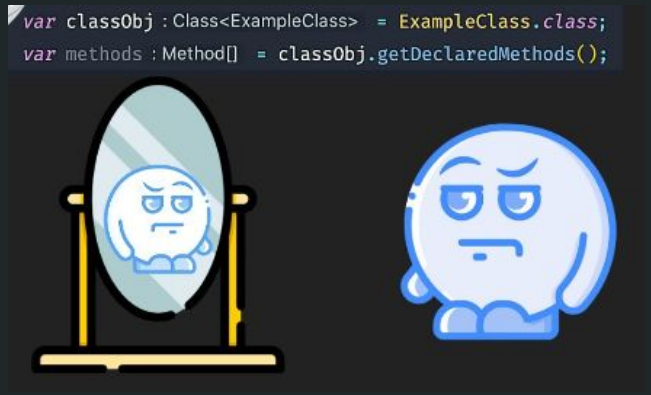
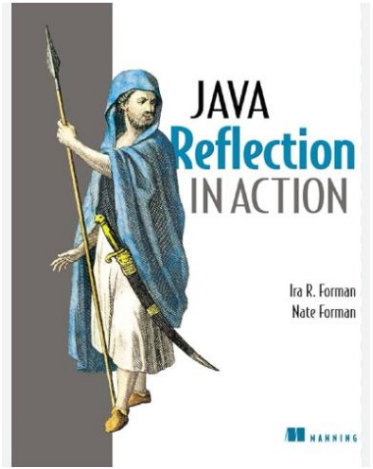


Programação Reflexiva em Java

Reflective Programming
Reflection

O que é programação reflexiva ?



Programação reflexiva é um tipo específico de metaprogramação que se refere à capacidade de um programa inspecionar e modificar sua própria estrutura e comportamento em tempo de execução. Em outras palavras, é a habilidade de um programa examinar e alterar seu próprio comportamento ou estrutura de dados enquanto está em execução.

Principais Características:

- **Introspecção**: Obter informações sobre o próprio código ou estrutura em tempo de execução, como classes, métodos e campos.
- **Modificação Dinâmica**: Alterar ou interagir com a estrutura ou o comportamento do programa em tempo de execução.
- **Anotações e Metadados**: Em algumas linguagens, você pode usar anotações ou metadados para modificar ou interagir com o comportamento do programa.

Exemplos de Programação Reflexiva

- **Java Reflection API**: Permite inspecionar e modificar a estrutura de classes, métodos e campos em tempo de execução.

  → java.lang.reflect

- **Python Reflection**: Usar funções como `getattr()`, `setattr()` e `hasattr()` para introspecção e modificação de objetos e suas propriedades em tempo de execução.

- **Ruby Metaprogramming**: Alterar ou adicionar métodos e propriedades a classes e objetos em tempo de execução.

Meta-Programação ?

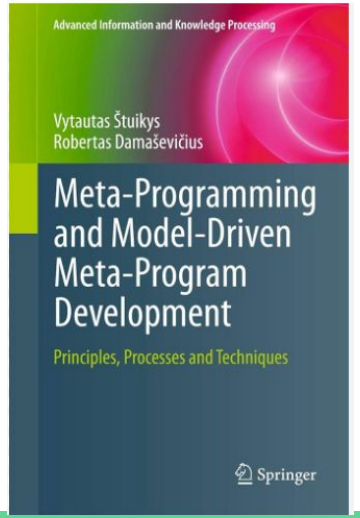
Metaprogramação é a prática de escrever programas que podem manipular ou gerar outros programas, ou mesmo o próprio código do programa que está em execução. Em outras palavras, é a capacidade de um programa para tratar seu próprio código como dados, para ler, modificar ou gerar código dinamicamente.

Principais Características:

- **Geração de Código**: Criar ou modificar código em tempo de compilação ou execução.
- **Macros e Templates**: Em algumas linguagens, como C++ e Lisp, você pode usar macros e templates para gerar código automaticamente.
- **Introspecção e Modificação**: Possibilidade de introspecção (analisar o próprio código ou estrutura) e modificação em tempo de execução ou compilação.

