Annotations

Annotations

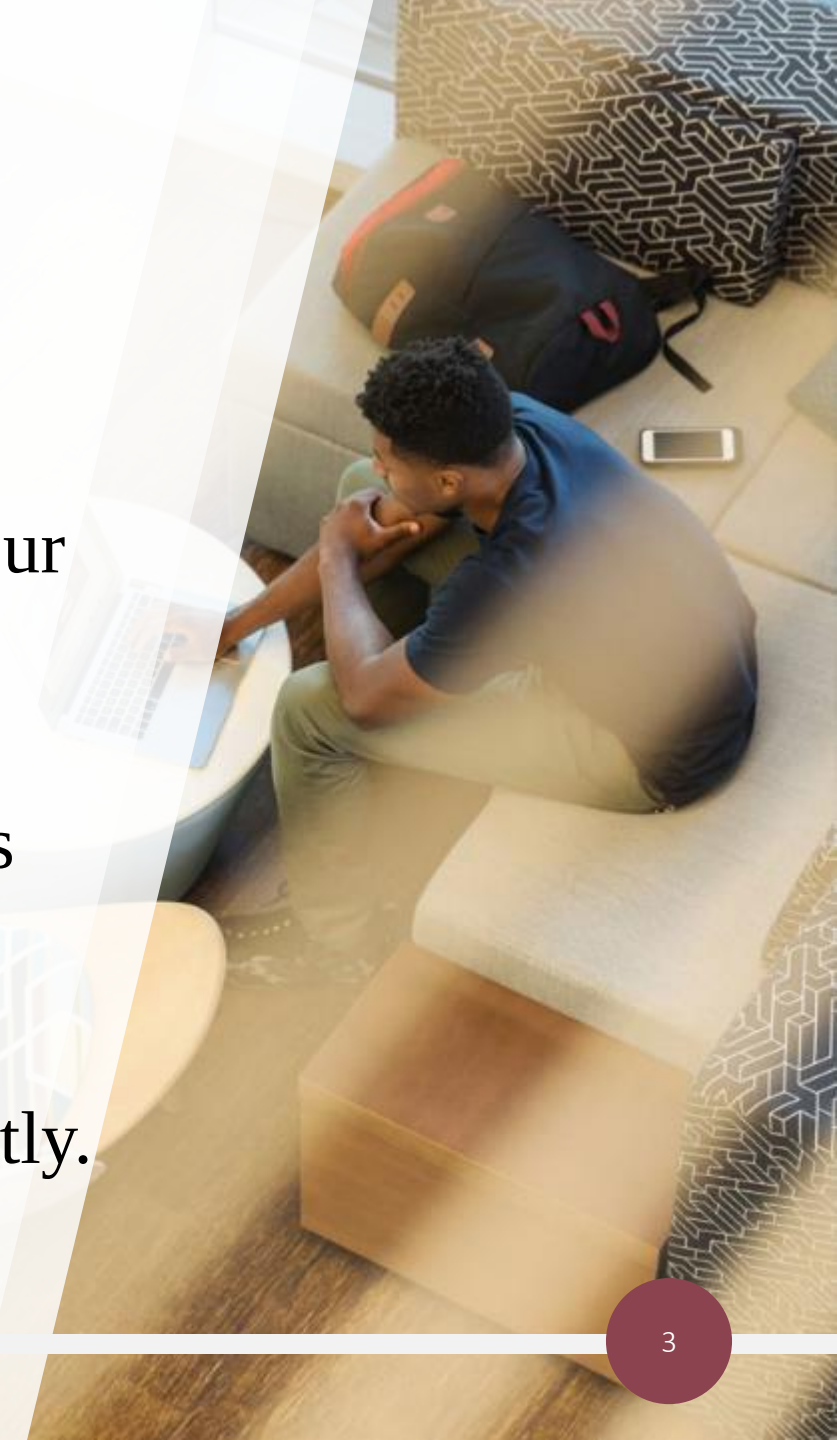
Java 11 (1Z0-819)

Annotations

- ✓ Create, apply, and process annotations

Annotations

- Metadata is information about information.
- Annotations, via metadata, enable us to add value to our code.
- We can annotate (assign metadata) to classes, methods variables etc..
- Though optional, when used they must be used correctly.

