Annotations

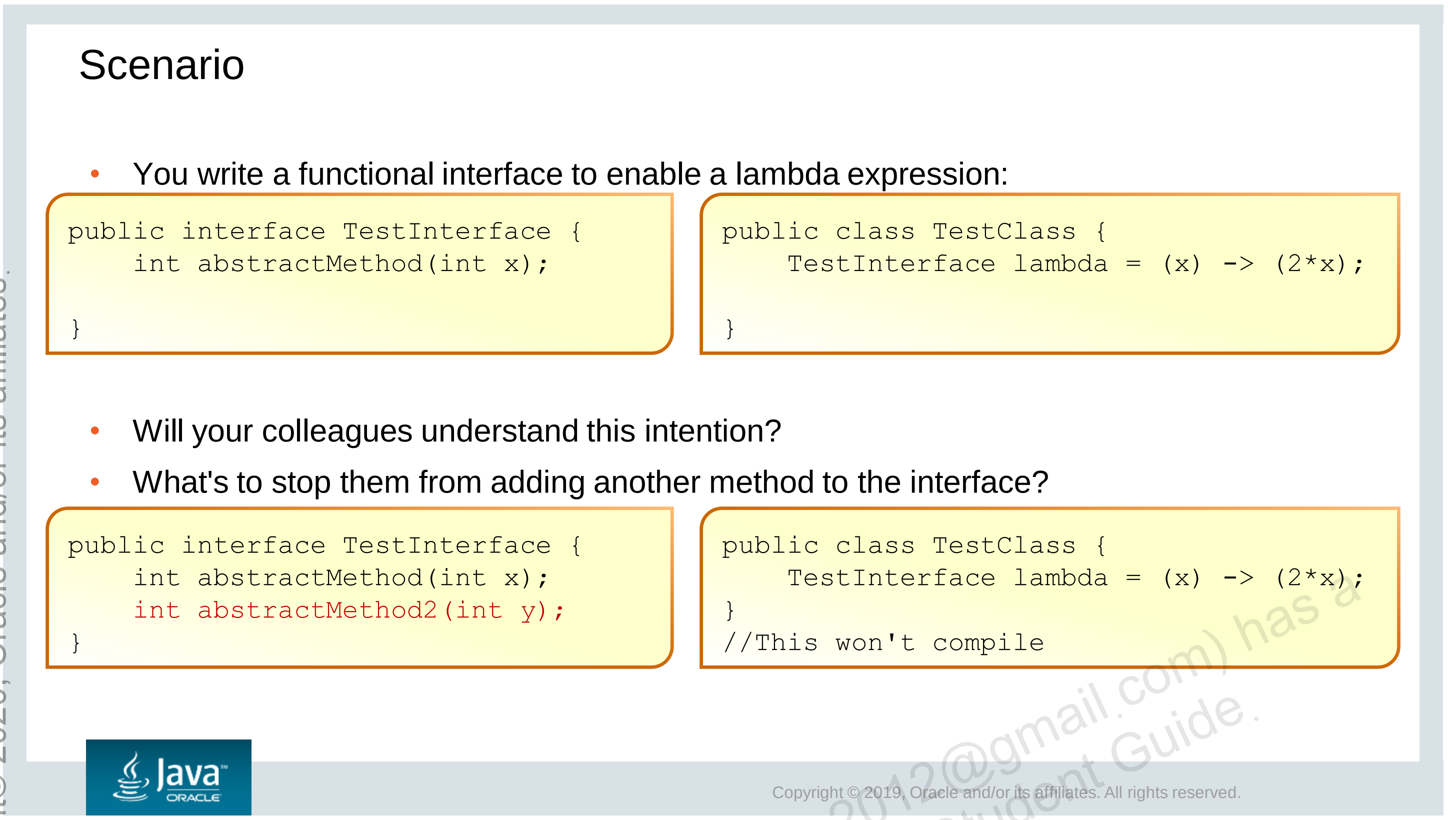
Objectives

After completing this lesson, you should be able to:

- Describe the purpose of annotations and typical usage patterns
- Apply annotations to classes and methods
- Describe commonly used annotations in the JDK
- Declare custom annotations

Topics

- Common Annotations in the JDK
- Meta Annotations
- Custom Annotations
- Annotations from Frameworks



Scenario

- You write a functional interface to enable a lambda expression:

```
public interface TestInterface {
    int abstractMethod(int x);

}
```

```
public class TestClass {
    TestInterface lambda = (x) -> (2*x);

}
```

- Will your colleagues understand this intention?
- What's to stop them from adding another method to the interface?

```
public interface TestInterface {
    int abstractMethod(int x);
    int abstractMethod2(int y);
}
```

```
public class TestClass {
    TestInterface lambda = (x) -> (2*x);
}
//This won't compile
```

A functional interface contains exactly one abstract method. The lambda expressions you write elsewhere are derived from a functional interface's one abstract method. Although an interface is still valid if it contains a second abstract method, it no longer works as a functional interface. A second abstract method ruins the functional interface and risks breaking code elsewhere. Your colleague may not realize this issue because the code breaks where they're not looking.

`@FunctionalInterface` Annotation

Solution: Insert this annotation above the interface declaration.

- This enforces the definition of a functional interface.
- A colleague's deviation is immediately flagged.

```
@FunctionalInterface
public interface TestInterface {
    int abstractMethod(int x);

}
```