Exemplos de Metaprogramação:

- **C++ Templates**: Permitem a criação de código genérico que é expandido durante a compilação.
- **Macros em Lisp**: Permitem a criação de novos constructos de linguagem e a transformação de código.
- **Geradores de Código**: Ferramentas que geram código com base em especificações ou metadados.



Fontes interessantes

- https://www.devmedia.com.br/reflection-api-reflexao-em-java/8455
- https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/package-summary.html
- https://www.oracle.com/technical-resources/articles/java/javareflection.html

Principais classes da Biblioteca Reflexion (java.lang.reflect)

Classe	Descrição
`Class<T>`	Representa a estrutura de uma classe em tempo de execução.
`Method`	Representa um método de uma classe ou interface.
`Field`	Representa um atributo (campo) de uma classe.
`Constructor<T>`	Representa um construtor de uma classe.
`Modifier`	Utilizada para analisar modificadores (public, private, static, etc).
`Array`	Permite manipular arrays dinamicamente.
`Proxy`	Cria proxies dinâmicos para interfaces.
`Parameter`	Representa os parâmetros de métodos ou construtores.
`AnnotatedElement`	Interface comum para elementos que podem ter anotações.

Principais métodos da Biblioteca Reflexion (java.lang.reflect)

✔ **Classe** `Class<T>`

- `getName()` – retorna o nome completo da classe.
- `forName(String className)` – carrega uma classe pelo nome.
- `getDeclaredMethods()` / `getMethods()` – lista métodos (com ou sem herança).
- `getDeclaredFields()` / `getFields()` – lista campos.
- `getDeclaredConstructors()` / `getConstructors()` – lista construtores.
- `newInstance()` – cria instância (obsoleto em Java 9+, usar `getConstructor().newInstance()`).

✔ **Classe** `Method`

- `getName()` – nome do método.
- `invoke(Object obj, Object... args)` – executa o método.
- `getParameterTypes()` – retorna os tipos dos parâmetros.
- `setAccessible(true)` – permite acessar métodos privados.

Principais métodos da Biblioteca Reflexion (java.lang.reflect)

✔ **Classe** `Field`

- `getName()` – nome do campo.
- `get(Object obj)` / `set(Object obj, Object value)` – lê/escreve o valor.
- `setAccessible(true)` – permite acesso a campos privados.

✔ **Classe** `Constructor<T>`

- `newInstance(Object... initargs)` – instancia um objeto usando o construtor.

Exemplo 1

- Neste exemplo, o objetivo é criar uma classe simples e obter informações básicas da classe em tempo de execução
- Passos:
  - Considere a classe Person
  - Crie um Main que usa reflexão para listar:
    - o nome da classe;
    - listar todos os métodos
    - listar todos os atributos

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

Resolução

```
import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class ClassInspector {

    public static void main(String[] args) {
        // Criar uma instância da classe Person
        Person person = new Person("John", 30);

        // Obter a classe do objeto
        Class<?> personClass = person.??????

        // Imprimir o nome da classe
        System.out.println("Class Name: " + personClass.getName());

        // Listar todos os métodos da classe
        ???? methods = personClass.????();
        System.out.println("Methods:");
        for (Method method : methods) {
            System.out.println(" - " + method.getName());
        }

        // Listar todos os atributos da classe
        ????? fields = personClass.?????;
        System.out.println("Fields:");
        for (Field field : fields) {
            System.out.println(" - " + field.getName());
        }
    }
}
```

Exemplo 2 - framework para inspecionar

Criar um “framework” cujo objetivo é inspecionar uma classe criada pelo usuário

Passos:

- Engenharia do framework (engenheiro do framework)
  - Criar uma classe abstrata que representa o framework em um pacote separado chamado Framework
  - Criar um método no framework que inspeciona a subclasse criada pelo usuário listando todos os métodos que existem na classe criada pelo usuário
- Engenharia da aplicação (engenheiro da aplicação)
  - Criar uma classe do usuário como subclasse da classe do framework (em um pacote chamado aplication). Adicione nessa classe alguns métodos só para teste
  - Criar um Main que invoca o método do framework para listar os métodos existentes na classe do usuário
- O Main deve funcionar assim:

```
public class Main {

    public static void main(String[] args) {

        // Instanciar a classe do usuário
        UserClass userInstance = new UserClass();

        // Usar o método do framework para inspecionar a classe do usuário
        userInstance.inspectClass();
    }
}
```

