Design patterns

Avancerad Javaprogrammering

Utbildare: Mikael Olsson

Dagens områden

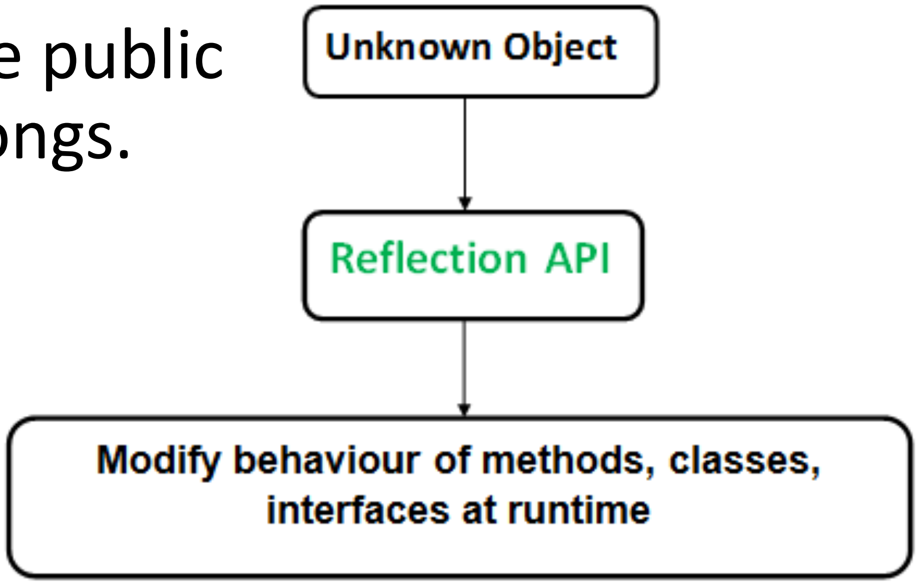
- Reflection
- Annotations
- Design patterns

Reflection

- Allows us to inspect or/and modify runtime attributes of classes, interfaces, fields, and methods.
- Additionally, we can instantiate new objects, invoke methods, and get or set field values using reflection.
- `import java.lang.reflect.*;`
- To get access to the class, method, and field information of an instance we call the `getClass` method which returns the runtime class representation of the object.
- https://www.geeksforgeeks.org/reflection-in-java/
- https://docs.oracle.com/javase/tutorial/reflect/index.html

Reflection – simple example

- The `getClass()` method is used to get the name of the class to which an object belongs.
- The `getConstructors()` method is used to get the public constructors of the class to which an object belongs.
- The `getMethods()` method is used to get the public methods of the class to which an objects belongs.
- code/15 - reflection/simple/Test.java



Reflection – observation 1

- We can invoke an method through reflection if we know its name and parameter types. We use below two methods for this purpose
  - `getDeclaredMethod()` : To create an object of method to be invoked. The syntax for this method is
    - `Class.getDeclaredMethod(name, parametertype)`
      - *name* - the name of method whose object is to be created
      - *parametertype* - parameter is an array of Class objects
  - `invoke()` : To invoke a method of the class at runtime we use following method
    - `Method.invoke(Object, parameter)`
- If the method of the class doesn’t accepts any parameter then `null` is passed as argument.

Reflection – observation 2

- Through reflection we can access the private variables and methods of a class with the help of its class object and invoke the method by using the object as discussed above. We use below two methods for this purpose.
  - `Class.getDeclaredField(FieldName)` : Used to get the private field. Returns an object of type Field for specified field name.
  - `Field.setAccessible(true)` : Allows to access the field irrespective of the access modifier used with the field.

Advantages of Using Reflection

- **Extensibility Features**: An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.
- **Debugging and testing tools**: Debuggers use the property of reflection to examine private members on classes.

Drawbacks of Using Reflection

- **Performance Overhead**: Reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications.
- **Exposure of Internals**: Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.

Uppgift: Reflection

- code/03 - cars/car
- Använd reflection för att lista konstruktorer, metoder och egenskaper.

Annotation

- Java Annotations are a special type of comments in your code that can:
  - Embed instructions for the Java compiler
  - Embed instructions for source code processing tools
  - Embed meta data which can be read at runtime by your Java application or third party tools
    - Spring

Accessing Java Annotations via Java Reflection

- Normally, Java annotations are not present in your Java code after compilation.
- It is possible, however, to define your own annotations that are available at runtime.
- These annotations can then be accessed via Java Reflection, and used to give instructions to your program, or some third party API.

Annotation Basics

- A Java annotation in its shortest form looks like this:
- `@Entity`
- The @ character signals to the compiler that this is an annotation. The name following the @ character is the name of the annotation. In the example above the annotation name is Entity.

Annotation Elements

- A Java annotation can have elements for which you can set values. An element is like an attribute or parameter. Here is an example of a Java annotation with an element:
- `@Entity(tableName = "vehicles")`
- The annotation in this example contains a single element named `tableName`, with the value set to *vehicles*. Elements are enclosed inside the parentheses after the annotation name. Annotations without elements do not need the parentheses.

Annotation Elements

- An annotation can contain multiple elements. Here is a multiple element Java annotation example:
- `@Entity(tableName = "vehicles", primaryKey = "id")`
- In case an annotation contains just a single element, it is convention to name that element value, like this:
- `@InsertNew(value = "yes")`
- When an annotation just contains a single element named value, you can leave out the element name, and just provide the value. Here is an example of an annotation element with only the value provided:
- `@InsertNew("yes")`

Annotation Placement

- You can place Java annotations above classes, interfaces, methods, method parameters, fields and local variables. Here is an example annotation added above a class definition:

```
@Entity
public class Vehicle {
}
```

- The annotation starts with the @ character, followed by the name of the annotation. In this case, the annotation name is Entity. The Entity annotation is an annotation I have made up. It doesn't have any meaning in Java.