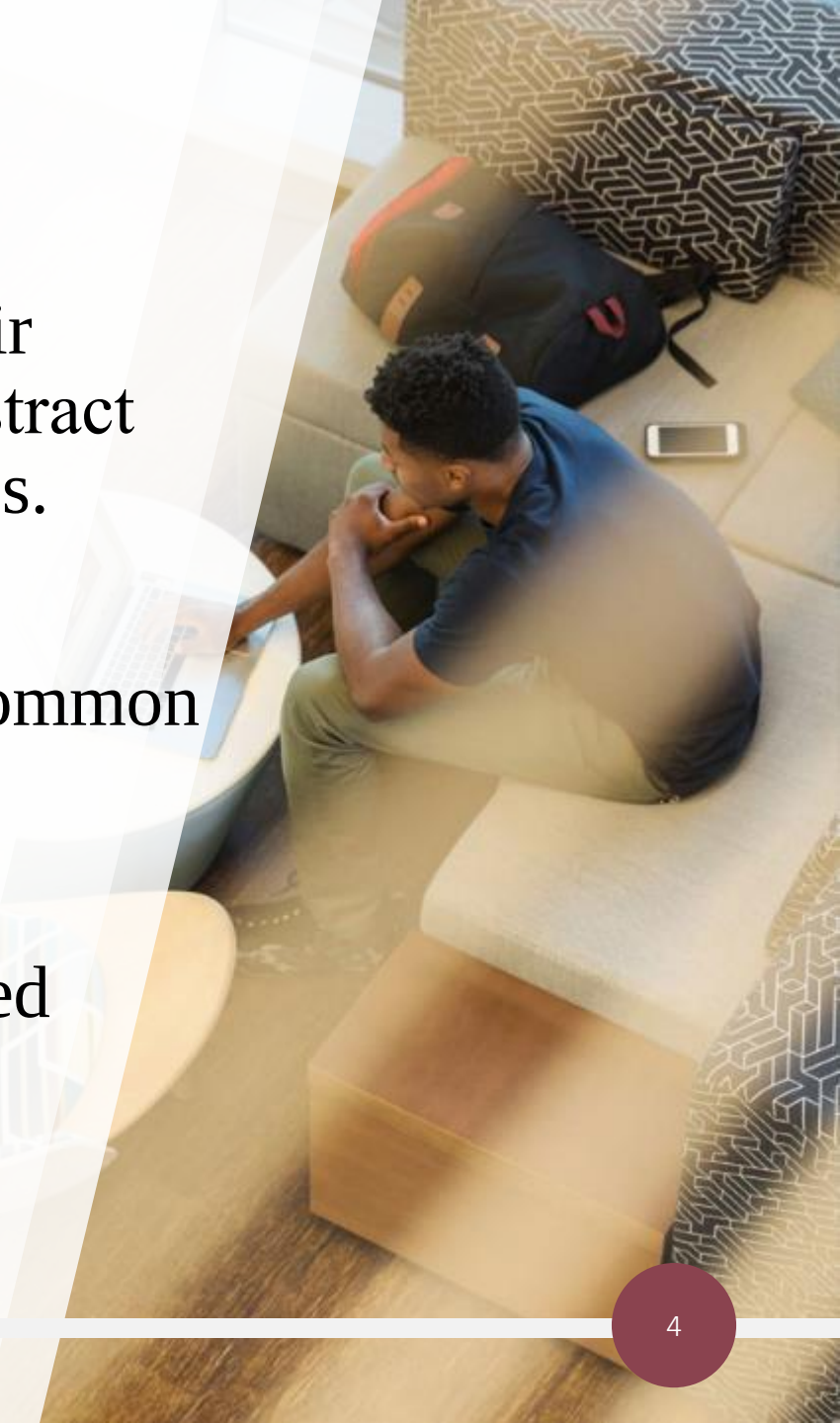


Annotations

- Annotations operate much like interfaces. In fact, their origins are in interfaces. Their “elements” look like abstract methods. This will be clearer when we look at examples.

- Annotation names are case sensitive; however, it is common practice to start the name with an uppercase letter.

- Like interfaces, annotations can be applied to unrelated classes.



