Singleton

Introdução

- Como a maioria dos programadores organizaria o código para acessar informação de configuração?
- Eis um exemplo:

```
public class Config {
  public static final String DEFAULT_READ_COMMUNITY_NAME = "
  public static final String DEFAULT_CONCENTRATOR_RETRIES =
  public static final String DEFAULT_CONCENTRATOR_TIMEOUT =
  public static final String DEFAULT_URL_CONSUMER_DATABASE =
}

public class Xpto {
    // ...
    façaAlgo(Config.DEFAULT_READ_COMMUNITY_NAME);
    // ...
}
```

- Tem vários problemas com esta forma de tratar a questão
  - Por exemplo, como manter os valores default em arquivo?
- Vamos ver uma solução aqui:

```
package salame.util;
import java.io.*;
import java.util.Properties;
import java.net.URL;
import salame.SalameException;

public class Config {
  private static final String configPropFile = "salame.prop"
  private static final String DEFAULT_READ_COMMUNITY_NAME =
  private static final String DEFAULT_CONCENTRATOR_RETRIES =
  private static final String DEFAULT_CONCENTRATOR_TIMEOUT =
  private static final String DEFAULT_URL_CONSUMER_DATABASE

  private String readCommunityName;
  private int concentratorRetries;
  private int concentratorTimeout;
```

```
  private String urlConsumerDatabase;

  static {
    loadConfiguration();
  }

  private void loadConfiguration() {
    prop = new Properties();
    try {
      URL resource = Config.class.getResource(configPropFile
      if(resource == null) {
        throw new SalameException("Nao pode achar recurso: "
      }
      prop.load(new BufferedInputStream(resource.openStream(
    } catch(Exception ex ) {
      System.err.println(ex.getMessage());
      System.exit(1);
    }
    // set config variables values
    readCommunityName = prop.getProperty("ReadCommunityName"
        DEFAULT_READ_COMMUNITY_NAME);
    concentratorRetries = Integer.parseInt(prop.getProperty(
        DEFAULT_CONCENTRATOR_RETRIES));
    concentratorTimeout = Integer.parseInt(prop.getProperty(
        DEFAULT_CONCENTRATOR_TIMEOUT));
    urlConsumerDatabase = prop.getProperty("UrlConsumerDatab
  }

  public static String getReadCommunityName() {
    return readCommunityName;
  }
  public static int getConcentratorRetries() {
    return concentratorRetries;
  }
  public static int getConcentratorTimeout() {
    return concentratorTimeout;
  }
  public static String getUrlConsumerDatabase() {
   return urlConsumerDatabase;
  }
}

public class Xpto {
    // ...
    façaAlgo(Config.getReadCommunityName());
    // ...
}
```

- E agora, qual é o problema?
  - Não temos polimorfismo, devido ao uso de métodos estáticos
  - Como posso ter dois (ou mais) objetos de configuração para situações diferentes?
- Para ter polimorfismo, precisamos usar métodos de instâncias e não estáticos
- Mas, devemos ter uma **única instância** da classe Config

Objetivo

- Assegurar que uma classe tenha uma única instância e prover um ponto de acesso global a esta instância

Motivação

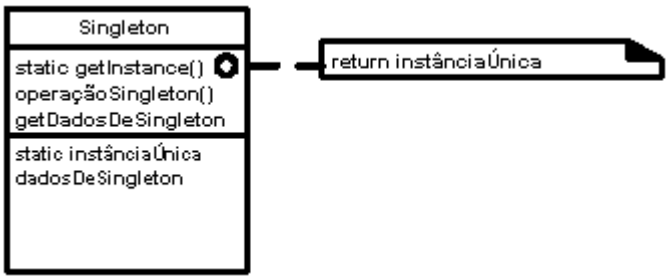
- Algumas classes devem ser instanciadas uma única vez:
  - Um spooler de impressão
  - Um sistema de arquivos
  - Um Window manager
  - Um objeto que contém a configuração de um programa
  - etc.
- Como assegurar que uma classe possua apenas *uma* instância e que esta instância seja facilmente acessível?
  - Uma variável global deixa a instância acessível mas não inibe a instanciação múltipla
- Uma melhor solução: faça com que a classe em si seja responsável pela manutenção da instância única
- Este é o padrão Singleton ("que-possui-apenas-um")