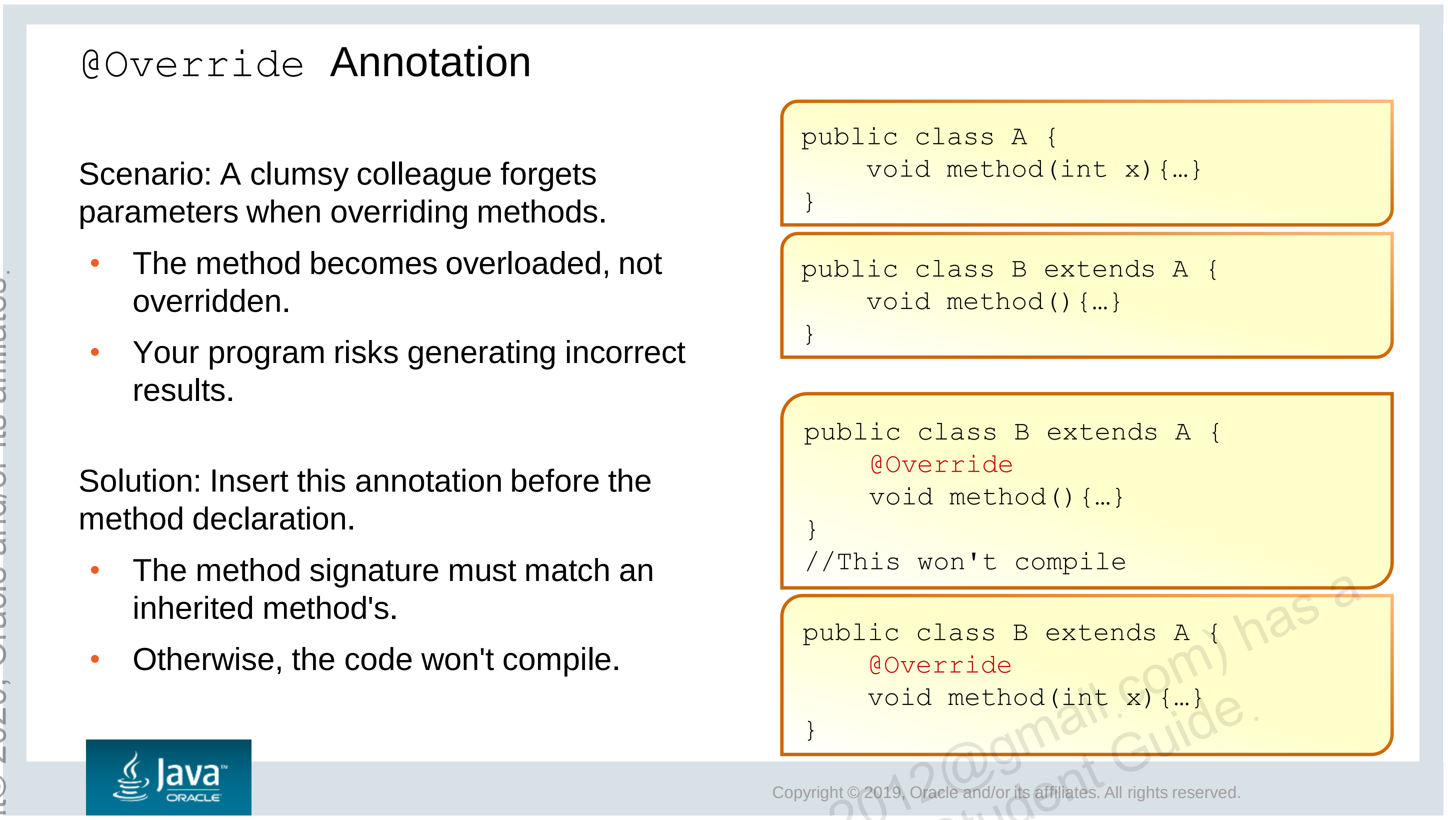
```
@FunctionalInterface
public interface TestInterface {
    int abstractMethod(int x);
    int abstractMethod2(int y);
}
//This won't compile
```

Insert the `@FunctionalInterface` annotation before the interface's definition. This forces the interface to abide by the rules of being a functional interface, which has exactly one abstract method. If a colleague unknowingly deviates by adding a second abstract method, the interface won't compile. The issue is immediately flagged in the very file your colleague is editing. Your colleague should immediately notice and fix the issue.

This code is available in the `functionalInterface` package of the `AnnotationTesting` project.

Annotation Characteristics

- Annotations start with the `@` symbol.
- Annotations are a form of metadata.
  - They provide data about a program that's not part of the program itself.
  - They have no direct effect on the operation of the code they annotate.
- The metadata is useful at the source-code level, compile time, or run time.
- Annotations are applicable to declarations of a class, method, field, the use of types, and to other annotations.



`@Override` Annotation

Scenario: A clumsy colleague forgets parameters when overriding methods.

- The method becomes overloaded, not overridden.
- Your program risks generating incorrect results.

Solution: Insert this annotation before the method declaration.

- The method signature must match an inherited method's.
- Otherwise, the code won't compile.

```
public class A {
    void method(int x){…}
}
```

```
public class B extends A {
    void method(){…}
}
```

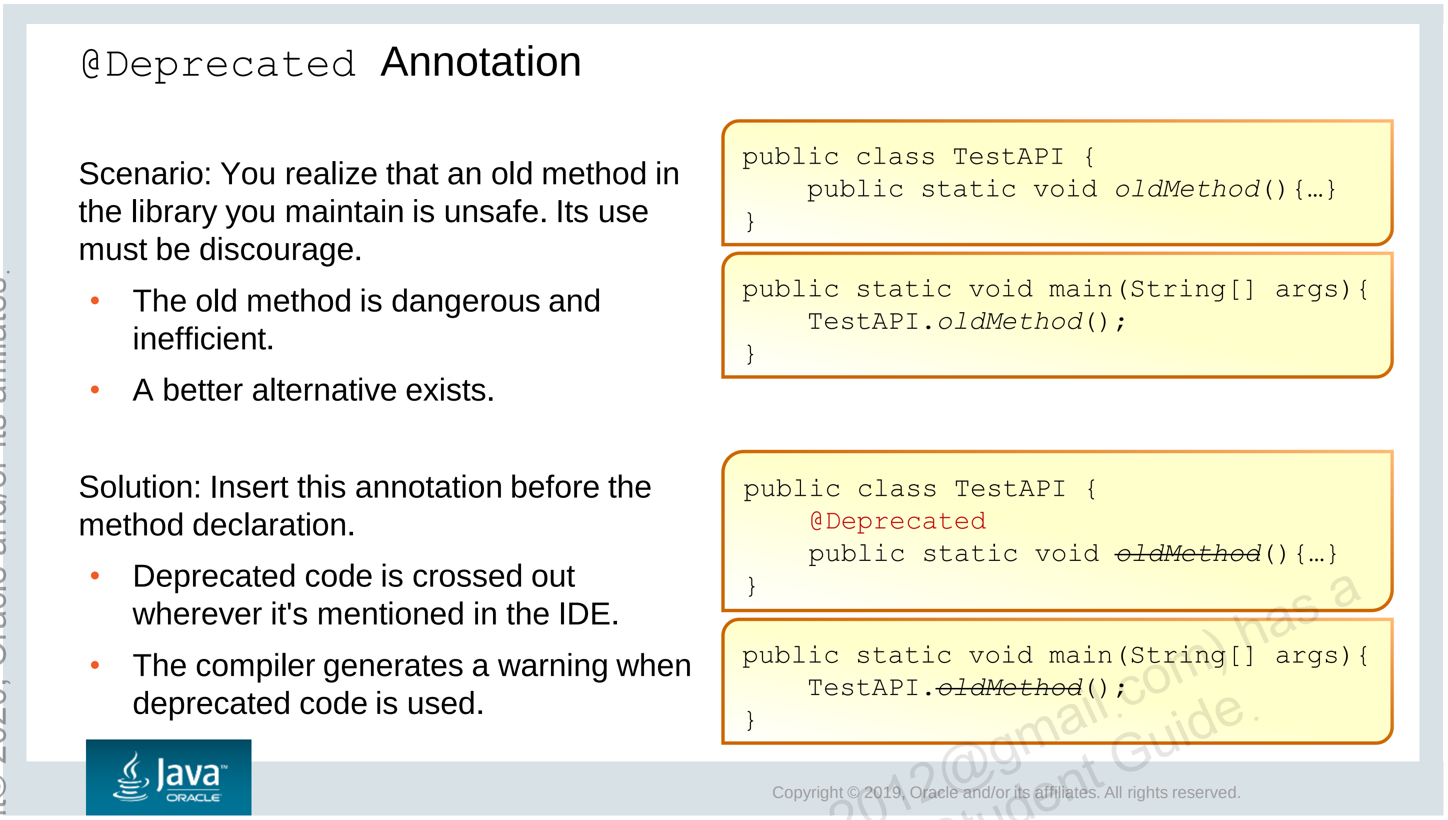
```
public class B extends A {
    @Override
    void method(){…}
}
//This won't compile
```

```
public class B extends A {
    @Override
    void method(int x){…}
}
```