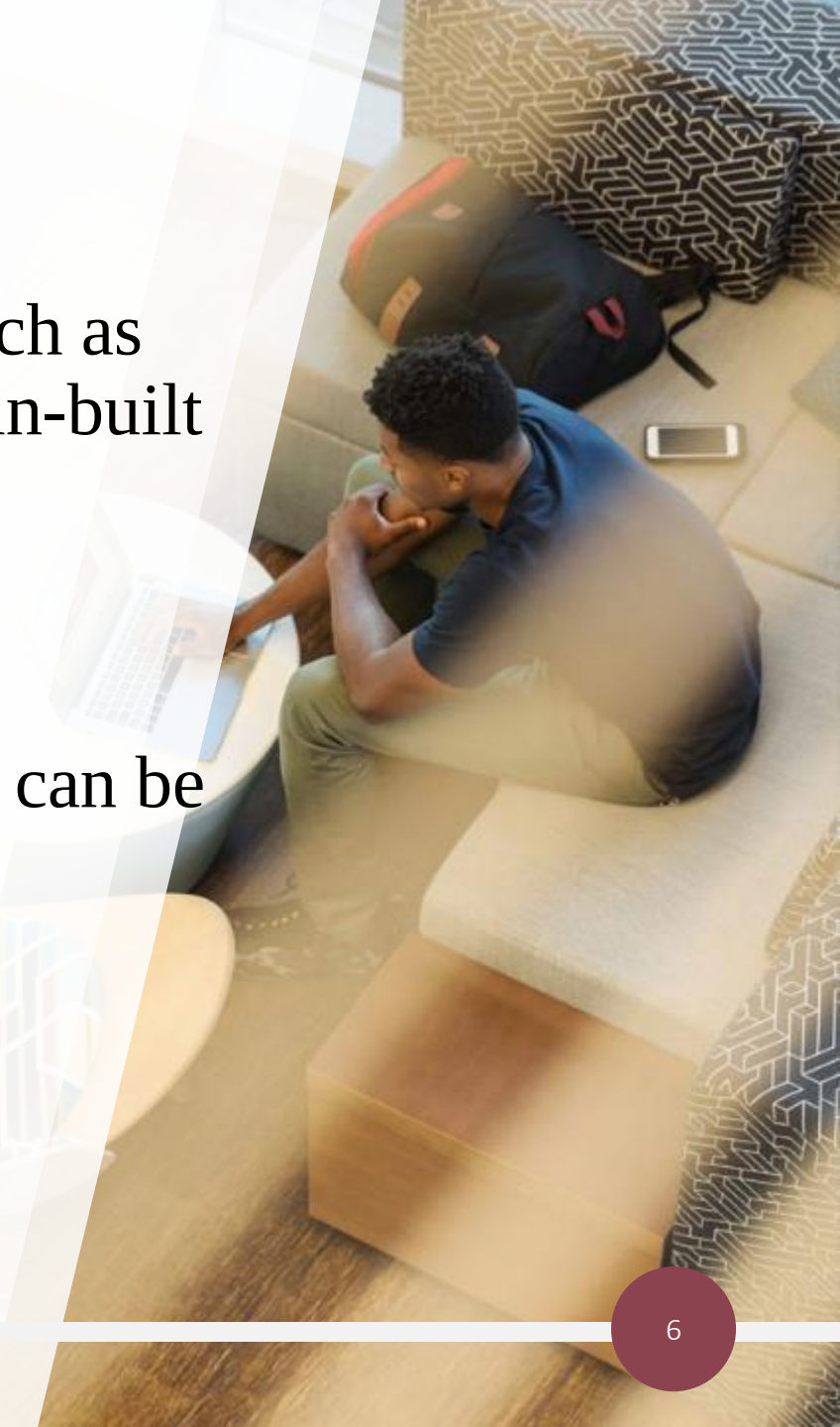
- CommonBuiltInAnnotations.java
- CommonBuiltInAnnotationsExtra.java

Annotations

- Previous programs focused on in-built annotations such as *@Override* and *@FunctionalInterface*. These specific in-built annotations can be applied to methods and interfaces respectively.

- Later we will deal with other in-built annotations that can be applied to *annotations* themselves.

- Now, we will turn our attention to defining custom annotations.



Annotations

- As stated earlier, annotations have a lot in common with interfaces; for example, a *marker annotation* has no elements (a *marker interface* has no methods). Annotations, as with interfaces, can be applied to unrelated classes.

- In fact, we annotate our annotation with *@interface*.

