**Аннотации** - это специальная форма синтаксических метаданных, которая может быть добавлена в исходный код. Они не оказывают влияния на работу программы. Является специальным интерфейсом.

Применение:

* **Информация для компилятора** – аннотации могут быть использованы компилятором для обнаружения ошибок или подавления предупреждений.
* **Обработка во время компиляции и развертывания** – программные инструменты могут обрабатывать информацию аннотаций для создания кода, XML-файлов и т.д. (Lombok, Spring).
* **Обработка во время выполнения** – аннотации могут быть проверены во время выполнения, чтобы настроить поведение программы

Аннотированы могут быть пакеты, классы, методы, переменные и параметры.

@Override

void mySuperMethod() { ... }

Можно применять несколько однотипных аннотаций:

@Author(name = "Jane Doe")

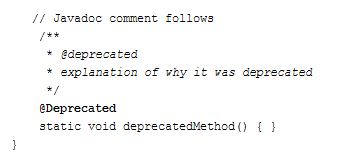
@Author(name = "John Smith")

class MyClass { ... }

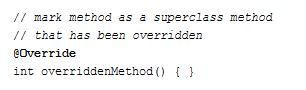
Аннотации могут также содержать параметры.

**Встроенные в Java аннотации**

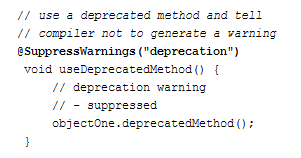
* @Deprecated – указывает, что элемент является устаревшим и больше не должен использоваться. Обычно в таком случае в Javadoc он документируется с помощью тега @deprecated.



* @Override – сообщает компилятору, что метод должен переопределять метод, объявленный в суперклассе. Компилятор выдаст ошибку, если метод не найден в родительском классе.



* @SuppressWarnings – указывает компилятору подавить определенные предупреждения, которые он мог выдать.



* @SafeVarargs – утверждает, что код не выполняет потенциально опасных операций над параметром varargs. Подавляет предупреждения, связанные с varargs.
* @ FunctionalInterface – указывает, что объявление типа должно быть функциональным интерфейсом. То есть интерфейс должен иметь единственный абстрактный метод.

**Аннотации, которые применяются к другим аннотациям**

Такие аннотации называются метаанотации.

* @Retention определяет, как хранится помеченная аннотация.

RetentionPolicy.SOURCE - помеченная аннотация сохраняется только на уровне исходного кода и игнорируется компилятором. Компилятор обрабатывает ее, но не сохраняет в скомпилированном байт-коде. Используется для дополнительных проверок кода на валидность. Либо генерация исходного кода как в Lombok.

RetentionPolicy.CLASS - помеченная аннотация сохраняется компилятором в ..class во время компиляции, но игнорируется виртуальной машиной Java (JVM).

RetentionPolicy.RUNTIME - помеченная аннотация сохраняется в JVM, чтобы ее можно было использовать во время выполнения.

* @Documented указывает, что аннотация будет указана в сгенерированной документации Javadoc. То есть в Javadoc будет указано, что класс помечен аннотацией.
* @Target указывает элементы, к которым может быть применена аннотация.

ElementType.ANNOTATION\_TYPE может быть применена к типу аннотации.

ElementType.CONSTRUCTOR может быть применен к конструктору.

ElementType.FIELD может быть применен к полю или свойству.

ElementType.LOCAL\_VARIABLE может быть применен к локальной переменной.

ElementType.METHOD может быть применен к аннотации на уровне метода.

ElementType.PACKAGE может быть применен к объявлению пакета. *Для этого нужно создать файл package-info.java и в нем написать*

*@PackageAnnotation(“value”)*

*package my\_pakage;*

ElementType.PARAMETER может быть применен к параметрам метода.

ElementType.TYPE может быть применен к классу, элементу или перечислению.

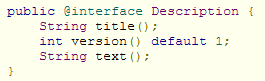
ElementType.TYPE\_PARAMETER может быть применена к параметру типа.

ElementType.TYPE\_USE может быть применена к использованию типа.

* @ Inherited – данная аннотация будет автоматически наследоваться в дочерних классах класса, помеченного этой аннотацией. (Работает только с классами. Если применит аннотацию к методу, она не будет наследоваться в потомках)
* @ Repeatable – указывает, что аннотация может быть применена несколько раз к одному и тому же элементу.

**Создание собственных аннотаций**

Для описания собственной аннотации используется ключевое слово **@interface.** Объявление аннотации может включать ключевое слово extends. Но все аннотации автоматически расширяют интерфейс Annotation.



Аннотации состоят только из объявления методов. Тело этих методов реализуется средствами java. Эти методы ведут себя аналогично полям.

* В качестве параметров аннотации могут быть использованы примитивы, String, Enum, Class, Annotaion либо массивом предыдущих элементов.
* Чтобы указать значение свойства по умолчанию используется ключевое слово default.
* Аннотации не наследуются.
* Все методы аннотаций должны быть без параметров и в них нельзя указывать оператор throws.
* Аннотации не могут быть обобщенными.