As method signatures grow complex, it's increasingly likely you may forget a parameter when attempting to override a method. The slide shows a simplified version of this scenario. Nevertheless, the effects of this mistake cause the method to be overloaded, not overridden. The program may generate incorrect results, as a method call may run the superclass's implementation rather than the subclass implementation you anticipated. You'll encounter similar issues if you misspell the method name.

The `@Overload` annotation guards against this scenario. The compiler generates an error if the annotated method fails to correctly override an inherited method.

This code is available in the `override` package of the `AnnotationTesting` project.



`@Deprecated` Annotation

Scenario: You realize that an old method in the library you maintain is unsafe. Its use must be discourage.

- The old method is dangerous and inefficient.
- A better alternative exists.

```
public class TestAPI {
    public static void oldMethod(){…}
}
```

```
public static void main(String[] args){
    TestAPI.oldMethod();
}
```

Solution: Insert this annotation before the method declaration.

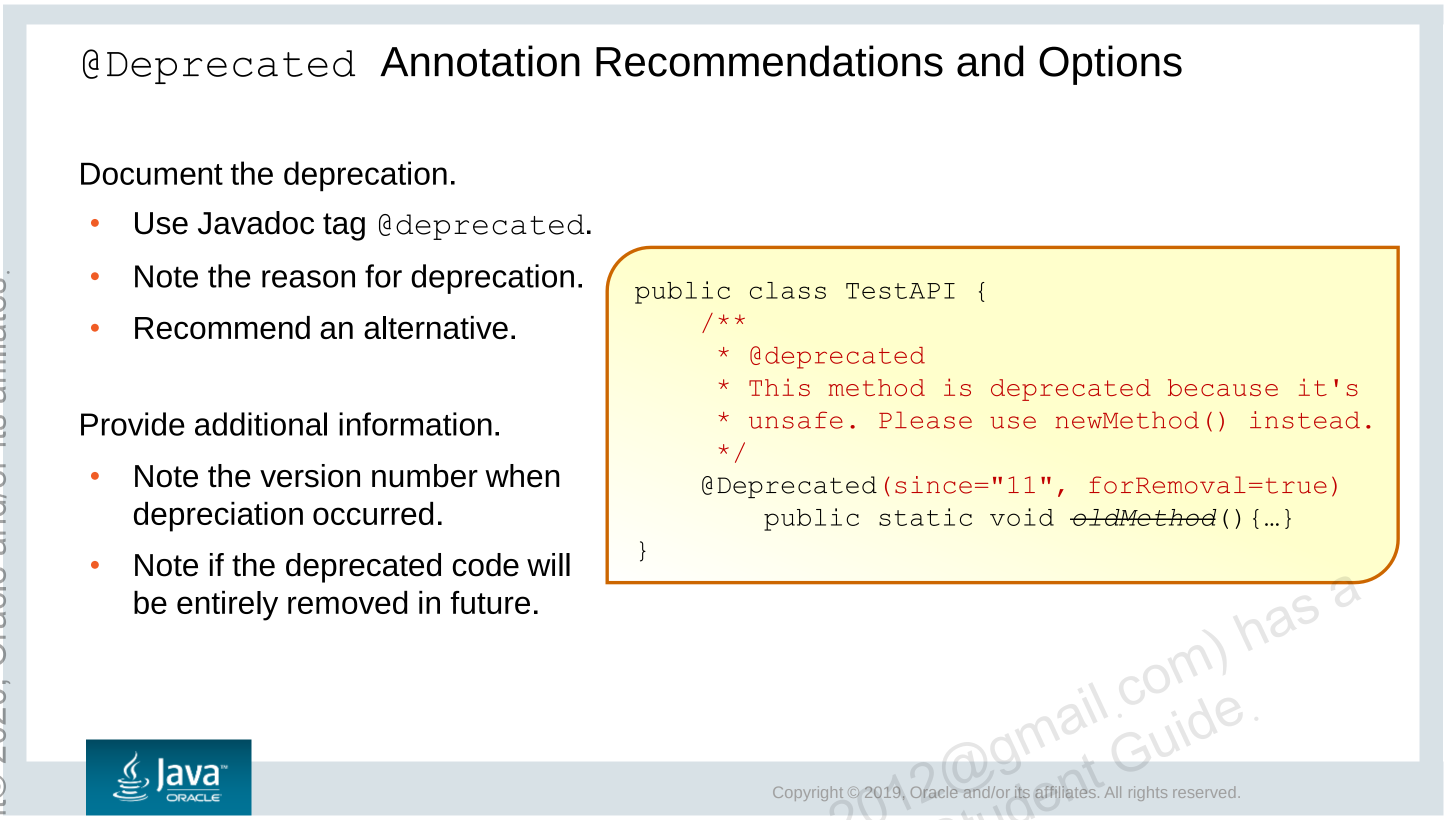
- Deprecated code is crossed out wherever it's mentioned in the IDE.
- The compiler generates a warning when deprecated code is used.

```
public class TestAPI {
    @Deprecated
    public static void ~~oldMethod~~(){…}
}
```

```
public static void main(String[] args){
    TestAPI.~~oldMethod~~();
}
```

The `@Deprecated` annotation discourages code use. This annotation may be applied to constructors, fields, local variables, methods, packages, module, parameters, classes, interfaces, and enums. Code may be deprecated for several reasons, for example, its usage likely leads to errors; it may be changed incompatibly or removed in a future version; it's been superseded by a more-preferable alternative; or it's obsolete. The compiler generates a warning when deprecated code is used or overridden in non-deprecated code.

This code is available in the `deprecated` package of the `AnnotationTesting` project.



@Deprecated Annotation Recommendations and Options

Document the deprecation.

- Use Javadoc tag `@deprecated`.
- Note the reason for deprecation.
- Recommend an alternative.

Provide additional information.