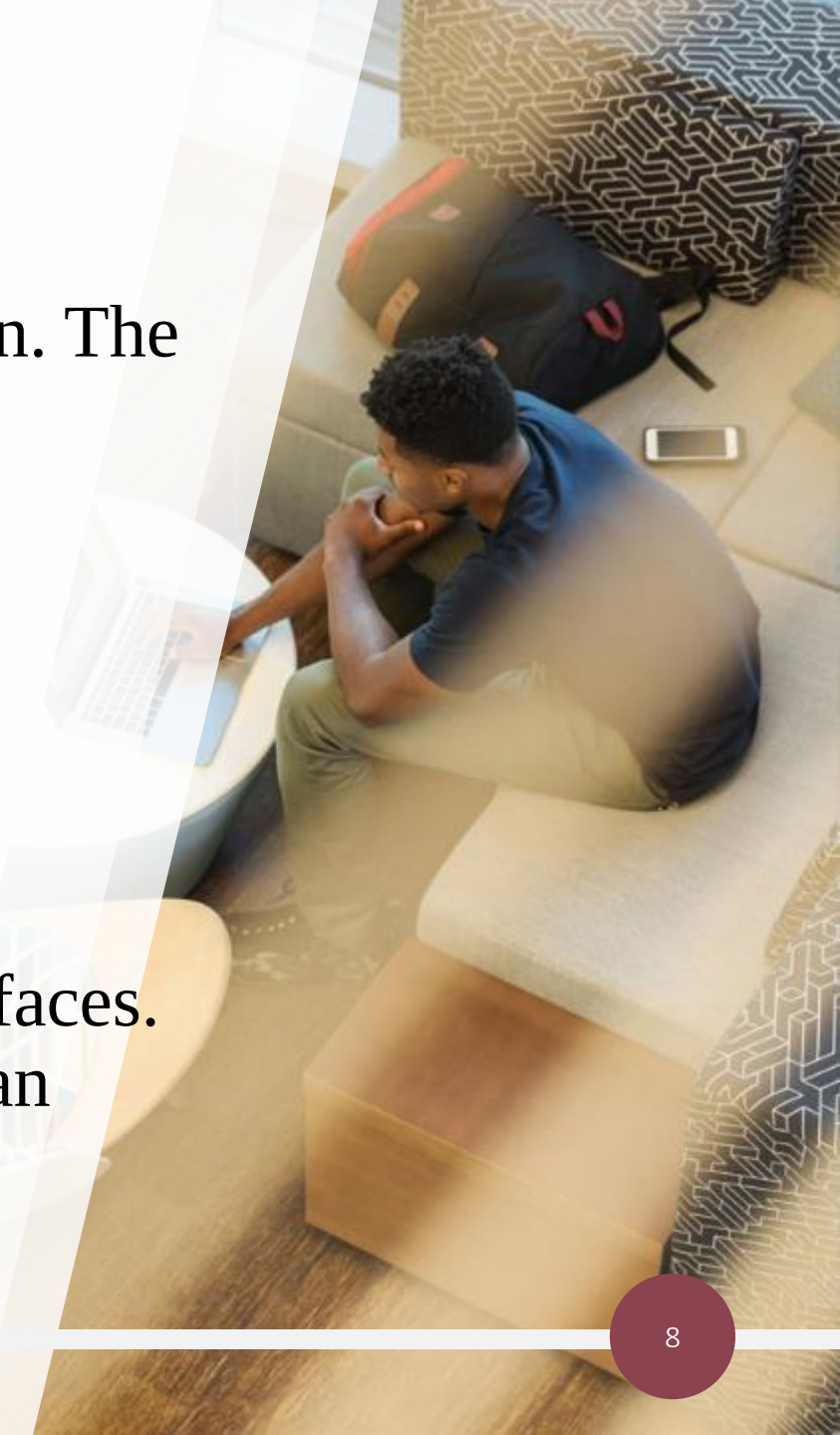
- *public @interface MyAnnotation { } // marker annotation*

Annotations

- An *annotation element* is an attribute of the annotation. The elements (attributes) can have values.

- *public @interface MyAnnotation {*
  int myElement(); // looks like an abstract method!
  }

- Remember that annotations have their origins in interfaces. Essentially, the JVM translates the above element into an interface method and the annotation itself as an implementation of the interface.



