Session: 14

Annotations and Base64 Encoding

Objectives

- Explain declaring an annotation type in Java
- Describe predefined annotation types
- Explain Type annotations
- Explain Repeating annotations
- Describe Base64 encoding

Annotation:

- ✓As the name implies, is a syntactic metadata that are added to the Java source.
- ✓Is a specific part of the original source, however, does not affect the operation of the code.

Following elements are annotated in Java:

- Tables
- Methods
- Variables
- Classes
- Parameters

Introduction 3/3

An enhanced feature is introduced in Java 8 called **Pluggable Annotation Processing** API.

API helps to

- Write a customized Annotation Processor
- Code dynamically to operate on the set of annotations

Uses of Annotations

Declaring Annotations

To declare an annotation, prefix with `@` as follows:

Syntax:

`@<name>`

```
@<name>
```

Example:

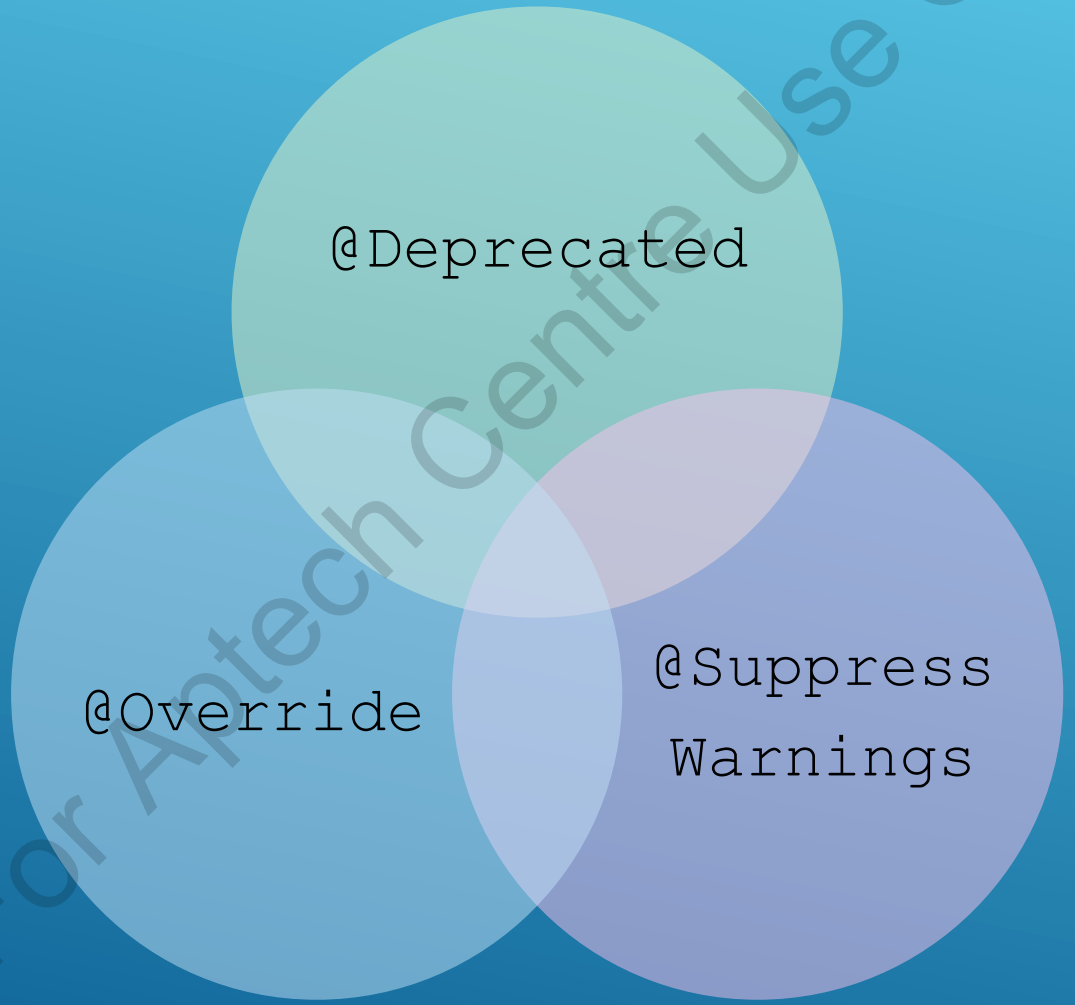
Here, `@(at)` symbol signals to the compiler that this is an annotation

```
@Item
```