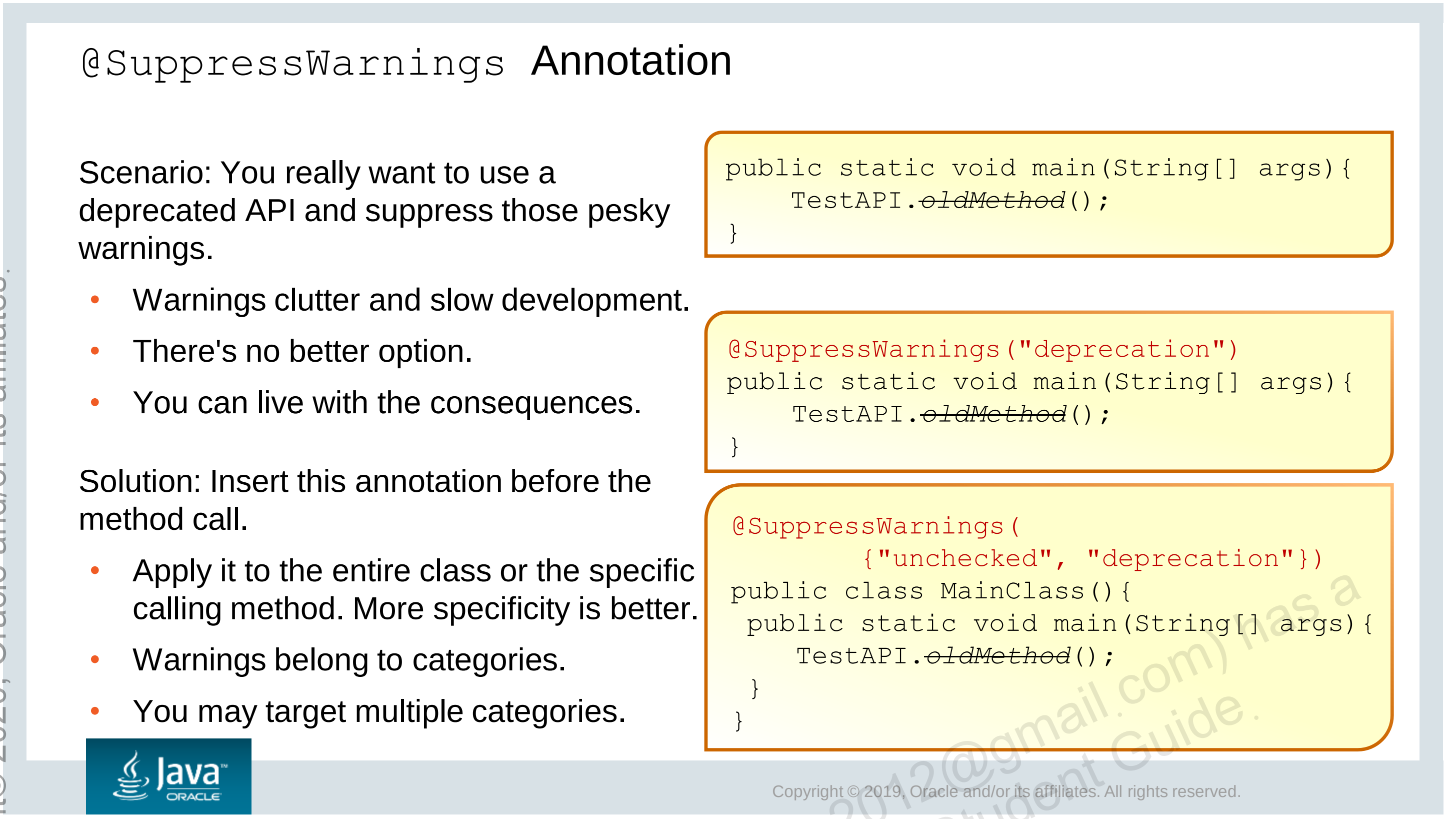
- Note the version number when depreciation occurred.
- Note if the deprecated code will be entirely removed in future.

```
public class TestAPI {
    /**
     * @deprecated
     * This method is deprecated because it's
     * unsafe. Please use newMethod() instead.
     */
    @Deprecated(since="11", forRemoval=true)
        public static void ~~oldMethod~~(){…}
}
```

It's strongly recommended that you document the reason for deprecating a program element using the `@deprecated` Javadoc tag. Documentation should also suggest and link to any recommended replacement API. A replacement API often has subtly different semantics, which should also be noted. The use of the at sign (@) in both Javadoc comments and annotations is not coincidental: they are related conceptually. Also, note the Javadoc tag starts with a lowercase d and the annotation starts with an uppercase D.

`@Deprecated` has a String element `since`. Its value indicates the version when deprecation first occurred. In this case, `oldMethod` was deprecated in version 11. It's recommended you specify a `since` value for any newly deprecated program elements.

`@Deprecated` has a boolean element `forRemoval`. A `true` value indicates intent to remove the deprecated element in a future version. A `false` value indicates that although use of the deprecated element is discouraged, there's no intent to remove it yet.



@SuppressWarnings Annotation

Scenario: You really want to use a deprecated API and suppress those pesky warnings.

- Warnings clutter and slow development.
- There's no better option.
- You can live with the consequences.

Solution: Insert this annotation before the method call.

- Apply it to the entire class or the specific calling method. More specificity is better.
- Warnings belong to categories.
- You may target multiple categories.

```
public static void main(String[] args){
    TestAPI.~~oldMethod~~();
}
```