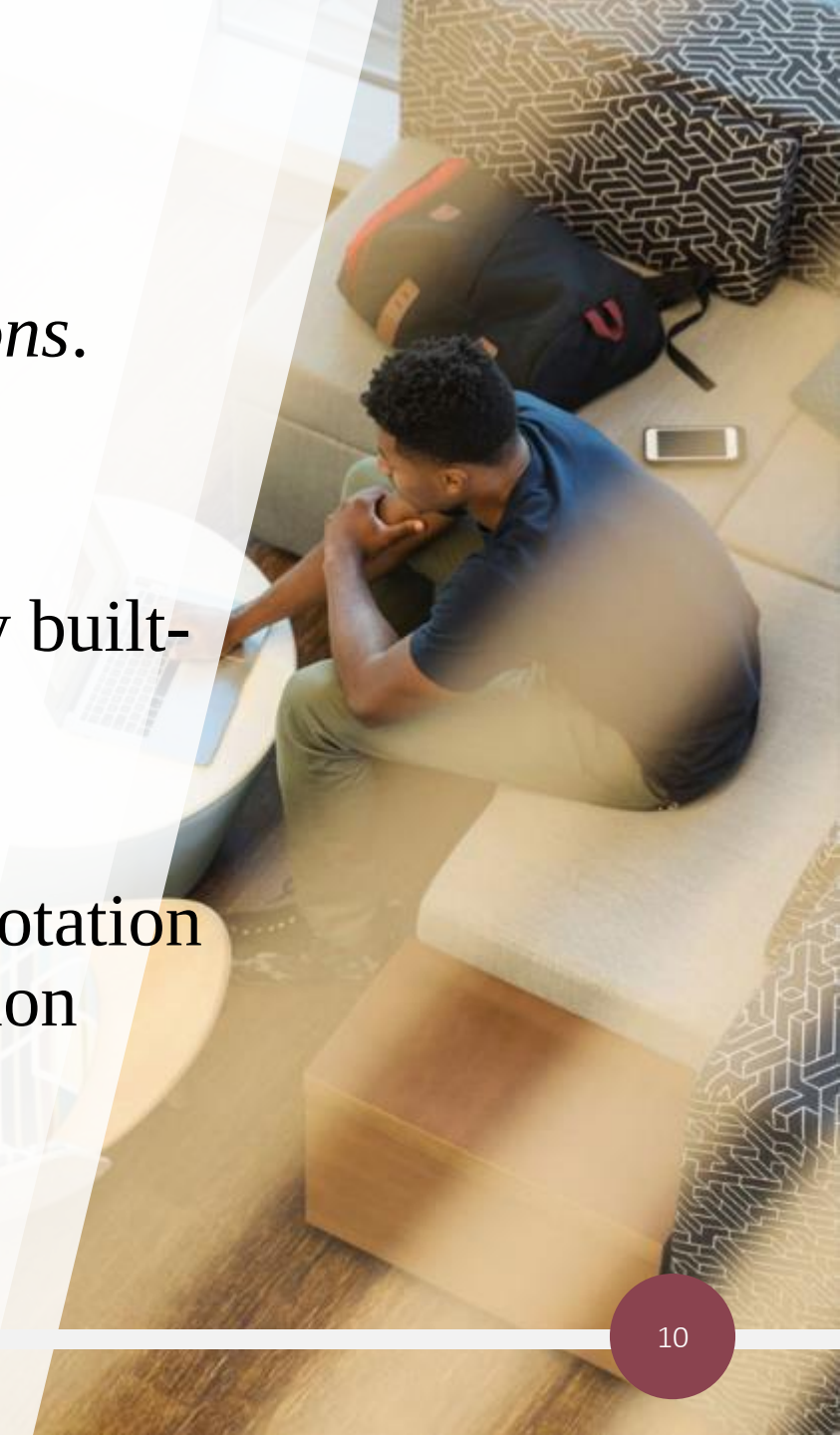
- CustomAnnotations.java

Annotations on Annotations

- There are annotations that can be applied *to annotations*.

- As opposed to applying built-in annotations to types (methods, interfaces, fields etc..); we can actually apply built-in annotations to annotations themselves.

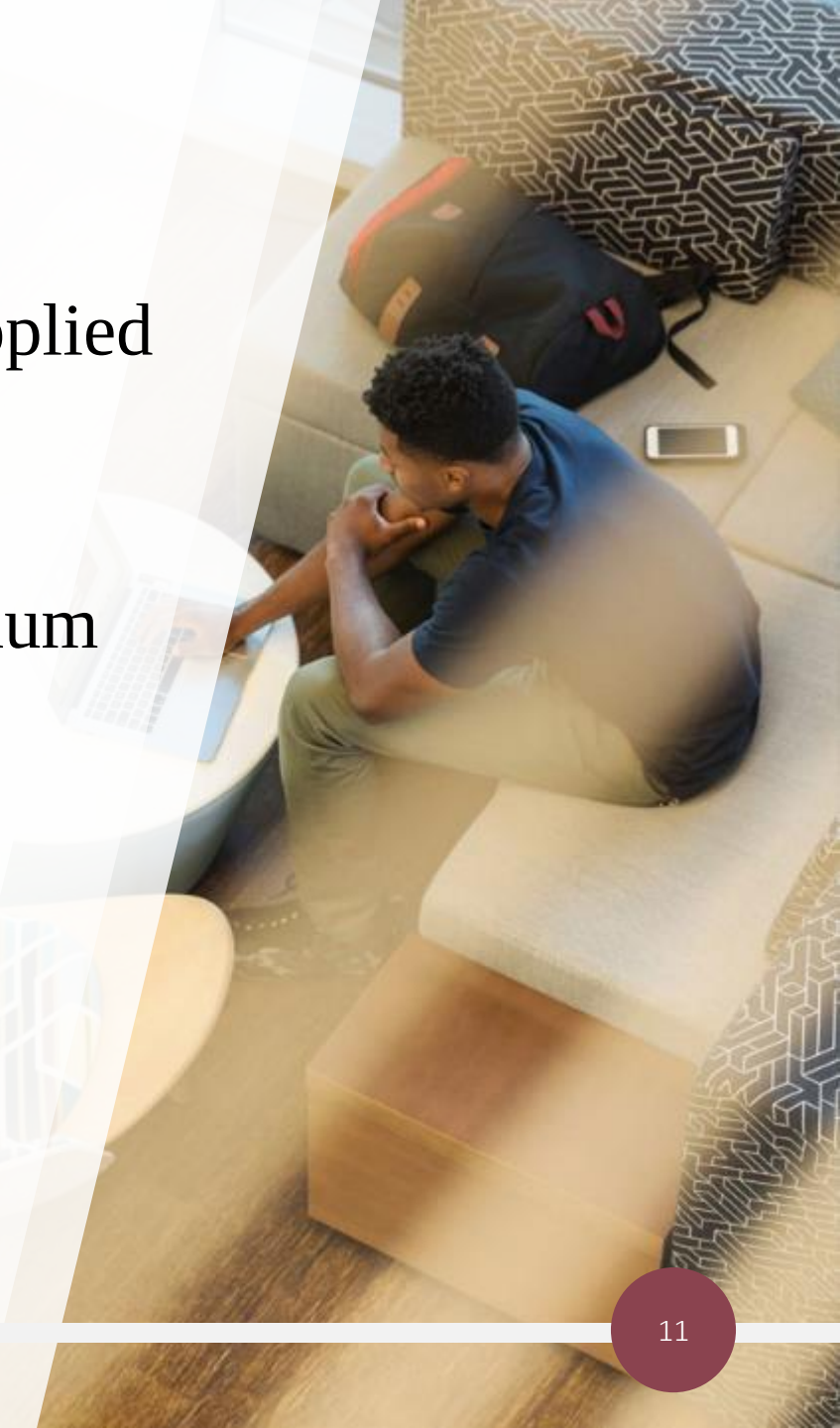
- For example, you might want to limit where your annotation can be used or you may want your annotation information included in Javadoc documentation.



