Java: ANNOTATIONS

"Concept && Coding" YT Video Notes

What is Annotation?

- It is kind of adding ***META DATA*** to the java code.
- Means, its usage is ***OPTIONAL***.
- We can use this meta data information at ***runtime*** and can add certain logic in our code if wanted.
- How to Read Meta data information? Using ***Reflection*** as discussed in previous video.
- Annotations can be applied at anywhere like Classes, Methods, Interface, fields, parameters etc.

Example:

```
public interface Bird {

    public boolean fly();
}
```

Annotation (denoted using @)

```
public class Eagle implements Bird{

    @Override
    public boolean fly() {
        return true;
    }
}
```

TYPES OF ANNOTATIONS:

- Pre-Defined Annotations
  - Used on Annotations (called Meta-Annotations)
    - @Target
    - @Retention
    - @Documented
    - @Inherited
    - @Repeatable
      Java8 feature
  - Used on Java code
    Like Classes, Methods etc.
    - @Deprecated
    - @Override
    - @SupressWarnings
    - @FunctionalInterface
    - @SafeVarargs
- Custom Annotations / User Defined
  @{*Our custom name*}

Annotations used on Java Code:

@Deprecated:

- Usage of Deprecated Class or Method or fields, shows you compile time ***WARNING***.
- Deprecation means, no further improvement is happening on this and use new alternative method or field instead.
- Can be used over: **Constructor**, **Field**, **Local Variable**, **Method**, **Package**, **Parameter**, **Type**(class, interface, enum)

```
public class Mobile {

    @Deprecated
    public void dummyMethod(){
        //do something
    }
}
```

```
public class Main {
    public static void main(String args[])  {

        Mobile mobileObj = new Mobile();
        mobileObj.dummyMethod();
    }
}
```

```
public class Main {
    public static void main(String args[])  {

        Mobile mobileObj = new Mobile();
        mobileObj.dummyMethod();
```

Inspection 'Deprecated API usage' options

Press F1 to toggle preview

Inspection 'Deprecated API usage' has no quick-fixes for this problem. Click to edit inspection options, suppress the warning, or disable the inspection completely.

@Override:

- During Compile time, it will check that the method should be Overridden.
- And throws compile time error, if it do not match with the parent method.
- Can be used over: ***METHODS***.

```java
public interface Bird {

    public boolean fly();
}
```



```java
public class Eagle implements Bird{

    @Override
    public boolean fly1() {
        return true;
    }
}
```

@SupressWarnings:

- It will tell compiler to ***IGNORE*** any compile time ***WARNING***.
- Use it safely, could led to **Run time exception** if, any valid warning is IGNORED
- Can be used over: **Field**, **Method**, **Parameter**, **Constructor**, **Local Variable**, **Type** (Class or interface or enum)

```java
public class Mobile {

    @Deprecated
    public void dummyMethod(){
        //do something
    }
}
```

```java
public class Main {

    public static void main(String args[])  {
    }

    public void unusedMethod(){

    }
}
```

```java
public class Main {
    public static void main(String args[])  {

        Mobile mobileObj = new Mobile();
        mobileObj.dummyMethod();
    }
}
```

```java
@SuppressWarnings("unused")
public class Main {
    public static void main(String args[])  {
    }

    public void unusedMethod(){

    }
}
```

```java
public class Main {

    @SuppressWarnings("deprecation")
    public static void main(String args[])  {
        Mobile mobileObj = new Mobile();
        mobileObj.dummyMethod();
    }
}
```

OR

```java
@SuppressWarnings("deprecation")
public class Main {

    public static void main(String args[])  {
        Mobile mobileObj = new Mobile();
        mobileObj.dummyMethod();
    }
}
```

OR

```java
public class Main {

    @SuppressWarnings("all")
    public static void main(String args[])  {
        Mobile mobileObj = new Mobile();
        mobileObj.dummyMethod();
    }
}
```

@FunctionalInterface:

- *Restrict Interface to have only 1 abstract method.*
- Throws Compilation error, if more than 1 abstract method found.
- Can be used over: **Type** (Class or interface or enum)

```java
@FunctionalInterface
public interface Bird {
    public boolean fly();
    public void eat();
}
```

@SafeVarargs:

- *Used to suppress "Heap pollution warning"*
- *Used over methods and Constructors which has **Variable Arguments** as parameter.*
- Method should be either static or final (i.e. methods which can not be overridden)
- In **Java9**, we can also use it on private methods too.

What is Heap Pollution?