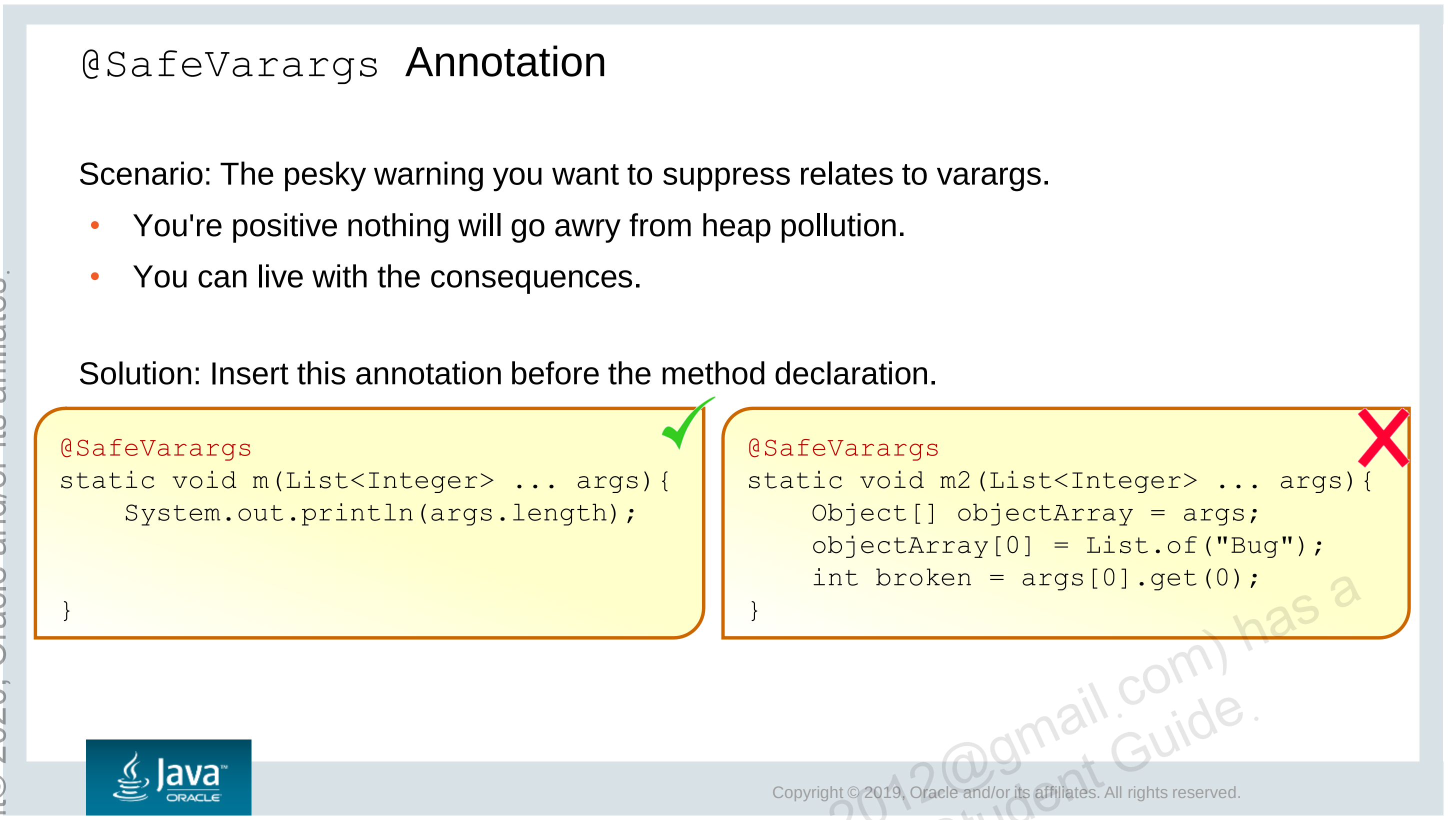
```
@SuppressWarnings("deprecation")
public static void main(String[] args){
    TestAPI.~~oldMethod~~();
}
```

```
@SuppressWarnings(
        {"unchecked", "deprecation"})
public class MainClass(){
 public static void main(String[] args){
    TestAPI.~~oldMethod~~();
  }
}
```

The `@SuppressWarnings` annotation indicates which categories of compiler warnings should be suppressed. This annotation may be applied to constructors, fields, local variables, methods, module, parameters, classes, interfaces, and enums.

The set of warnings suppressed in a given code element is a superset of the warnings suppressed in all containing elements. For example, if you annotate a class to suppress one warning and annotate a method to suppress another, both warnings will be suppressed in the method. As a matter of style, always use this annotation on the most deeply nested element where it is effective. If you want to suppress a warning in a particular method, you should annotate that method rather than its class.

Every compiler warning belongs to a category. The Java Language Specification lists two categories: deprecation and unchecked. The unchecked warning can occur when interfacing with legacy code written before the advent of generics.



`@SafeVarargs` Annotation

Scenario: The pesky warning you want to suppress relates to varargs.

- You're positive nothing will go awry from heap pollution.
- You can live with the consequences.

Solution: Insert this annotation before the method declaration.

✓

```
@SafeVarargs
static void m(List<Integer> ... args){
    System.out.println(args.length);


}
```

✗

```
@SafeVarargs
static void m2(List<Integer> ... args){
    Object[] objectArray = args;
    objectArray[0] = List.of("Bug");
    int broken = args[0].get(0);
}
```

The `@SafeVarargs` annotation is applicable to methods and constructors. It asserts your code doesn't perform potentially unsafe operations on its varargs parameter. These unsafe operations may result in heap pollution. Heap pollution occurs when a variable of a parameterized type refers to an object that is not of that parameterized type. Because this is difficult for the compiler to verify, a warning is issued instead. Your code may be perfectly safe and free of heap pollution issues. If this is true, the compiler warnings may feel like annoying clutter. The `@SafeVarargs` annotation lets you suppress these unchecked warnings.

Be very sure of your assertion before using this annotation. The example on the right misuses `@SafeVarargs`. This code results in a `ClassCastException`. The reason has to do with how parameterized varargs are handled. When the compiler encounters a varargs method, it translates the varargs parameter into an array. In this example, the compiler translates the varargs parameter `List<Integer>... args` to an array parameter `List<Integer>[] args`. However, Java does not allow arrays with parameterized types. Parametrized information may be stripped away at compile time thanks to type erasure, resulting in an array of `List`s with `Object`s. The example creates an `Object` array that points to the same location in memory as the `List<Integer>[] args`. This provides a means of assigning a `String` value to the array where an `Integer` should be.

This code is available in the `safeVarargs` package of the `AnnotationTesting` project.

Topics

- Common Annotations in the JDK
- Meta Annotations
- Custom Annotations
- Annotations from Frameworks

Examples from the `Deprecated` Annotation

- Meta annotations are applied to other annotations.
- `@Documented` specifies the annotation information appear in Javadoc-generated documentation.
- `@Retention` specifies where the annotation and its affects are retained.
- `@Target` specifies where the annotation is applicable.
- To supply more than one value, surround the list of values with curly braces.

```
import java.lang.annotation.*;
import static java.lang.annotation.ElementType.*;

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({CONSTRUCTOR, FIELD, LOCAL_VARIABLE,
         METHOD, PACKAGE, MODULE,
         PARAMETER, TYPE})
public @interface Deprecated {
    String since() default "";
    boolean forRemoval() default false;
}
```