@Target

- *@Target* limits the types that the annotation can be applied to.

- The types are specified as an array of *ElementType* enum values.



@Target

ElementType value	Scope (what it applies to)
TYPE	Interfaces, enums, classes, annotations.
METHOD	Method declarations
PARAMETER	Constructor and method parameters
FIELD	Instance and static variables
CONSTRUCTOR	Constructor declarations
LOCAL_VARIABLE	Local variables
ANNOTATION_TYPE	Annotations
TYPE_USE	Anywhere there is a Java data type. This includes where types are *used* e.g. object creation with *new*.

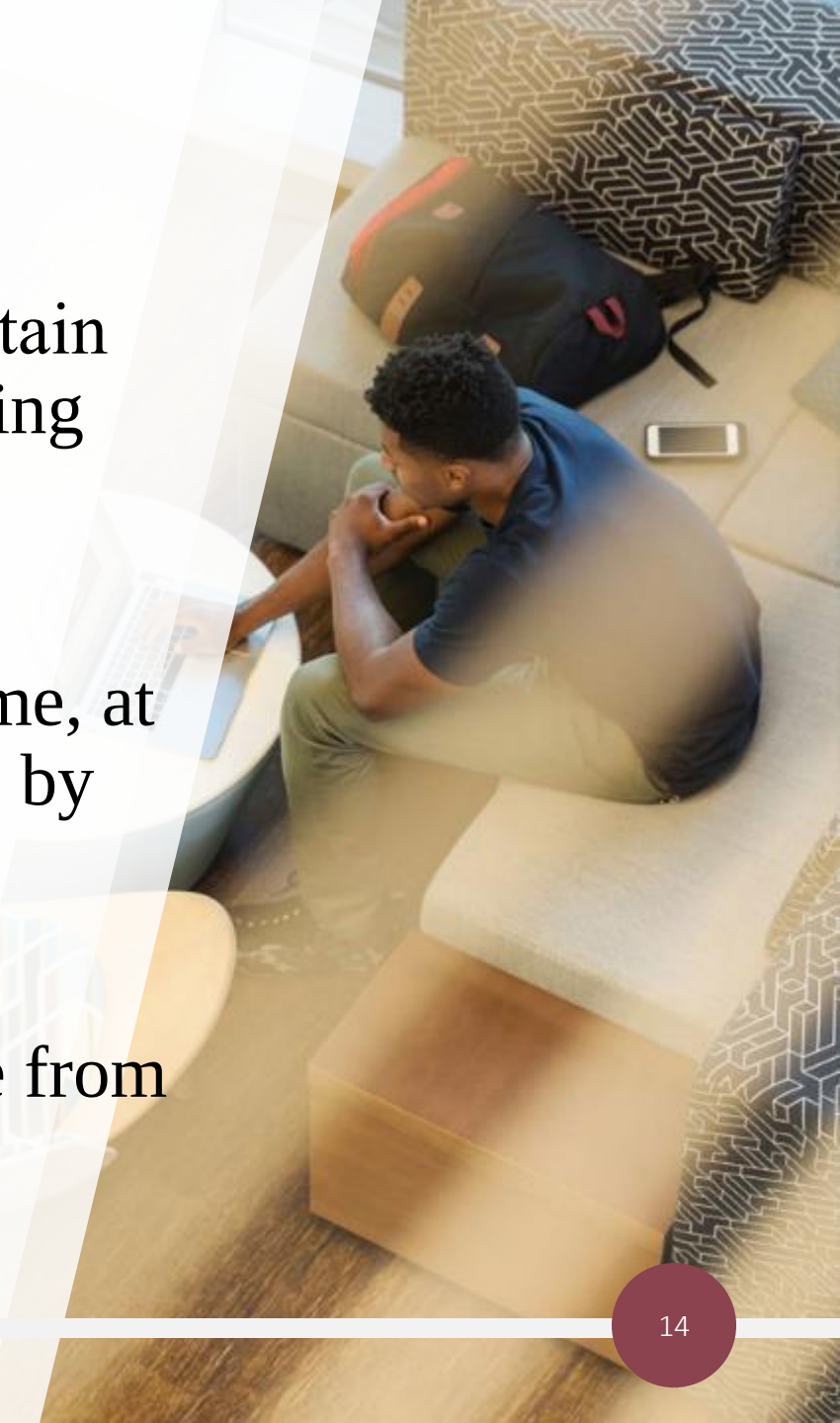
- TargetExample.java

@Retention

- In generics, we encountered “type erasure” where certain information is discarded by the compiler when converting source code into .class file.

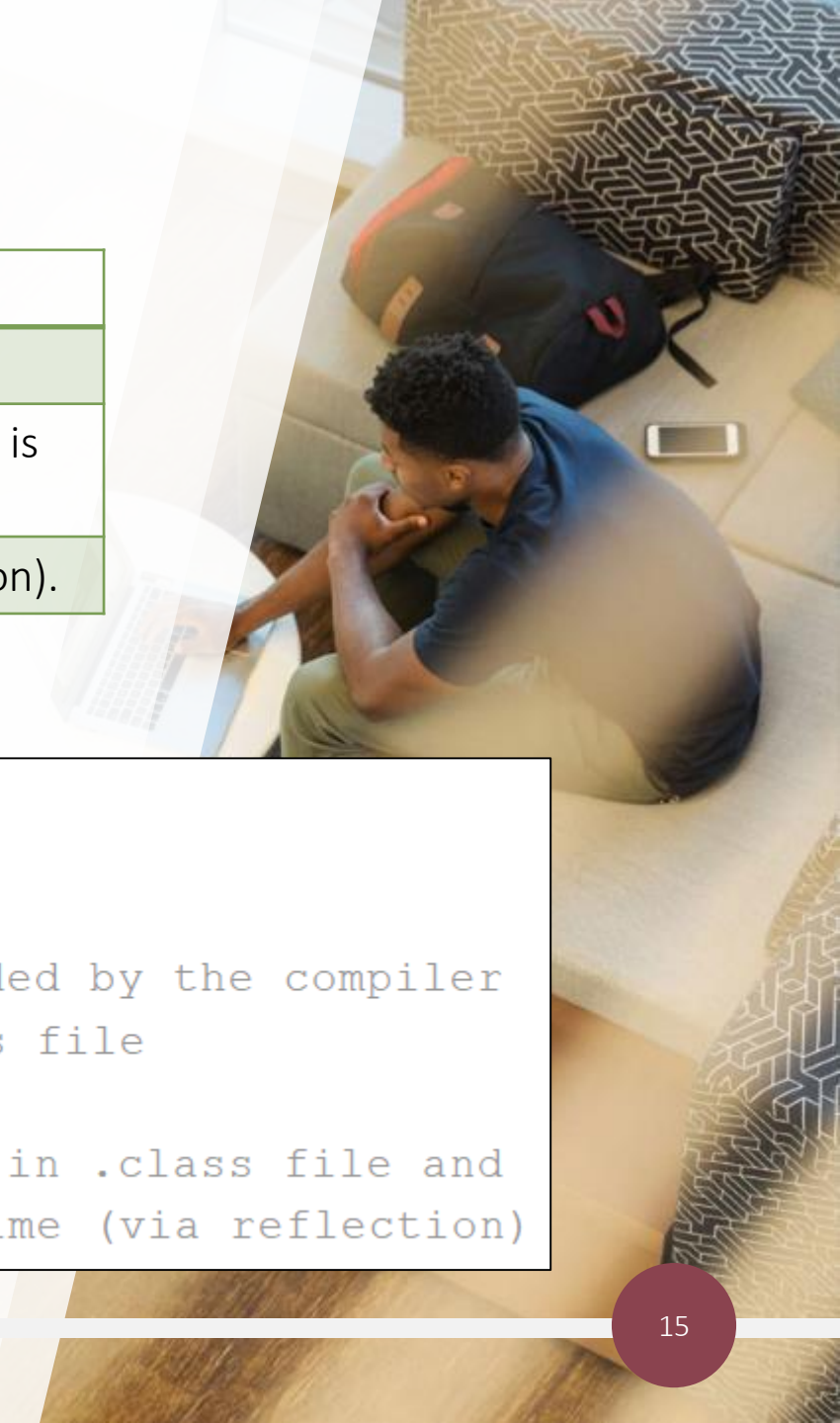
- Similarly annotations may be discarded at compile time, at runtime or not at all. We can control when this happens by using the *@Retention* annotation.

- We specify the level of retention using an enum value from *RetentionPolicy*.