Classe do Usuário - quem usa o framework

```
public class UserClass extends BaseFrameworkClass {

    public void methodOne() {
        System.out.println("Executing methodOne.");
    }

    public void methodTwo() {
        System.out.println("Executing methodTwo.");
    }
}
```

Classe do Framework

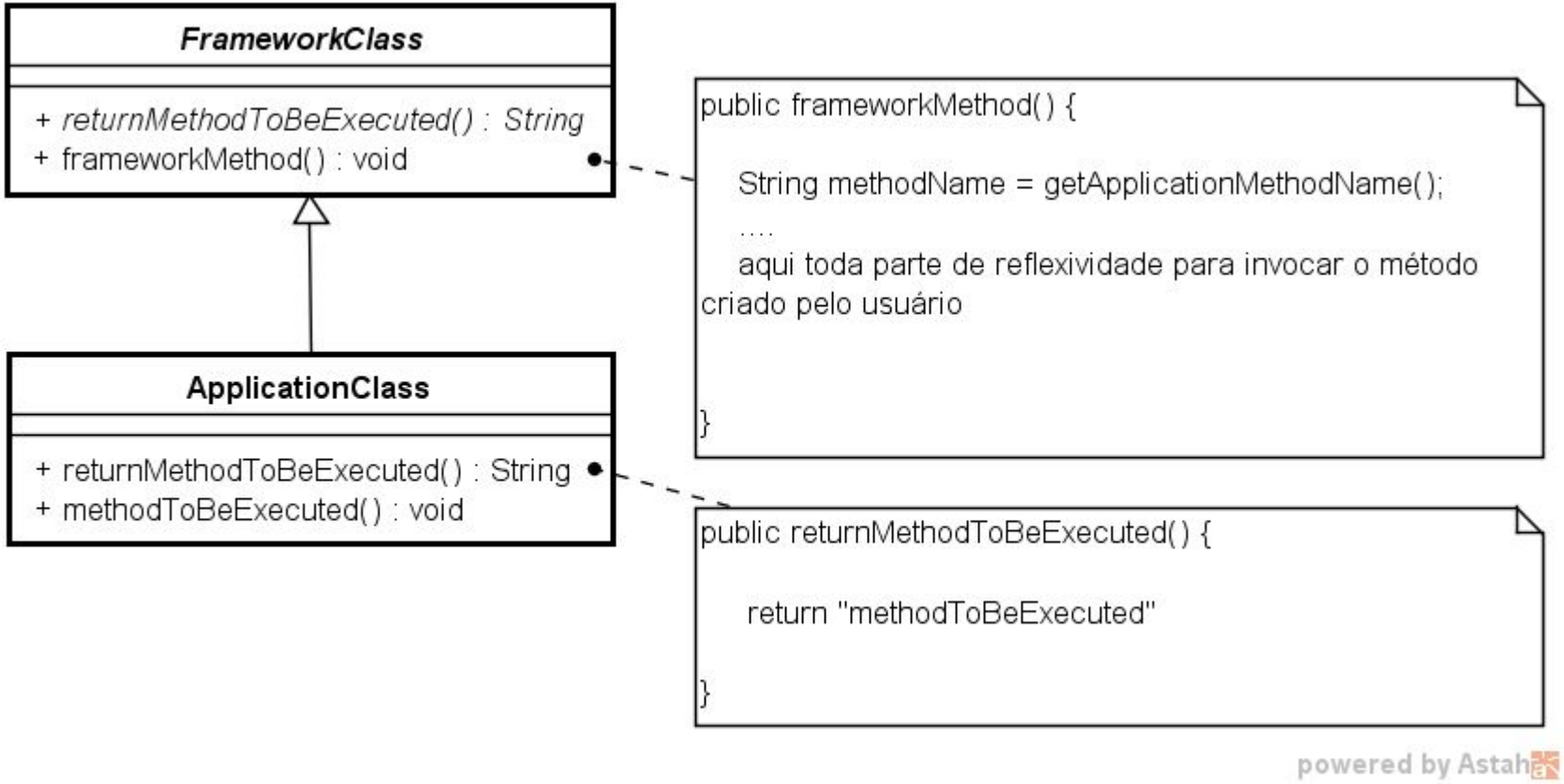
```
import java.lang.reflect.Method;

public abstract class BaseFrameworkClass {

    // Método que será usado para inspecionar a classe filha

    public void inspectClass() {
        Class<?> clazz =  this.getClass();   //this refere-se à classe que estiver chamando
este método

        System.out.println("Inspecting class: " + clazz.getName());

        // Listar todos os métodos da classe filha
        { pegar todos os métodos da classe filha ??? }   ←--  preencher …
        Method methods[] = this.getClass().getDeclaredMethods();
        for (Method method : methods) {
            System.out.println("Method: " + method.getName());
        }
    }
}
```