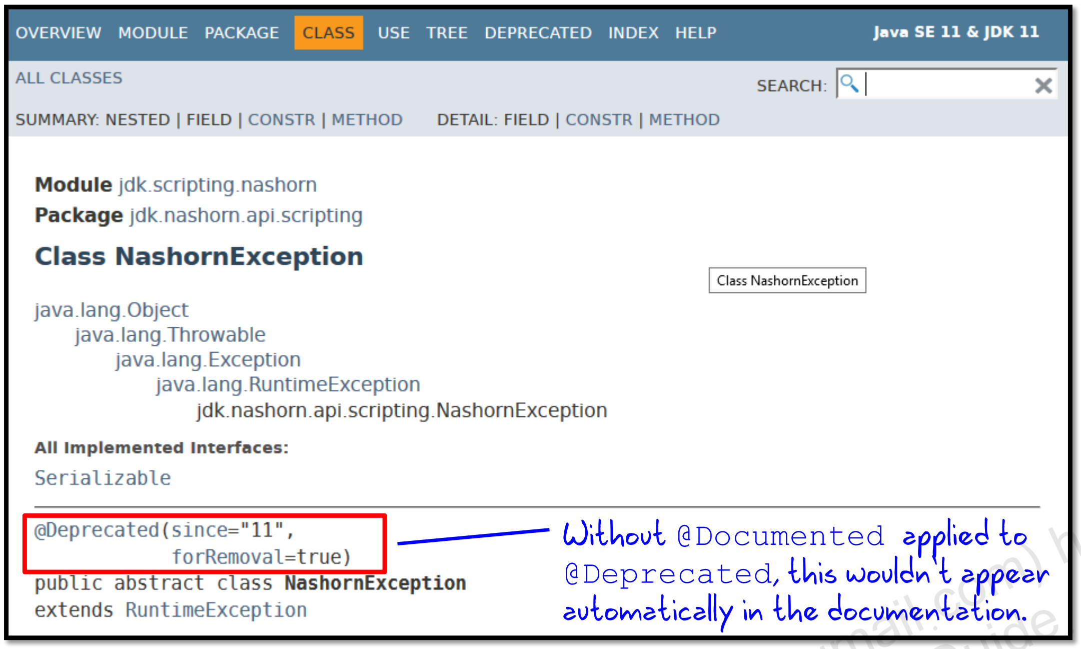
`@Retention` accepts `RetentionPolicy` enum values.

- `RetentionPolicy.SOURCE` is retained in the source code, but discarded by the compiler.
- `RetentionPolicy.CLASS` is retained by the compiler, but ignored by the JVM.
- `RetentionPolicy.RUNTIME` is retained by the JVM and readable at run time.

In this example, `@Target` specifies where the `@Deprecated` annotation can be applied. `@Target` accepts `Element Type` enum values. There is an example in the `targetType` package of the `AnnotationTesting` project.

- `ElementType.ANNOTATION_TYPE` can be applied to an annotation type.
- `ElementType.CONSTRUCTOR` can be applied to a constructor.
- `ElementType.FIELD` can be applied to a field including enum constants.
- `ElementType.LOCAL_VARIABLE` can be applied to a local variable.
- `ElementType.METHOD` can be applied to a method-level annotation.
- `ElementType.MODULE` can be applied to a module.
- `ElementType.PACKAGE` can be applied to a package declaration.
- `ElementType.PARAMETER` can be applied to the parameters of a method.
- `ElementType.TYPE` can be applied to a class, interface, annotation, or enum declaration.
- `ElementType.TYPE_PARAMETER` can be applied to a type parameter declaration.
- `ElementType.TYPE_USE` can be applied to the use of a type.

`@Documented` Meta Annotation Effects



OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
Java SE 11 & JDK 11
ALL CLASSES
SEARCH:
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD
Class NashornException

Module jdk.scripting.nashorn
Package jdk.nashorn.api.scripting

Class NashornException

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                jdk.nashorn.api.scripting.NashornException
```

All Implemented Interfaces:

`Serializable`

```
@Deprecated(since="11",
            forRemoval=true)
public abstract class NashornException
extends RuntimeException
```

Without `@Documented` applied to `@Deprecated`, this wouldn't appear automatically in the documentation.

Topics

- Common Annotations in the JDK
- Meta Annotations
- Custom Annotations
- Annotations from Frameworks

Scenario

- Company guidelines require you to cite authorship information at the beginning of each class.
- This has been traditionally done through comments.
- Colleagues make mistakes:
  - Forgotten categories
  - Misspelt categories
  - Commenting in different locations
  - Different data formats

```
// Author: Maureen DeLawn
// Revision: 6
// Revision Date: 4/15/2019
// Reviewers: Ben Ng, Yu Wong

public class Gen1List{

}
```

```
public class Gen2List extends Gen1List{
    // Author: Yu Wong
    // Version: Alpha
    // Reverion Date: 4/15/2019

}
```

The second examples differs from the first in that:

- **Forgotten categories:** Reviewers is missing
- **Misspelt categories:** Revision is spelt as Reverion
- **Commenting in different locations:** Comments occur after the class declaration rather than before
- **Different data formats:** Alpha is used instead of a number

Solution: Write a Custom Annotation

- Annotations enforce structure:
  - What data to include
  - Where it can be written
- Create a file for your annotation.
- Declare its type as `@interface`. Annotations are an interface variant.
- Declare any annotation elements.
  - Elements store values.
  - An element declaration looks similar to a method declaration.
  - You may set default values.
  - You may declare an element as an array to contain many values.

ClassPreamble.java

```
@Retention(RetentionPolicy.RUNTIME)
@Target(TYPE)
public @interface ClassPreamble {
    String author();
    int revision() default 0;
    String revisionDate() default "N/A";
    String[] reviewers();
}
```

You can have your IDE create a Java annotations file like you would stub any class or interface. The type is declared as `@interface`. In fact, annotations are an interface form. The symbol @ is a play on words. Pronounced "AT", it's an acronym for "Annotation Type".

The `@Retention` and `@Target` meta annotations are optional. `@Target` ensures the `ClassPreamble` annotation is only applicable to classes, interfaces, and enums.

There are four annotation elements declared in this example: author, revision, revisionDate, and reviewers. Element declaration looks similar to method declaration. You can declare elements to contain different types of data, including Strings, ints, and arrays. You may also set default values for a elements.

This code is available in the `customAnnotation` package of the `AnnotationTesting` project.

Applying a Custom Annotation

- Apply your annotation using the `@` symbol.
- List any element-value pairs between parentheses, separated by commas.
- If not specified, elements take on their default value.
- You must set a value for all elements that lack a default value.
- Values must match their declared type:
  - Good: `revision = 6`
  - Bad: `revision = "Alpha"`
- Array values require curly braces.