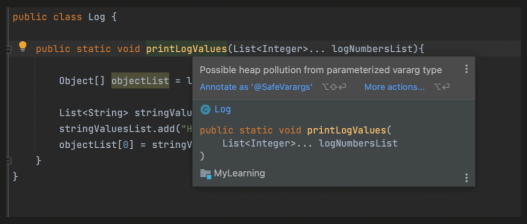
Object of One Type (Example String), storing the reference of another type Object (Example Integer)

```java
public class Log {

    public static void printLogValues(List<Integer>... logNumbersList){

        Object[] objectList = logNumbersList;

        List<String> stringValuesList = new ArrayList<>();
        stringValuesList.add("Hello");
        objectList[0] = stringValuesList;
    }
}
```




```java
public class Log {

    @SafeVarargs
    public static void printLogValues(List<Integer>... logNumbersList){

        Object[] objectList = logNumbersList;

        List<String> stringValuesList = new ArrayList<>();
        stringValuesList.add("Hello");
        objectList[0] = stringValuesList;
    }
}
```

Annotations used over Another Annotations (META-ANNOTATIONS):

@Target:
- This meta-annotation will restrict, where to use the annotation.
Either at method or Constructor or fields etc..

```
@Target(ElementType.METHOD)

public @interface Override {
}
```

```
@Target({ElementType.CONSTRUCTOR, ElementType.METHOD})
public @interface SafeVarargs {}
```

Element Type:

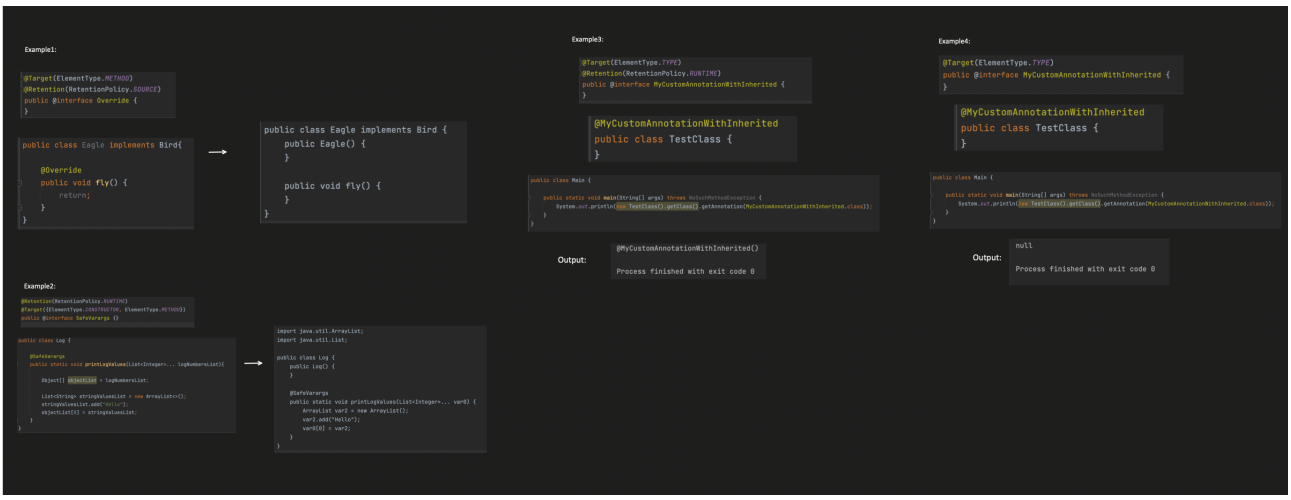
TYPE
FIELD,
METHOD,
PARAMETER,
CONSTRUCTOR,
LOCAL_VARIABLE,
ANNOTATION_TYPE,
PACKAGE,
TYPE_PARAMETER (allow you to apply on generic types <T>)
TYPE_USE (Java 8 feature, allow you to use annotation at all places where Type you can declare (like List< @annotation String>

@Retention:
- This meta-annotation tells, how Annotation will be stored in java.

RetentionPolicy.SOURCE: Annotations will be discarded by the compiler itself and it will not be recorded in .class file

RetentionPolicy.CLASS: Annotations will be recorded in .class file but will be ignore by JVM at run time.

RetentionPolicy.RUNTIME: Annotations will be recorded in .class file + available during run time. Usage of reflection can be done.



@Documented:

- By default, Annotations are ignored when Java Documentation is generated.
- With this meta-annotation even Annotations will come in Java Docs.