Here, the annotation name is `Item`

Predefined Annotations 1/4

Following are three built-in annotations:



Predefined Annotations 2/4

`@Deprecate`	Annotation deprecates a class, method, or field
`@Override`	Creates a compile time check
`@SuppressWarnings`	Can compress compiler warnings

Predefined Annotations 3/4

Predefined annotations are demonstrated through following codes:

@Deprecate

```
@Deprecated
public class
SampleContent {
}
public class Test {
    public static void
    main(String args[]){
    SampleContent c = new
    SampleContent();
    …
}
}
```

@Override

```
public class ClassOne
{
public void show (String testmsg){
   System.out.println(testmsg);
}
public static void main(String args[]){
  SubClass obj = new SubClass();
  obj.show ("Good day!!");
}
}
class SubClass extends ClassOne
{
  @Override
  public void show (int testmsg){
  System.out.println(" I want to say: "+
  testmsg);
}
}
```

@SuppressWarnings

```
@SuppressWarnings
public void alertMethod() {
}

@SuppressWarnings("unmarked")
@SuppressWarnings({"unmarked",
"deprecation"})
```

Predefined Annotations 4/4

There are two advantages of using `@Override` annotation:

- If a programmer makes any unintentional error while overriding, then it results in a compile time error. Using `@Override` instructs compiler that it is overriding this method.
- It helps to enhance code readability.

Creating Custom Annotations 1/2

Custom Annotations created in Java are defined in their own files.

```
@interface SampleAnnotate{
String samplValue();
String name();
int age();
String[] addNames();
}
```

Creating Custom Annotations 2/2

Custom Annotations uses following syntax:

@Retention

```
@Retention(RetentionPolicy.RUNTIME)
@interface SampleAnnotate
{
...
}
```

@Target

```
@Target({ElementType.METHOD})
public @interface
SampleAnnotate{...
}
```

@Inherited

```
import
java.lang.annotation.Inherited;
@Inherited
public @interface
SampleAnnotate {
} @SampleAnnotate
class Person { ... }
public class Employee
extends Person { ... }
```

@Documented

```
import
java.lang.annotation.Documented;
@Documented
@interface TestAnnotate {
. . .
}
@TestAnnotate
public class Employee {
... }
```

Type Annotations

- Are formed to maintain better analysis of Java programs, ensuring **better type checking**.
- For example, to ensure that a variable in program is never assigned to null or to avoid triggering a `NullPointerException`, a custom plugin can be written.
- Then, modify code to annotate variable specifying that it is never assigned to null.
- Following syntax shows the variable declaration:

```
@NonNull String str;
```

Here,
`str` is a String variable in the annotation.

Compiler shows a warning if it notices a potential problem, allowing you to modify the code to avoid the error while calculating the NonNull module at the command line during code compilation. After editing the code to remove all warnings, this specific error will not occur when the program runs.

Repeating Annotations 1/4

Helps to apply the same annotations at multiple times for the same declaration.

```
@ScoreSchedule(dayOfMonth="last")
@ScoreSchedule(dayOfWeek="Wed", hour="21")
public void scorePapers() { ... }
```

Here, the annotation `@ScoreSchedule` has been applied twice, thus, it is a repeating annotation. Repeating annotations can be applied not only to methods also to any item that can be annotated.

Repeating Annotations 2/4

For compatibility, repeating annotations are loaded in a container annotation generated by Java compiler.