```
@ClassPreamble(
    author = "Maureen DeLawn",
    revision = 6,
    revisionDate = "4/15/2019",
    reviewers = {"Ben Ng", "Yu Wong"}
)
public class Gen1List {}
```

```
@ClassPreamble(
    author = "Yu Wong",
    reviewers = {"Ben Ng", "Robin Peck"}
)
public class Gen2List extends Gen1List{}
```

`@ClassPreamble` can now be written at the top of classes. This is followed by a set of parentheses where you specify the values of each annotation element. List any element-value pairs, separated by commas. Make sure your values match the expected type. An array annotation element may have more than one value. Use curly braces to supply multiple values.

The second example shows how you do not need to specify an element's value if the annotation provides a default value. You must specify element values that have no default value. The program won't compile otherwise.

Reading Annotation Elements Through Reflection

```
public static void main(String[] args) {
  System.out.println(
       Gen1List.class.getAnnotation(ClassPreamble.class)           +"\n\n"
        +Gen2List.class.getAnnotation(ClassPreamble.class)         +"\n\n"
        +Gen2List.class.getAnnotation(ClassPreamble.class).author());
}
```

```
@customAnnotation.ClassPreamble(revisionDate="4/15/2019", revision=6,
author="Maureen DeLawn", reviewers={"Ben Ng", "Yu Wong"})

@customAnnotation.ClassPreamble(revisionDate="N/A", revision=0, author="Yu Wong",
reviewers={"Ben Ng", "Robin Peck"})

Yu Wong
```

Annotation element values are accessible through reflection. This example shows how you can reveal all an annotation's elements, or simply one specific element.

Note: `@ClassPreamble` requires the meta annotation `@Retention(RetentionPolicy.RUNTIME)` to enable this access. Otherwise null is returned.

Other Details

- You may omit an element's name when assigning a value if it's named `value`.

```
public @interface ClassPreamble {
    String value();
}
```

```
@ClassPreamble(value = "Ben Ng")
```
✓

```
@ClassPreamble("Ben Ng")
```
✓

- You may omit curly braces if an array contains only one value.

```
public @interface ClassPreamble {
    String[] value();
}
```

```
@ClassPreamble({"Ben Ng"})
```
✓

```
@ClassPreamble("Ben Ng")
```
✓

- Beware of null values.

```
public @interface ClassPreamble {
    String value() default null;
}
```
✗

```
@ClassPreamble(null)
```
✗

- Although annotations are an interface form, they cannot extend other interfaces.

```
public @interface ClassPreamble extends List {}
```
✗

An element named `value` gets special treatment. If you name an element `value`, you can omit its name later when you apply the annotation and assign its value.

When assigning a value to an array element, you may omit the curly braces if the array contains only one value.

Beware of null values. These examples won't compile.

Although annotations are an interface form, they cannot extend other interfaces.

Inheriting an Annotation

Scenario: Management believes that because superclass edits impact subclasses, authorship data from superclasses should also be reflected on subclasses.

Solution: Add the `@Inherited` meta annotation.

- A subclass without an annotation gains the inherited annotations from a superclass.
- A subclass cannot inherit an annotation from an interface.

```
@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target(TYPE)
public @interface ClassPreamble {
    String author();
    int revision() default 0;
    String revisionDate() default "N/A";
    String[] reviewers();
}
```

```
@ClassPreamble(
    author = "Maureen DeLawn",
    revision = 6,
    revisionDate = "4/15/2019",
    reviewers = {"Ben Ng", "Yu Wong"}
)
public class Gen1List {}
```

```
public class Gen2List extends Gen1List{}
```

The meta annotation `@Inherited` is added to the annotation type `ClassPreamble`. The `Gen1List` class is annotated with `@ClassPreamble`. `Gen2List` extends `Gen1List`, but contains no annotation. Because `@ClassPreamble` is now an inherited annotation, `Gen2List` behaves as if it were annotated with the same `@ClassPreamble` found in the superclass. We'll examine the impacts of this on the next slide.

Reading an Inheriting an Annotation Through Reflection

```
public static void main(String[] args) {
  System.out.println(
        Gen1List.class.getAnnotation(ClassPreamble.class)          +"\n\n"
         +Gen2List.class.getAnnotation(ClassPreamble.class)        +"\n\n"
         +Gen2List.class.getAnnotation(ClassPreamble.class).author());
}
```

```
@customAnnotation.ClassPreamble(revisionDate="4/15/2019", revision=6,
author="Maureen DeLawn", reviewers={"Ben Ng", "Yu Wong"})

@customAnnotation.ClassPreamble(revisionDate="4/15/2019", revision=6,
author="Maureen DeLawn", reviewers={"Ben Ng", "Yu Wong"})

Maureen DeLawn
```

When you query the annotation type for a class that lacks the annotation, the class' superclass is then queried for the annotation type. The `Gen2List` class lacks `@ClassPreamble`. When that annotation type is queried through reflection, the information is gathered by pulling from the superclass' `@ClassPreamble`.

Repeating an Annotation

Scenario: Management wants authorship data available for each edit.

Solution: Store repeated annotations in a single compiler-generated container.

- Add the `@Repeatable` meta annotation.
  - In parentheses, specify a container.
- Declare the container annotation type.
  - Add an element that's an array of the repeated annotation type.
  - Name this element `value`.

```
@Repeatable(PreamblesContainer.class)
@Retention(RetentionPolicy.RUNTIME)
@Target(TYPE)
public @interface ClassPreamble {
    String author();
    int revision() default 0;
    String revisionDate() default "N/A";
    String[] reviewers();
}
```