Aplicabilidade

- Use o padrão Singleton quando:
  - Deve haver uma única instância de uma classe e

esta instância deve ser acessada a partir de um ponto de acesso bem-conhecido
- Quando a instância única deve ser extensível através de subclasses e clientes podem usar instâncias diferentes polimorficamente, sem modificação de código

Estrutura



Participantes

- **Singleton**
  - Define uma operação getInstance() que permite que clientes acessem sua instância única
    - É um método estático (class method)
  - Pode ser responsável pela criação de sua instância única

Colaborações

- Clientes acessam a instância apenas através da operação getInstance() do Singleton

Consequências

- Vários benefícios existem:
  - Acesso controlado à instância única
    - O singleton tem controle sobre como e quando clientes acessam a instância
  - Espaço de nomes reduzido
    - O Singleton é melhor que variáveis globais, já

que as "globais" podem ser encapsuladas na instância única, deixando um único nome externo visível

- Permite refinamento de operações e de representação
  - Várias classes Singleton (relacionadas ou não via herança) podem obedecer a mesma interface, permitindo que um singleton particular seja escolhido para trabalhar com uma determinada aplicação em tempo de execução
- Permite a existência de um número variável de instâncias
  - É fácil fazer com que o Singleton crie um número fixo, ou um número máximo de instâncias em vez de apenas uma única instância
  - Apenas a implementação interna do Singleton precisa mudar
- Mais flexível que métodos estáticos
  - Embora possa parecer que podemos fazer o equivalente a um Singleton com métodos estáticos, lembre que isso não permitiria o polimorfismo:

```
NomeDeClasse.xpto();  // chamada não é polimórfica
// ... versus ...
Config conf = Config.getInstance();
conf.xpto();  // essa chamada é polimórfica
```

Considerações de implementação

Como assegurar que haja uma única instância?

- Uma forma é de *não* permitir chamadas ao construtor

```
public class Config implements ConfigIF {
  private static ConfigIF instânciaÚnica = null;

  private Config() {} // o compilador não vai gerar um const
```

```
  public static ConfigIF getInstance() {
    if(instânciaÚnica == null) {
      // "lazy instantiation"
      instânciaÚnica = new Config();
    }
    return instânciaÚnica;
  }
}
```

- Observe que o uso de lazy instantiation é preferível a retornar uma instância estática da classe
  - Lazy instantiation pode ser mais rápida se a instanciação/inicialização for lenta

Exemplo de código

- Veremos um exemplo de Singleton junto com Abstract Factory

Para estudar em casa

- Verifique o que significa "Double-Checked Locking"
  - http://www.javaworld.com/javaworld/jw-02-2001/jw-0209-double.html
  - http://www.cs.umd.edu/~pugh/java/memoryModel/
    - Também aqui
  - http://www.javaworld.com/javaworld/jw-02-2001/jw-0209-toolbox.html
  - http://www.javaworld.com/javaworld/jw-05-2001/jw-0525-double.html
  - http://www.javaworld.com/javaworld/jw-11-2001/jw-1116-dcl.html
  - http://www.javaworld.com/javaworld/jw-03-2001/jw-0323-letters.html
  - http://www.javaworld.com/javaworld/jw-06-2001/jw-0622-letters.html
  - http://www.javaworld.com/javaworld/jw-07-2001/jw-0727-letters.html
  - http://www.javaworld.com/javaworld/jw-11-2001/jw-1130-letters.html
- Que problema double-checked locking tenta resolver?

- Qual é o problema com double-checked locking?
- Quais são as possíveis soluções?

programa