Exemplo 3

- O código do framework precisa **invocar** um método que não existe ainda. O método será criado pelo usuário, mas o desenvolvedor do framework não sabe ainda qual é o nome do método.
- Operacionalização
  - Criar uma classe abstrata que pertence ao framework
  - Nessa classe abstrata incluir um método que invoca o método criado pelo usuário em uma classe filha (Padrão Template Method).
  - O nome do método a ser executado precisa ser retornado pelo eng. da app
  - O framework também não sabe de antemão o nome da classe que será criada pelo usuário
  - O usuário deve criar uma classe qualquer e também um método com qualquer nome e o framework deve ser capaz de invocar o método criado

Detalhes de como implementar...

- Como instanciar um objeto de uma classe que não se sabe o nome ainda ?
  - pegar o nome da classe em String a partir de um objeto existente:
    - **String algumaClasse = this.getClass().getName();**
  - Carregar a classe dinamicamente após ter o nome dela
    - **Class<?> nomeDaClasse = Class.forName(algumaClasse);**
  - Como verificar se uma classe é filha de outra ?
    - **if (FrameworkClass.class.isAssignableFrom(algumaClasse))**
  - Como instanciar uma classe ?
    - **Object instance = nomeDaClasse.getDeclaredConstructor().newInstance();**
- Como executar um método a partir do objeto de uma classe ?
  - Indicar um método existente como uma String
    - String methodName = “metodoExistente”;
  - Cria uma instância da classe Method
    - Method method = userClass.getMethod(metodoExistente);
  - Invocar/Executar o método
    - method.invoke(instance);


FrameworkClass
+ returnMethodToBeExecuted() : String
+ frameworkMethod() : void
public frameworkMethod() {
String methodName = getApplicationMethodName();
....
aqui toda parte de reflexividade para invocar o método criado pelo usuário
}
ApplicationClass
+ returnMethodToBeExecuted() : String
+ methodToBeExecuted() : void
public returnMethodToBeExecuted() {
return "methodToBeExecuted"
}
powered by Astah

Main

```
package aplicacao;
import framework.FrameworkClass;
public class Main {
   public static void main(String[] args) {
      // Cria uma instância da classe do framework (classe concreta do usuário)
      FrameworkClass user = new UserClass();
      // Executa o método especificado
      user.execute();
   }
}
```

Classes da aplicação

```java
package aplicacao;
import framework.FrameworkClass;
public class UserClass extends FrameworkClass {
  @Override
  protected String getMethodName() {
    return "customMethod"; // Nome do método que será chamado pelo framework
  }
  // Define o método que será invocado pelo framework
  public void customMethod() {
    System.out.println("Método da aplicação que está sendo executado - o framework conseguiu !!!");
  }
}
```

```java
package aplicacao;
import framework.FrameworkClass;
public class Main {
  public static void main(String[] args) {
    // Cria uma instância da classe do framework (classe concreta do usuário)
    FrameworkClass user = new UserClass();
    // Executa o método especificado
    user.execute();
  }
}
```

Classe do Framework

```
package framework;
import java.lang.reflect.Method;
public abstract class FrameworkClass {

  // Método abstrato que deve ser implementado pelo usuário
  protected abstract String getMethodName();

  // Método concreto que usa reflexão para invocar o método especificado na instância atual
  public void execute() {
     try {
       // Obtém o nome do método a ser invocado
        String methodName = getMethodName();
        // Obtém a classe atual (classe filha)
        Class<?> userClass = this.getClass();
        // Obtém o método da classe filha baseado no nome
        Method method = userClass.getMethod(methodName);
        // Invoca o método na instância atual
        method.invoke(this);


     } catch (Exception e) {
        System.out.println("Error executing method: " + e.getMessage());
     }
  }
}
```

