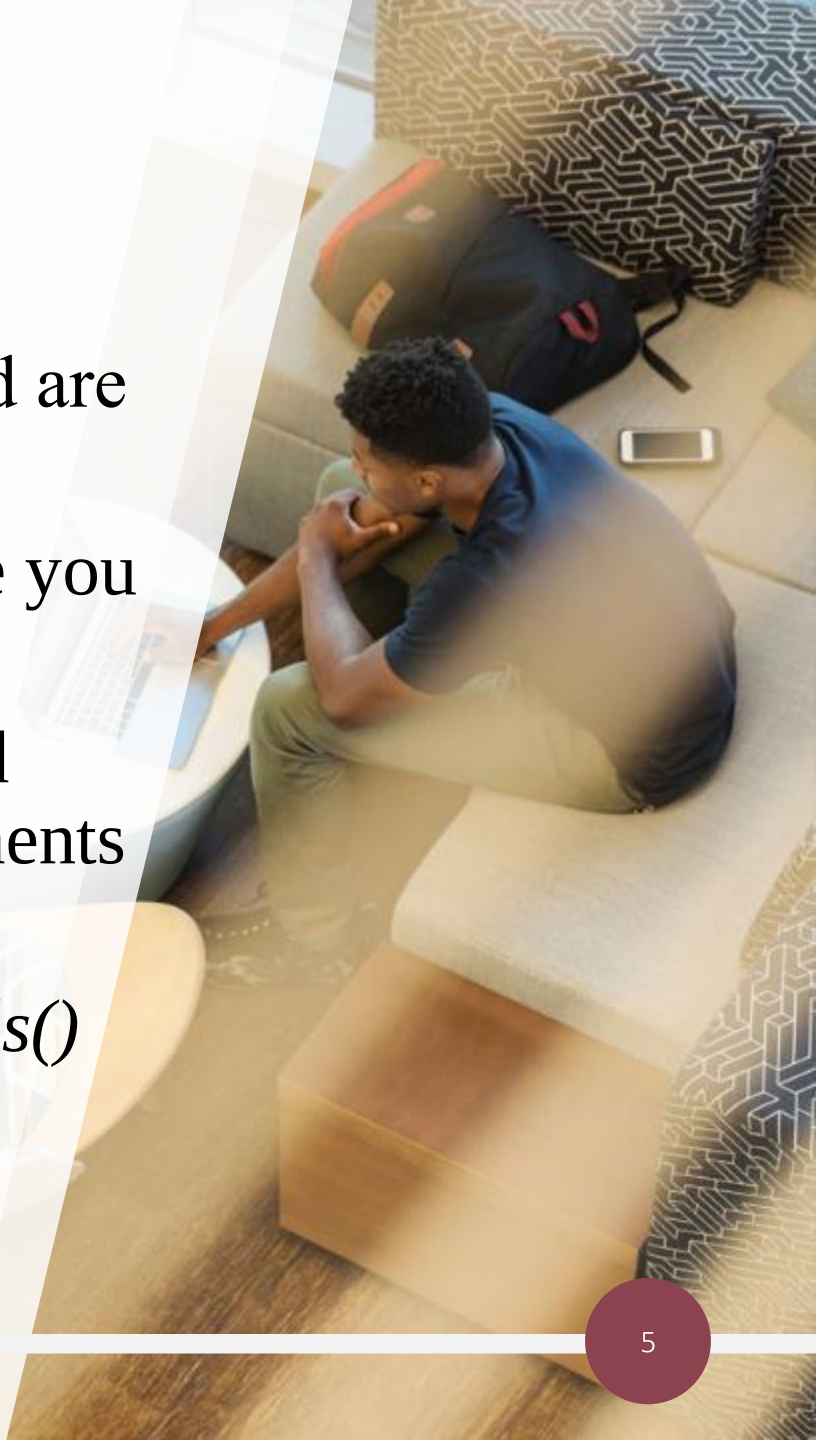
```java
// new post-Java 16 idiom
if(obj instanceof String s){
    System.out.println(s.toUpperCase());
}
```





Records

- Records are a special type of class that save us a lot of boilerplate code. They are considered “data carriers” and are immutable.
- Records are specified using a *record* declaration where you specify the “components” of the record.
- These components become *final* instance variables and accessor methods having the same names as the components are provided automatically.
- In addition, a (canonical) constructor, *toString()*, *equals()* and *hashCode()* methods are also generated.