For the compiler to perform this, two declarations are necessary in the code:

1) Declare a Repeatable Annotation type.

```
/*
*Aptech Java8
*Java tester
*/
import java.lang.annotation.Repeatable;
@Repeatable(ScoreSchedules.class)
public @interface ScoreSchedule {
String monthDay() default "1st";
String weekDay() default "Monday";
int hour() default 12;
}
```

Repeating Annotations 3/4

2) Declare containing annotation type.

```
public @interface ScoreSchedules {
ScoreSchedule[] value();
}
```

Here, `ScoreSchedules` is the class for `@interface` annotation.

Repeating Annotations 4/4

The complete code is as follows:

```
import java.lang.annotation.Repeatable;
@Repeatable(ScoreSchedules.class)
@interface ScoreSchedule {
String monthDay() default "1st";
String weekDay() default "Monday";
int hour() default 12;
}
@interface ScoreSchedules {
ScoreSchedule[] value();
}
public class RepeatingDemo {
public static void main(String args[]){}
@ScoreSchedule(monthDay="last")
@ScoreSchedule(weekDay="Fri", hour=23)
public void scorePapers() { }
}
```

Processing Annotations Using Reflection 1/4

- Reflection API of Java can be used to access annotations on any type such as class or interface or methods.
- Several methods in Reflection API help to retrieve annotations.
- Methods that generate a single annotation, such as `AnnotatedElement.getAnnotationByType (Class <T>)` remain unchanged.
- They return only a single annotation when one annotation of the requisite type is available.
- If one or more annotation of the required type is available, it can be acquired by getting the container annotation.

Processing Annotations Using Reflection 2/4

```
import java.lang.reflect.Method;
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.WildcardType;
import java.util.List;

public class SampleRefl {
public static void main(String[] args)
throws NoSuchMethodException, SecurityException {
Method sampMeth = SampleRefl.class.getMethod("sampMeth", List.class);//here Method defined as sampMeth
ParameterizedType sampleLiTy = (ParameterizedType)sampMeth. getGenericParameterTypes()[0];//here List Type
//defined as SampleLiTy
ParameterizedType sampleClTy = (ParameterizedType)sampleLiTy. getActualTypeArguments()[0];//here Class Type
//defined as SampleClTy
WildcardType sampleGenTy = (WildcardType)sampleClTy. getActualTypeArguments()[0];//here generic Type
//defined as SampleGenTy
Class<?> SampleGenCl = (Class<?>)sampleGenTy.getUpperBounds()[0];// here generic Class defined as sampleGenCl
boolean isException = Exception.class. isAssignableFrom(SampleGenCl);
//to display whether the statement is true or false
System.out.println("This Class extends RuntimeException: " + isException);
boolean isRuntimeException = RuntimeException.class. isAssignableFrom(SampleGenCl);
// to display whether the statement is true or false"
System.out.println("This Class extends RuntimeException: " + isRuntimeException);
}
public void sampMeth(List<Class<? extends Exception>> exceptionClasses) {
}
}
```

Processing Annotations Using Reflection 3/4

Output:

```
This Class extends RuntimeException:true
This Class extends RuntimeException:false
```

Here, `SampleGenCl` class is reflected through `sampMeth()` method.

Processing Annotations Using Reflection 4/4

`@Functional` Interface:

The annotated element will operate as a functional interface and compiler generates an error if the element does not comply with the requirements.

```
@FunctionalInterface
interface MyCustomInterface
{
....}
```

Known Libraries Using Annotations

Used for purposes such as:

- Unit testing
- Code quality analysis
- XML parsing
- Dependency injection

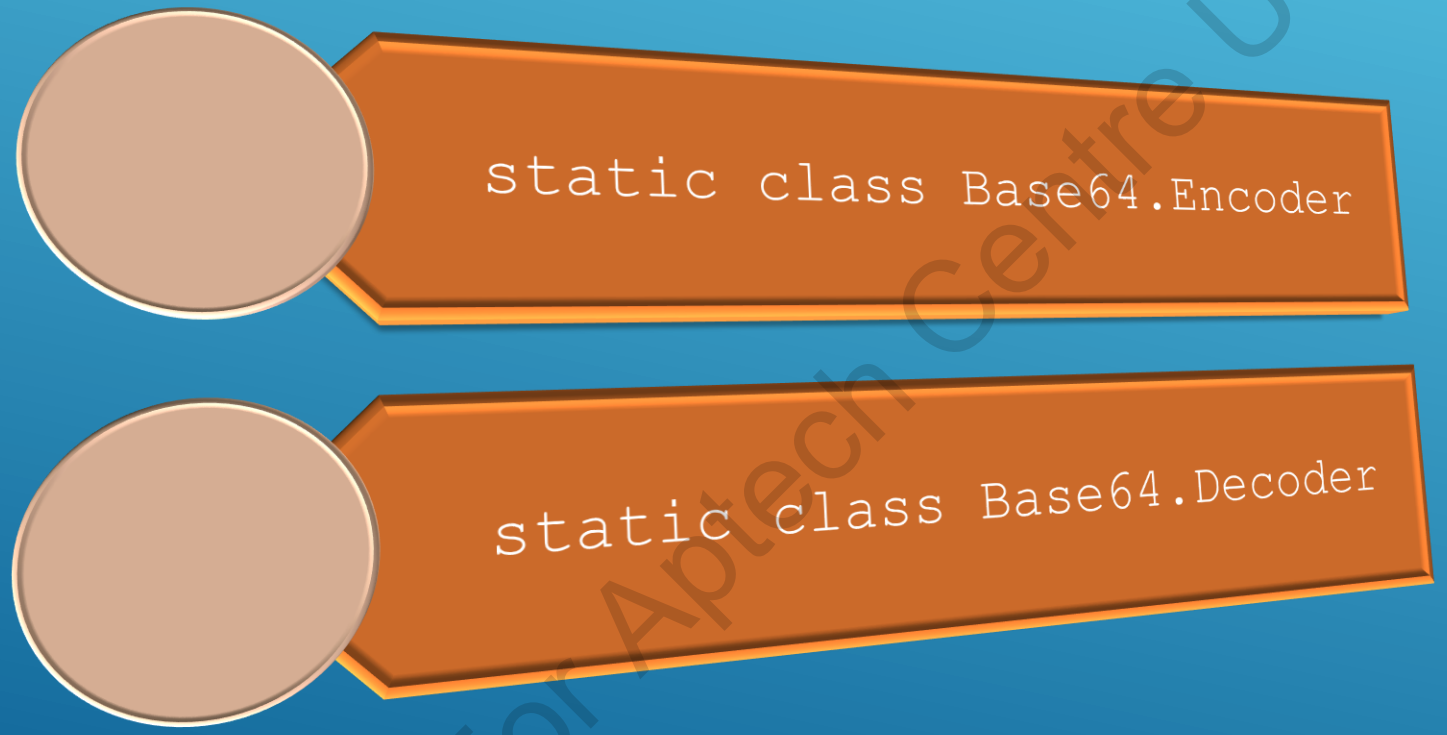
Base64 (java.util.Base64) Encoding 1/5

Java 8 encloses an inbuilt encoder and decoder for Base64 encoding.

Three types of Base64 encoding:

Simple	• Output is limited to a set of characters between A-Z, a-z, 0-9, and +.
URL	• The final output is safe from filename and URL and is limited to a set of characters between A-Z, a-z, 0-9, and +_.
MIME	• Output is limited to MIME friendly format.