Exemplo 4 - Anotações

- Criar um novo framework cujo objetivo é rastrear métodos que estejam anotados com a anotação “Track Execution”
- O framework deve proporcionar uma funcionalidade (método) que faz duas coisas: executa e rastreia métodos anotados.
- O usuário deve criar uma classe concreta que herda da classe do Framework e deve anotar quais métodos ele gostaria que fossem executados e rastreados
- A classe do usuário pode ter outros métodos, mas somente os anotados é que devem ser rastreados e executados
- Rastrear o método significa simplesmente imprimir algo antes da execução dele e após a execução dele (*Tracing*)

Como criar uma annotation e pegá-la em Runtime...

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)

      public @interface TrackExecution {

}
```

```
if (method.isAnnotationPresent(TrackExecution.class)) {
```

Classe Main

```
public class Main {

    public static void main(String[] args) {
        // Instanciar a classe do usuário
        UserClass userInstance = new UserClass();

        // Chamar o método do framework para rastrear e executar métodos anotados
        userInstance.trackAndExecuteMethods();
    }
}
```

```
Tracking and executing methods in class: aplicacao.UserClass
Tracking method: methodTwo
Starting execution of method: methodTwo
Executing methodTwo.
Finished execution of method: methodTwo
Tracking method: methodOne
Starting execution of method: methodOne
Executing methodOne.
Finished execution of method: methodOne
```

Classe do Usuário

```
public class UserClass extends BaseFrameworkClass {

    @TrackExecution
    public void methodOne() {
        System.out.println("Executing methodOne.");
    }

    @TrackExecution
    public void methodTwo() {
        System.out.println("Executing methodTwo.");
    }

    public void methodThree() {
        System.out.println("Executing methodThree.");
    }
}
```

Classe do Framework

```
import java.lang.reflect.Method;

public abstract class BaseFrameworkClass {

    // Método template para rastrear e executar métodos anotados
    public void trackAndExecuteMethods() {

        Class<?> clazz = this.getClass();

        System.out.println("Tracking and executing methods in class: " + clazz.getName());

        // Listar e executar métodos anotados
        Method[] methods = clazz.getDeclaredMethods();
        for (Method method : methods) {
            if (method.isAnnotationPresent(TrackExecution.class)) {
                System.out.println("Tracking method: " + method.getName());
                try {
                    // Antes da execução
                    System.out.println("Starting execution of method: " + method.getName());
                     method.invoke(this);
                    // Depois da execução
                    System.out.println("Finished execution of method: " + method.getName());

                } catch (Exception e) {
                    System.out.println("Failed to execute method: " + method.getName());
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Framework de Persistência

Descrição do Framework

- É um framework de persistência, isto é, ele deve facilitar o desenvolvimento da parte de persistência de dados
- O usuário do framework (desenvolvedor) deve usar o framework da seguinte forma:

    1 - Cria uma classe de aplicação qualquer que seja uma “classe de entidade”, isto é, uma classe que possui uma tabela de BD correspondente. Usuário deve usar anotações para mapear a classe a uma tabela do banco

    2 - É de responsabilidade do usuário criar o BD

    3 - Cria um programa principal que instancia a sua classe e que invoca métodos do framework

    4 - Usuário não precisará escrever nada de SQL, só usar o framework

- Veja no próximo slide ➜

SQLite

- Comandos básicos
  - criar um novo BD ou abrir um que já existe
    - C:\bancos > sqlite3 novoBanco.db
    - sqlite> .open novoBanco.db
  - listar tabelas de um BD existente
    - sqlite> .open novoBanco.db
    - sqlite> .tables
  - ver a estrutura de uma tabela
    - sqlite> .schema nomeDaTabela
  - Saber onde o sqlite está criando os bancos (caminho absoluto)
    - .databases
  -