Создавать обработчики RUNTIME аннотаций можно с помощью рефлексии. Имея объект типа Class<?> мы можем использовать следующие методы для получения доступа к аннотациям:

* *getAnnotation(Class<A> annotationClass)* – получить аннотацию по объекту Class аннотации.
* *getAnnotations()* – получить все аннотации класса.
* *getDeclaredAnnotations()* – получить аннотации, используемые непосредственно в классе (не возвращает аннотации суперкласса).
* *isAnnotationPresent(Class<? extends Annotation> annotationClass) –* проверить, есть ли аннотация в классе.
* *getAnnotationsByType(Class<A> annotationClass), getDeclaredAnnotationsByType(Class<A> annotationClass) –* используются для повторяющихся аннотаций.

Чтобы написать обработчик SOURCE аннотаций нужно наследоваться от класса javax.annotation.processing.AbstractProcessor. Обработчик будет являться отдельным приложением (Jar file). Если компилятор находит такой обработчик, то он запускает его в отдельной jvm, и наш обработчик дополнительно обрабатывает исходники.

Нужно переопределить метод process, который и будет обрабатывать аннотации.

// типы аннотаций, поддерживаемые обработчиком  
@SupportedAnnotationTypes("annotations.source.ToString")  
public class AnnotationProcessor extends AbstractProcessor {  
 @Override  
 public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {  
 // перебираем в цикле аннотации  
 for (TypeElement annotation : annotations) {  
 // получаем аннотированные элементы  
 Set<? extends Element> annotatedElements = roundEnv.getElementsAnnotatedWith(annotation);  
  
 // map в котором проверяем геттер или не геттер  
 Map<Boolean, List<Element>> annotatedMethods = annotatedElements.stream().collect(Collectors.*partitioningBy*(element -> element.getSimpleName().toString().startsWith("get")));  
 List<Element> getters = annotatedMethods.get(true);  
 List<Element> otherMethods = annotatedMethods.get(false);  
   
 // кидаем ошибку, если аннотированы неправильные методы  
 otherMethods.forEach(element -> processingEnv.getMessager().printMessage(Diagnostic.Kind.*ERROR*, "@ToString must be applied to a getXxx method", element));  
  
 if (getters.isEmpty()) {  
 continue;  
 }  
  
 String className = ((TypeElement) getters.get(0).getEnclosingElement()).getQualifiedName().toString();  
  
 List<String> stringGetters = getters.stream().map(getter -> getter.getSimpleName().toString()).collect(Collectors.*toList*());  
 try {  
 writeBuilderFile(className, stringGetters);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 return true;  
 }

Обработка CLASS аннотаций может выполнятся до загрузки байт кода в jvm и создания объекта Class. Таким образом, мы можем поменять байт код. Retention.CLASS используется в так называемых агентах, которые позволяют проводить манипуляции с байткодом.

**Маркерные аннотации –** аннотации которые не содержат членов.

**Одночленная аннотация –** аннотация которая состоит из единственного члена. Если имя этого члена value, то при написании аннотации не нужно указывать имя члена (@Bean(“class”)). Такой синтаксис можно использовать если другие члены имеют значение по умолчанию.

**Типовые аннотации**

В JDK8 появилась возможность аннотировать типы данных. Аннотировать можно:

* тип, возвращаемый методом;



* Тип поля



* тип объекта по ссылке this в теле метода; можно явно указывать первым параметром метода ссылку this. Если не указывать, она передается неявно.



* приведение типов;



* уровни доступа к массиву;



* наследуемый класс;



* оператор new



* оператор throws;



* обобщенные типы, включая границы параметров и аргументы обобщенного типа.





**Повторяющиеся аннотации**

Чтобы создать повторяющуюся аннотацию, необходимо указать @Repeatable и в поле value указать тип контейнера для повторяющейся аннотации. Для доступа к повторяющимся аннотациям с помощью методов, следует воспользоваться контейнерной, а не самой повторяющейся аннотацией.

@RepAnn(name="main")  
@RepAnn(name="start")  
public class RepAnnMain {  
 public static void main(String[] args) {  
 *getAnn*(RepAnnMain.class);  
 }  
  
 // обработчик  
 public static void getAnn(Class<?> cls) {  
 if(cls.isAnnotationPresent(RepAnnContainer.class)) {  
 RepAnnContainer ann = cls.getAnnotation(RepAnnContainer.class);  
 System.*out*.println(ann);  
 RepAnn[] arr = ann.value();  
 for (RepAnn repAnn: arr){  
 System.*out*.println(repAnn.name());  
 }  
 }  
 }  
}  
  
// повторяющаяся аннотация  
@Retention(RetentionPolicy.*RUNTIME*)  
@Repeatable(RepAnnContainer.class)  
@interface RepAnn {  
 String name();  
}  
  
  
// контейнер  
@Retention(RetentionPolicy.*RUNTIME*)  
@interface RepAnnContainer {  
 RepAnn[] value();  
}

Так же для получения доступа можно воспользоваться методами:

* *getAnnotationsByType(Class<T> type)*
* *getDeclaredAnnotationsByType(Class<T> type)*

этим методы возвращают массив аннотаций указанного типа.

// обработчик  
public static void getAnn2(Class<?> cls) {  
 RepAnn[] arr = cls.getAnnotationsByType(RepAnn.class);  
 for (RepAnn repAnn: arr){  
 System.*out*.println(repAnn.name());  
 }  
}