@ Retention

RetentionPolicy value	Description
SOURCE	Source file only, compiler discards it.
CLASS	Stored in the .class file but not available at runtime. This is the default compiler behaviour.
RUNTIME	Stored in the .class and available at runtime (via reflection).



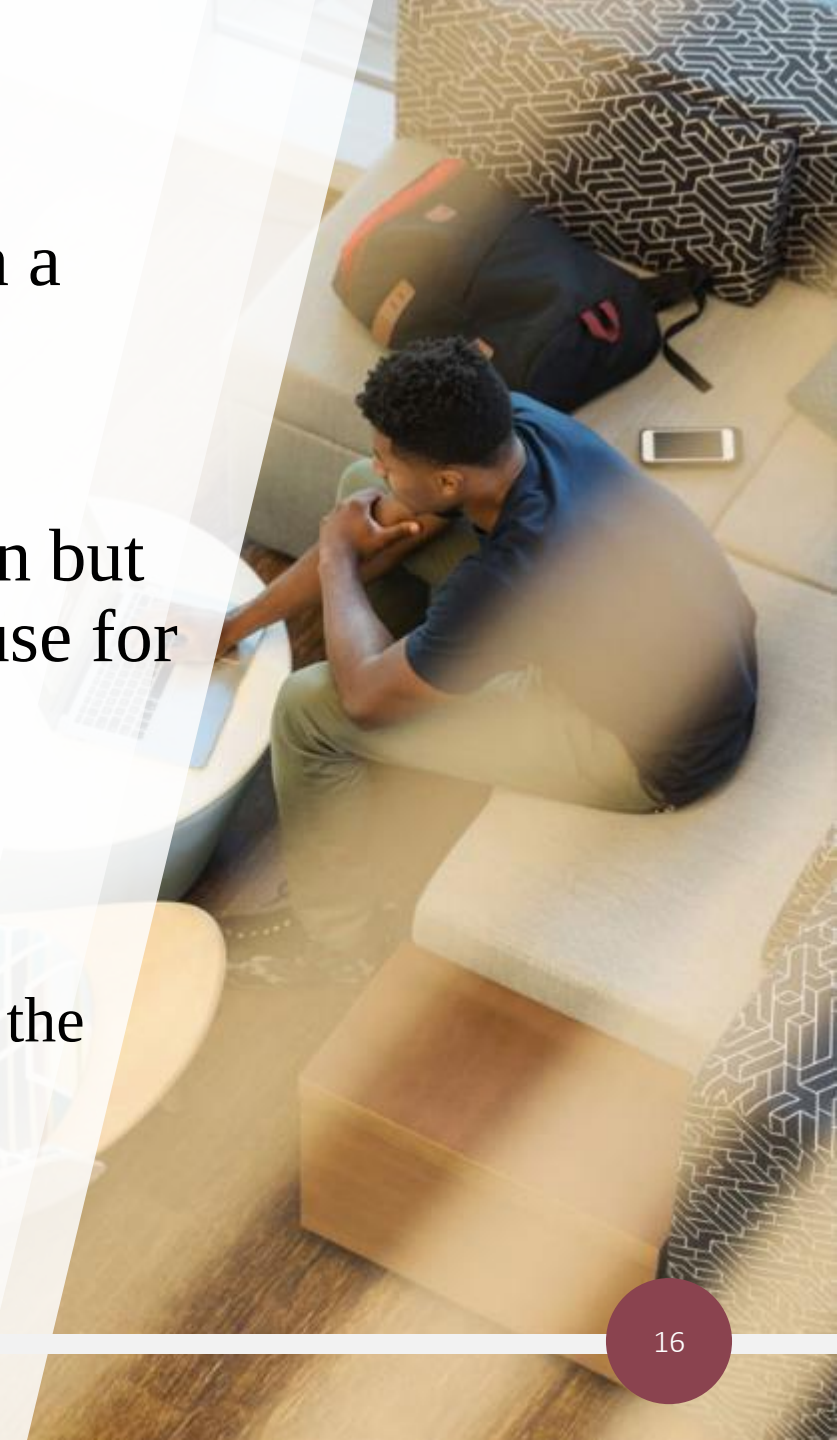
```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.SOURCE)   // annotation discarded by the compiler
@interface Mouse{}                   // i.e. not in .class file

@Retention(RetentionPolicy.RUNTIME) // annotation stored in .class file and
@interface Keyboard{}                // available at runtime (via reflection)
```

@Repeatable

- This annotation enables us to specify an annotation on a type more than once.

- This is useful if you wanted to use the same annotation but with different values each time; thus, it is not of much use for marker annotations (which have no elements).

- Requires two annotations:
  1. A container annotation which has a *value()* array element; the type of the array is the annotation you want to repeat.
  2. The annotation to want to repeat; which is annotated with: *@Repeatable(ContainerAnnotationName.class)*



- RepeatableExample.java