Classes criadas pelo Usuário do Framework

```java
import framework.Column;
import framework.Entity;

@Entity(tableName = "Cliente")

public class Cliente {

  @Column(name = "id")
  private int id;

  @Column(name = "name")
  private String name;

  @Column(name = "email")
  private String email;

  // Construtor, getters e setters
...
```

```java
import java.sql.SQLException;
import framework.PersistenceFramework;

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

      PersistenceFramework framework = new PersistenceFramework();

      framework.setDBAbsolutePath("/Users/Valter/NovoBanco.db");

      Cliente cliente = new Cliente(7, "John Doe", "john.doe@example.com");
      try {

          framework.insert(cliente);

      } catch (SQLException | IllegalAccessException e) {
          e.printStackTrace();
      }
  }
}
```

Obtém acesso ao framework

Informa onde está o BD

Invoca o método Insert() que insere um cliente no BD

```java
import java.sql.SQLException;
import framework.PersistenceFramework;

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

        PersistenceFramework framework = new PersistenceFramework();

        framework.setDBAbsolutePath("/Users/Valter/NovoBanco.db");

        Cliente cliente = new Cliente(7, "John Doe", "john.doe@example.com");
        try {

            framework.insert(cliente);

        } catch (SQLException | IllegalAccessException e) {
            e.printStackTrace();
        }
    }
}
```

O que o método o.insert(cliente) deve fazer ?

- verificar se a classe do objeto O está anotada com @Entity
- Se estiver, deve começar a criar uma String SQL de INSERT
- Para isso, precisa pegar todos os atributos da classe que estejam anotados com @column
- continua montando a String SQL com o nome das colunas da tabela
- Depois, pega os valores dos atributos do objeto o
- Abre a conexão com o BD
- Cria um statement com a String
- executa o statement com o executeUpdate()

Conexão com Banco de Dados

- Recomenda-se o uso do SQLite
- Baixar driver jdbc e slf4j de https://drive.google.com/drive/u/1/folders/1vGlkd567qcqOeZ9-eQehQ_hNjy1jhYrQ
- Se estiver usando Eclipse, carregar essas libraries nas libraries do projeto
- String de conexão com o BD ➜ `String url = "jdbc:sqlite:/Users/Valter/NovoBanco.db";`
- Estabelecer a conexão ➜ `Connection conn = DriverManager.getConnection(url);`
- Criar um statement and executar o comando SQL

```
Statement stmt = conn.createStatement();
// Criar a tabela se não existir
String createTableSQL = "CREATE TABLE IF NOT EXISTS Cliente (" +
                        "id INTEGER PRIMARY KEY, " +
                        "name TEXT NOT NULL, " +
                        "email TEXT NOT NULL);";
stmt.executeUpdate(createTableSQL);
```

- Fechar o statement e fechar a conexão ➜ stmt.close e conn.close()

Conexão com Banco de Dados

- Em Java, uma forma bastante usual de executar uma String SQL é por meio de um PreparedStatement.
- Veja um exemplo

```
String sql = "INSERT INTO users (username, email) VALUES (?, ?)";

PreparedStatement statement = connection.prepareStatement(sql);

// Substituindo os placeholders
statement.setString(1, "john_doe"); // O primeiro ? será substituído por "john_doe"
statement.setString(2, "john.doe@example.com"); // O segundo ? será substituído por "john.doe@example.com"

// Executando a instrução
statement.executeUpdate();
```

```
Object value = field.get(entity);
 statement.setObject(parameterIndex++, value);
```

Neste caso, não precisa se preocupar com o tipo do valor que será inserido.

Como pegar o valor do atributo do Annotation ?

```
Entity entityAnnotation = clazz.getAnnotation(Entity.class);
String tableName = entityAnnotation.tableName();
```

Dessa forma, pode-se pegar o nome da tabela do BD correspondente à classe criada pelo usuário

Como ter acesso a um atributo privado ?

```
field.setAccessible(true);
```

Resolução Parcial Framework Persistência

Criação dos Anotations

Anotação para classes
(tabelas do BD)

```
package framework;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)


public @interface Entity {
    String tableName();
}
```

Anotação para atributos
(colunas do DB)

```
package framework;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)

public @interface Column {
   String name();
}
```