```
@Retention(RetentionPolicy.RUNTIME)
@Target(TYPE)
public @interface PreamblesContainer {
    ClassPreamble[] value();
}
```

Repeating annotations are stored in a container annotation that's automatically generated by the Java compiler. For the compiler to do this, two declarations are required in your code.

First, mark your annotation with the meta annotation `@Repeatable`. In parentheses is the container annotation type that the Java compiler generates to store repeating annotations. In this example, the container is `PreamblesContainer`.

Second, declare the container annotation type. It must have an element that's an array type of your repeatable annotation. In this example, `ClassPreamble[]`. Name this element `value`.

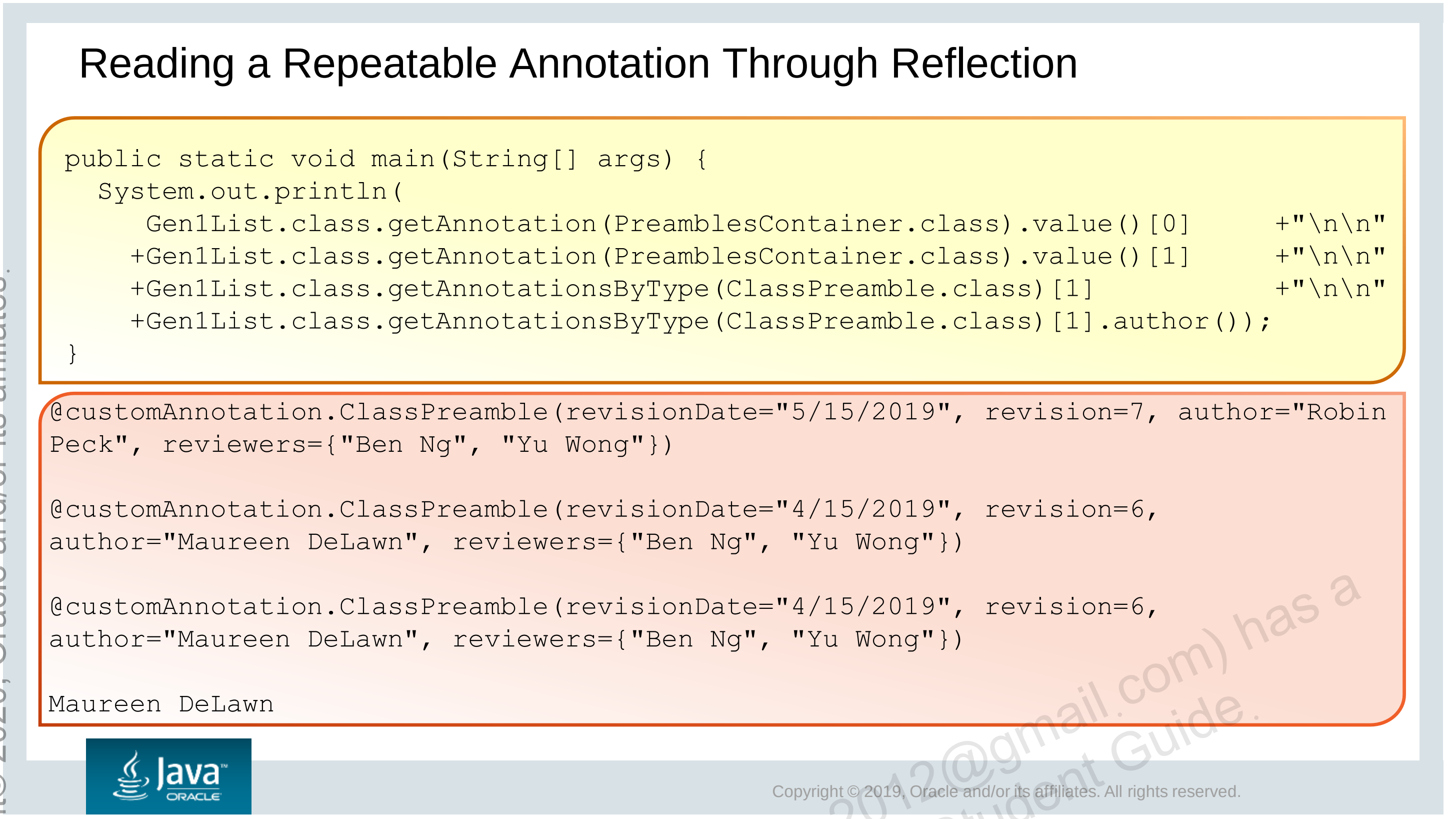
Repeating Annotation Example

```
@ClassPreamble(
    author = "Robin Peck",
    revision = 7,
    revisionDate = "5/15/2019",
    reviewers = {"Ben Ng", "Yu Wong"}
)
@ClassPreamble(
    author = "Maureen DeLawn",
    revision = 6,
    revisionDate = "4/15/2019",
    reviewers = {"Ben Ng", "Yu Wong"}
)
public class Gen1List {}
```

```
@Repeatable(PreamblesContainer.class)
@Retention(RetentionPolicy.RUNTIME)
@Target(TYPE)
public @interface ClassPreamble {
    String author();
    int revision() default 0;
    String revisionDate() default "N/A";
    String[] reviewers();
}
```

```
@Retention(RetentionPolicy.RUNTIME)
@Target(TYPE)
public @interface PreamblesContainer {
    ClassPreamble[] value();
}
```

`@ClassPreamble` can be repeated as many times as you need on the same class.



Reading a Repeatable Annotation Through Reflection

```
public static void main(String[] args) {
  System.out.println(
     Gen1List.class.getAnnotation(PreamblesContainer.class).value()[0]     +"\n\n"
    +Gen1List.class.getAnnotation(PreamblesContainer.class).value()[1]     +"\n\n"
    +Gen1List.class.getAnnotationsByType(ClassPreamble.class)[1]           +"\n\n"
    +Gen1List.class.getAnnotationsByType(ClassPreamble.class)[1].author());
}
```

```
@customAnnotation.ClassPreamble(revisionDate="5/15/2019", revision=7, author="Robin
Peck", reviewers={"Ben Ng", "Yu Wong"})

@customAnnotation.ClassPreamble(revisionDate="4/15/2019", revision=6,
author="Maureen DeLawn", reviewers={"Ben Ng", "Yu Wong"})

@customAnnotation.ClassPreamble(revisionDate="4/15/2019", revision=6,
author="Maureen DeLawn", reviewers={"Ben Ng", "Yu Wong"})

Maureen DeLawn
```

Repeatable annotations were introduced in Java SE 8, along with a few enhancements to the reflection API to handle them. This slide shows two methods of accessing information from repeated annotations: `getAnnotation()`, which existed previously, and `getAnnotationsByType()`. The `getAnnotation()` method must search for the container. The `value()` method is called to return an array of all `ClassPreamble` annotations within the container.

The `getAnnotationsByType()` method can search for the specific annotation type rather than the container. It also returns an array of the annotation type you're searching for.

Topics

- Common Annotations in the JDK
- Meta Annotations
- Custom Annotations
- Annotations from Frameworks

Frameworks

- There are many Java frameworks.
- Annotations may act as the interface between Java code and Java frameworks.
- You'll often use annotations to leverage framework features.

Popular Java frameworks include:

- Spring Boot
- Spring MVC
- JUnit
- Struts
- JavaServer Faces (JSF)
- Dropwizard

`@NonNull` Type Annotation and Pluggable Type Systems

- It's a non-standardized annotation.
- It's found through various frameworks and tools.
- It's a type annotation, which are applicable anywhere you'd use or declare a type.
- Use it in conjunction with defect detection tools to warn or protect against possible null values.

```
public class TestClass {
    @NonNull String str;
    BiConsumer lambda = (@NonNull var x, final var y) -> x.process(y);
}
```

One example of an annotation found in frameworks is the type annotation `@NonNull`. Type annotations were created to support improved analysis of Java programs by ensuring stronger type checking. Types include class instance creation expressions (new), casts, implements clauses, throws clauses, fields, and local variables. As of Java SE 11, this also includes lambda parameters.

You may want to ensure a particular variable in your program is never assigned to `null` to avoid triggering a `NullPointerException`. You can write or find a custom plug-in to check for this. You would then modify your code to annotate that particular variable. With the judicious use of type annotations and the presence of pluggable type checkers, you can write code that is stronger and less prone to error.

You might want to take advantage of the Checker Framework created by the University of Washington. This framework includes a NonNull module, as well as a regular expression module, and a mutex lock module. For more information, see http://types.cs.washington.edu/checker-framework/.

Summary

In this lesson, you should have learned how to:

- Describe the purpose of annotations and typical usage patterns
- Apply annotations to classes and methods
- Describe commonly used annotations in the JDK
- Declare custom annotations