Records

```
public record Dog(String name, Integer age) {}
```

```
public class Dog{
    private final String name;
    private final Integer age;

    public Dog(String name, Integer age) {
        this.name = name;
        this.age = age;
    }
    public String name() {
        return name;
    }
    public Integer age() {
        return age;
    }
    // also toString(), equals() and hashCode() generated
}
```

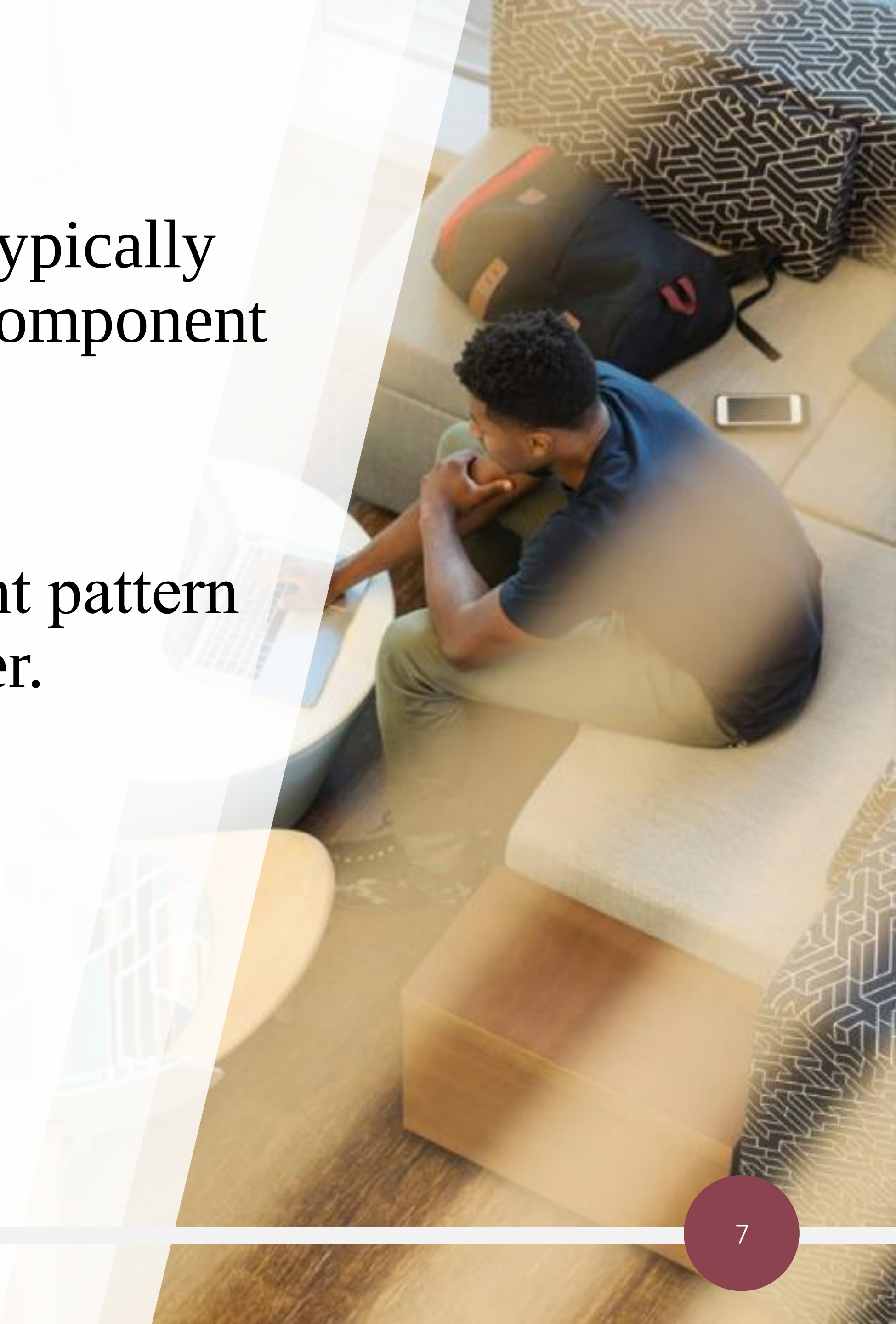
Record Patterns

- Code that receives an instance of a record class, typically extracts the data (components), using the built-in component accessor methods.

- A “record pattern” consists of a type, a component pattern list (which may be empty) and an optional identifier.

- A record pattern does two things for us:
    1. checks if an object passes the *instanceof* test.
    2. disaggregates the record instance into its components.

- Record patterns support nesting.



Record Patterns

```
public record Person(String name, Integer age) {}
```

```
if(obj instanceof Person p){ // type pattern
    // 'p' is only being used to invoke the accessor
    // methods name() and age().
    String name = p.name();
    int age     = p.age();
    System.out.println(name + ", "+ age);
}
// Person(String s, Integer nAge) is a "record pattern" which does 2 things:
//   1. Tests whether the object is of type Person (as usual)
//   2. Extracts the records components by invoking the component accessor
//      methods on our behalf.
if(obj instanceof Person(String s, Integer nAge)){ // record pattern
    System.out.println(s + "; "+ nAge);
}
```