Annotation Placement

- Here is a bigger example with annotations above both the class, fields, methods, parameters and local variables:
- `code/16 - annotation/example/Vehicle.java`
- The annotations are again just annotations I have made up. They have no specific meaning in Java.

Built-in Java Annotations

- Java comes with three built-in annotations which are used to give the Java compiler instructions.
  - @Deprecated
  - @Override
  - @SuppressWarnings

Annotations - @Deprecated

- The @Deprecated annotation is used to mark a class, method or field as deprecated, meaning it should no longer be used. If your code uses deprecated classes, methods or fields, the compiler will give you a warning. Here is @Deprecated Java annotation example:

```
@Deprecated
public class MyComponent {
}
```

- The use of the @Deprecated Java annotation above the class declaration marks the class as deprecated.
- You can also use the @Deprecated annotation above method and field declarations, to mark the method or field as deprecated.

Annotations - @Deprecated

- When you use the @Deprecated annotation, it is a good idea to also use the corresponding @deprecated JavaDoc symbol, and explain why the class, method or field is deprecated, and what the programmer should use instead. For instance:

```
@Deprecated
/**
  @deprecated Use MyNewComponent instead.
*/
public class MyComponent {
}
```

Annotations - @Override

- The @Override Java annotation is used above methods that override methods in a superclass. If the method does not match a method in the superclass, the compiler will give you an error.
- The @Override annotation is not necessary in order to override a method in a superclass. It is a good idea to use it still, though. In case someone changed the name of the overridden method in the superclass, your subclass method would no longer override it. Without the @Override annotation you would not find out. With the @Override annotation the compiler would tell you that the method in the subclass is not overriding any method in the superclass.

Annotations - @Override

```
public class MySuperClass {
    public void doTheThing() {
        System.out.println("Do the thing");
    }
}

public class MySubClass extends MySuperClass{
    @Override
    public void doTheThing() {
        System.out.println("Do it differently");
    }
}
```

Annotations - @SuppressWarnings

- The @SuppressWarnings annotation makes the compiler suppress warnings for a given method.
- For instance, if a method calls a deprecated method, or makes an insecure type cast, the compiler may generate a warning.
- You can suppress these warnings by annotating the method containing the code with the @SuppressWarnings annotation.

```
@SuppressWarnings
public void methodWithWarning() {

}
```

Uppgift: Annotations

- Skapa klassen Person med subklassen Student.
- Skapa metoden print() i båda klasserna.
  - Låt dem skriva ut olika saker och testa att de fungerar.
- Sätt en annotation på metoden i subklassen.
- Byt namn på metoden i föräldraklassen och se vad som händer när ni kompilerar programmet.
- Lägg till en annotation för att kompilatorn ska strunta i varningen.
- Märk metoden som deprecated och se vad som händer.

Creating Your Own Java Annotations

- Om man vill veta mer om att skapa egna annotations:
- http://tutorials.jenkov.com/java/annotations.html#creating-your-own-java-annotations

Creating Your Own Java Annotations

- Om man vill veta mer om att skapa egna annotations:
- http://tutorials.jenkov.com/java/annotations.html#creating-your-own-java-annotations

Design patterns

- A design patterns are well-proved solution for solving the specific problem/task. Example:
  - Problem Given:
    - Suppose you want to create a class for which only a single instance (or object) should be created and that single object can be used by all other classes.
  - Solution:
    - Singleton design pattern is the best solution of above specific problem. So, every design pattern has some specification or set of rules for solving the problems. What are those specifications, you will see later in the types of design patterns.
- Design patterns are programming language independent strategies for solving the common object-oriented design problems.
- By using the design patterns you can make your code more flexible, reusable and maintainable. It is the most important part because java internally follows design patterns.

Advantage of design pattern

- They are reusable in multiple projects.
- They provide the solutions that help to define the system architecture.
- They capture the software engineering experiences.
- They provide transparency to the design of an application.
- They are well-proved and testified solutions since they have been built upon the knowledge and experience of expert software developers.
- Design patterns don’t guarantee an absolute solution to a problem. They provide clarity to the system architecture and the possibility of building a better system.

When should we use the design patterns?

- We use the design patterns during the analysis and requirement phase of Software Development Life Cycle.
- Design patterns ease the analysis and requirement phase of SDLC by providing information based on prior hands-on experiences.

Categorization of design patterns

- Basically, design patterns are categorized into two parts:
  - Core Java (or JSE) Design Patterns
  - JEE Design Patterns

Core Java Design Patterns

- In core java, there are mainly three types of design patterns, which are further divided into their sub-parts:
  - Creational Design Pattern
  - Structural Design Pattern
  - Behavioral Design Pattern

Creational Design Pattern

- Factory Pattern
- Abstract Factory Pattern
- Singleton Pattern
- Prototype Pattern
- Builder Pattern

Structural Design Pattern

- Adapter Pattern
- Bridge Pattern
- Composite Pattern
- Decorator Pattern
- Facade Pattern
- Flyweight Pattern
- Proxy Pattern

Behavioral Design Pattern

- Chain Of Responsibility Pattern
- Command Pattern
- Interpreter Pattern
- Iterator Pattern
- Mediator Pattern
- Memento Pattern
- Observer Pattern
- State Pattern
- Strategy Pattern
- Template Pattern

Uppgift – Design Pattern

- Ni ska få några olika design patterns att titta på, förstå och prova i era grupper.
- När vi återsamlas ska ni få förklara för resten av klassen hur era design patterns fungerar.