Annotations, also called meta-data, are:
- structured information that describes, explains, locates or otherwise makes easier to retrieve, use or manage an information resource.

Annotations are information about information that are interpreted at some point by code or data analysis tools.

Annotations can be used to:
- document the code;
- extract some specific data from the program (e.g., lint, metrics, and so on);
- automatically generate configuration files;
- ...

Do annotations facility introduce a new concept in Java? No, many applications require or produce files associated with a specific class,
- JAX-RPC, XML-RPC, SOAP, . . . ;
- EJB, JavaBean “BeanInfo” class.
- . . .

Creating and maintaining these associated files is painful.
Moreover many API need the use of special “placemakers”
- Remote, Serializable, Cloneable, . . . ;

This feature adds a customizable mechanism for annotating classes, methods and fields.
Java Annotations.

An Example: Javadoc.

Javadoc represents the main example of annotation use.

```java
public class Hello {
    /** it greets the «name» passed as an argument *
     * @param name the name to greet. */
    public void hello(String name) {
        System.out.println("Hello " + name);
    }
}
```

From this, we get its documentation, through the command:

```
[18:16] cazzola@ulik:tsp-javadoc>Hello.java
Loading source file Hello.java...
Constructing Javadoc information...
Standard Doclet version 1.5.0
Building tree for all the packages and classes...
Generating Hello.html...
```

Standard annotation types are those provided "out-of-the-box".

- `@Override`, to mark that a method overrides another method in its superclass;
- `@Deprecated`, to indicate that the use of this method is discouraged; and
- `@ SuppressWarning`, to turn off compiler warning for classes, methods, or field and variable initializers.

```java
@Override
public String toString() {
    return super.toString() + "[modified by subclass]";
}
```

Categories of Custom Annotations.

There are three categories of custom annotations:

- **Marker Annotations**, these are annotations without parameters or that uses default values for all parameters.

```java
@MarkerAnnotation
```

- **Single-Value Annotations**, the annotations of this kind have just a single member named `value`.

```java
@SingleValueAnnotation("some value");
```

- **Full Annotations**, these annotations exploits the full range of the annotation syntax.

```java
@Reviews({ // curly braces indicate an array of values
    @Review(grade=Review.Grade.EXCELLENT, reviewer="DF"),
    @Review(grade=Review.Grade.UNSATISFACTORY, reviewer="EG",
        comment="This method needs and @Override annotation.")
})
```

Annotation types are, basically, Java interfaces.

- They look similar to a normal Java interface definition, but you use the `@interface` which tells to the compiler that you are writing an annotation type.

```java
public @interface TODO {
    String value();
}
```

Annotations' members are defined as abstract methods (without body).

```java
public @interface TODO {
    String value();
}
```

You are actually defining a method named `value()`, and the compiler then automatically creates a member variable with the same name.
In the same way, it is possible to define annotation types with several members.

```java
public @interface GroupTODO {
    public enum Severity {CRITICAL, IMPORTANT, TRIVIAL, DOCUMENTATION;}
    Severity severity() default Severity.IMPORTANT;
    String item();
    String assignedTo();
}
```

by omitting severity the default value would be used.

The `java.lang.reflect` package has several additions that allows to introspect on the annotations.

The `java.lang.reflect` package has several additions that allows to introspect on the annotations.

To Be or Not to Be ... Annotated?

The easiest way to check for an annotation is by using the `isAnnotationPresent()` method.

```java
public void testAnnotationPresent() {
    Class c = Test.class;
    boolean todo = c.isAnnotationPresent(TODO.class);
    if (todo) System.out.println("The Test class is not completely implemented yet");
    else System.out.println("The Test class is completely implemented");
}
```

During the execution of the ReflectionTester class we get one of the two messages.

The `meta-annotations` are annotations on annotations.

There are four kind of predefined meta-annotations:

- `@Target`, specifies which program elements (types, methods, ... ) can have annotations of the defined type.
- `@Retention`, indicates whether an annotation is tossed out by the compiler or retained in the compiled class file.
- `@Documented`, indicates that the annotation should be considered as part of the public API of the annotated program element.
- `@Inherited`, is used on annotation types targeted at classes and indicates that the annotated type is an inherited one.

If you are not checking a marker annotation you can reify the annotation and get the values of its members.

```java
public void testGetAnnotation() throws NoSuchMethodException {
    Class c = Test.class;
    Method element = c.getMethod("calculateInterest", float.class, float.class);
    GroupTODO groupTodo = element.getAnnotation(GroupTODO.class);
    String assignedTo = groupTodo.assignedTo();
    System.out.println("TODO item on Test is assigned to: "+assignedTo.".");
}
```

During the execution we get the members' values.

```
[14:47] cazzola@ulik:tsp> java ReflectionTester
TODO item on Test is assigned to: 'Walter Cazzola'.
```

Look at the Annotation class and at the AnnotatedElement interface to know the whole reflective API.
- Brett McLaughlin and David Flanagan.  
  *Java 1.5 Tiger: A Developer’s Notebook.*  
  O'Reilly, June 2004.

- Walter Cazzola, Antonio Cisternino, and Diego Colombo.  
  *Freely Annotating C#.*  