NOT DOCUMENTED:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

```
public class Eagle implements Bird{

    @Override
    public void fly() {
        return;
    }
}
```

Class Eagle

java.lang.Object
  Eagle

All Implemented Interfaces:
Bird

```
public class Eagle
extends java.lang.Object
implements Bird
```

Constructor Summary

Constructors

Constructor and Description
Eagle()

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
void	fly()

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Eagle

public Eagle()

Method Detail

fly

public void fly()

Specified by:
fly in interface Bird

```java
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.CONSTRUCTOR, ElementType.METHOD})
public @interface SafeVarargs {}
```

```java
public class Log {

    @SafeVarargs
    public static void printLogValues(List<Integer>... logNumbersList){

        Object[] objectList = logNumbersList;

        List<String> stringValuesList = new ArrayList<>();
        stringValuesList.add("Hello");
        objectList[0] = stringValuesList;
    }
}
```


Class Log

java.lang.Object
  Log

public class **Log**
extends java.lang.Object

Constructor Summary

Constructors

Constructor and Description
`Log()`

Method Summary

All Methods | **Static Methods** | **Concrete Methods**

Modifier and Type	Method and Description
static void	`printLogValues(java.util.List<java.lang.Integer>... logNumbersList)`

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Log

```
public Log()
```

Method Detail

printLogValues

```
@SafeVarargs
public static void printLogValues(java.util.List<java.lang.Integer>... logNumbersList)
```

@Inherited:

- By default, Annotations applied on parent class are not available to child classes.
- But it is after this meta-annotation.
- This Meta-annotation has no effect, if annotation is used other than a class.

```java
@Inherited
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyCustomAnnotationWithInherited {
}
```

```java
@MyCustomAnnotationWithInherited
public class ParentClass {
}
```

```java
public class ChildClass extends ParentClass{
}
```

```java
public class Main {

    public static void main(String[] args) {
        System.out.println(new ChildClass().getClass().getAnnotation(MyCustomAnnotationWithInherited.class));
    }
}
```

```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyCustomAnnotationWithInherited {
}
```

```java
@MyCustomAnnotationWithInherited
public class ParentClass {
}
```

```java
public class ChildClass extends ParentClass{
}
```

```java
public class Main {

    public static void main(String[] args) {
        System.out.println(new ChildClass().getClass().getAnnotation(MyCustomAnnotationWithInherited.class));
    }
}
```

Output

```
@MyCustomAnnotationWithInherited()

Process finished with exit code 0
```

Output

```
null

Process finished with exit code 0
```

@Repeatable:
- Allow us to use the same annotation more than 1 at same place.

We can not do this before JAVA8:

```
@Deprecated
@Deprecated
public class Eagle{
    public void fly() {

    }
}
```

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Category {

    String name();
}
```

```
@Category(name = "Bird")
@Category(name = "LivingThing")
public class Eagle{
    public void fly() {

    }
}
```

We need to use @Repetable Meta-annotation

```
@Repeatable(Categories.class)
@interface Category {
    String name();
}
```

```
@Retention(RetentionPolicy.RUNTIME)
 @interface Categories {

    Category[] value();
 }
```



```
@Category(name = "Bird")
@Category(name = "LivingThing")
@Category(name = "carnivorous")
public class Eagle{
    public void fly() {

    }
}
```

```java
public class Main {

    public static void main(String[] args) {

        Category[] categoryAnnotationArray = new Eagle().getClass().getAnnotationsByType(Category.class);
        for(Category annotation: categoryAnnotationArray){
            System.out.println(annotation.name());
        }
    }
}
```

Output:

```
Bird
LivingThing
carnivorous

Process finished with exit code 0
```

User Defined or Custom Annotations:

- We can create our own ANNOTATION using keyword ***"@interface"***

Creating an Annotation with empty body:

```java
public @interface MyCustomAnnotation {
}
```

```java
@MyCustomAnnotation
public class Eagle{
    public void fly() {

    }
}
```

Creating an Annotation with method (its more like a field):

- No parameter, no body.
- Return type is restricted to Primitive, Class, String, enums, annotations and array of these types.

```java
public @interface MyCustomAnnotation {

    String name();
}
```

```java
@MyCustomAnnotation(name = "testing")
public class Eagle{
    public void fly() {

    }
}
```

Creating an Annotation with an element with Default values:

- **Default value can not be null**

```java
public @interface MyCustomAnnotation {

    String name() default "hello";
}
```

```java
@MyCustomAnnotation
public class Eagle{
    public void fly() {

    }
}
```