Two nested classes in Base64:



Methods:

```
static Base64.Decoder getDecoder()
```

```
static Base64.Encoder getEncoder()
```

```
getMimeDecoder()
```

```
getMimeEncoder()
```

```
getMimeEncoder(intlineLength, byte[] lineSeparator)
```

```
getUrlDecoder()
```

```
getUrlEncoder()
```

Base64 (java.util.Base64) Encoding 4/5

Following code encodes a string to base64, then decode the same String back to a base64 encoded output stream:

```
import java.util.Base64;
import java.util.UUID;
import java.io.UnsupportedEncodingException;
/**Aptech Java
*Base64 example in detail
*/
public class HelloWorld {
public static void main(String args[]){
try {
// Encoding a string using Base64
String sampleBase64EncoStr = Base64.getEncoder(). encodeToString("AptechJava8".getBytes("utf-8"));
System.out.println("Encoded String (Basic) looks like this: " + sampleBase64EncoStr);
// Decoding a string using Base64
byte[] base64decodedBytes = Base64.getDecoder(). decode(sampleBase64EncoStr);
System.out.println("Decoded String is : " + new String(base64decodedBytes, "utf-8"));
sampleBase64EncoStr = Base64.getUrlEncoder(). encodeToString("AptechJava8".getBytes("utf-8"));
System.out.println("Encoded String (URL)looks like this: " + sampleBase64EncoStr);
StringBuilder strBuild = new StringBuilder();
for (int j = 0; j < 10; ++j) {
strBuild.append(UUID.randomUUID().toString());
}
byte[] sampleMimeBytes = strBuild.toString().getBytes("utf-8");
String sampleMimeEncoStr = Base64.getMimeEncoder(). encodeToString(sampleMimeBytes);
System.out.println("Encoded String (MIME) looks like this: " + sampleMimeEncoStr);
}catch(UnsupportedEncodingException u){// to display error
System.out.println("Unsupported Encoding Error: " + u.getMessage());
} } }
```

Base64 (java.util.Base64) Encoding 5/5

Output:

```
Encoded String (Basic) looks like this: QXB0ZWNoSmF2YTg=
Decoded String is: AptechJava8
Encoded String (URL) looks like this: QXB0ZWNoSmF2YTg=
Encoded String (MIME) look like this:
YzNlNmYyNjctY2JkNi00OTFiLThiMzQtZTMwZTJkZjBkZTU2YTljM2FlN2YtZTJlMi00YmVlLWI2
NDQtNjFjMGZiYmI4NDc5NWFmOTA3YjktYTc5Ni00MDBkLTg2ZjItOTllOGRhMDJkODE4YmZmZmYx
YzQtNjA1Yy00M2ZhLWJkNTMtNzEwOWI5NzYwMGY4NGE0YTZkZjUtZWQ5Zi00NWY0LWE3YTAtYmZi
MDU5MWRjMGM2OWVjOTYyNDEtZGU1Yi00YWFiLTllMmYtM2MyYjg3MjY4ZTYzNDg0MGMzYzctZjEy
MS00OTJiLThjM2MtZTQxMDExNmQzM2RlMDdlYjMyNTYtNmI2Mi00NjVmLTgyYzItOWY5OGQ3OTdh
ZmJlMzJiYzNhNWEtZmM1Mi00ZmI1LTg5NDAtZjdjZGJjOWFhYWVjNTg4Y2YxODktYjk2NC00YTJk
LTlhMWYtYmNmNGRmOTQ5MTI4
```

Summary

- Annotations are comments, notes, remarks, or explanations. In Java, annotations help to associate these additional information (also called metadata) to the program elements. Annotations can be determined from source files or class files at runtime.
- The @Deprecated annotation is used for deprecating or marking a class, method or field as deprecated, signifying that the part of code no longer will no longer be used.
- The @SuppressWarnings annotation can suppress the compiler warnings in any available method.
- Since, annotation types are piled up and stored in byte code files such as classes, the annotations reverted by these methods can be enquired as any systematic Java object.
- Base64 encoding has an in-built encoder and a decoder. In Java 8, there are three types of Base64 encoding namely, Simple, URL, and MIME.