SE 350 - Object-Oriented Software Development

Software Design Patterns: Singleton

Instructor: Stefan Mitsch

Learning Objectives

- Understand the problem solved by Singleton
- Understand the pattern

GoF Design Pattern Space

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype **Singleton**	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Introduction

Singleton ensures that a class has only one instance

- Creational design pattern
- Provides global access to class instance
- Examples
  - Unique account number generator
  - Configuration
  - Logging

Overview

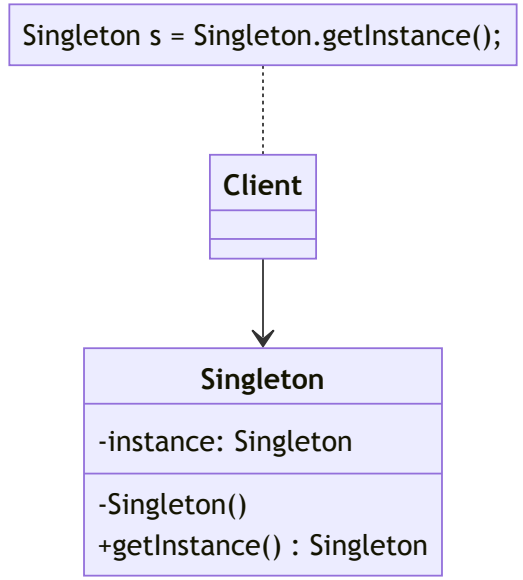
Problem

- Application needs exactly 1 instance
- Do not want to pass instance explicitly to all places needing it

Intent

- Ensure a class has only one instance
- Provide global access to instance
- Lazy initialization (on first use)

Structure



Implementation

- `private` constructors to prohibit external instantation of the class
- `private static` instance variable
- `public static` accessor method to obtain instance

```
public class Configuration {
  // the singleton instance, initialized immediately
  private static Configuration instance = new Configuration();

  // private constructor prohibits external instantiation
  private Configuration() {
    // initialization code
  }

  // public accessor method returns the instance
  public static Configuration getInstance() {
    return instance;
  }
}
```

❓ What are the consequences of this code, what could be done differently?

Eager vs. Lazy Singletons

Eager

Initialized when program loads

- ⊕ Easy implementation
- ⊖ Instance created even if unused
- ⊖ Increased application load time

```
public class Singleton {
  private static final Singleton instance = new Singleton();
  private Singleton() {}
  public static Singleton getInstance() {
    return instance;
  }
}
```

Lazy

Initialized on first access

- ⊕ Avoids unnecessary instantiation
- ⊖ Careful in multi-threaded apps
- ⊖ Increased operation wait time
- ⊖ May result in delayed failures

```
public class Singleton {
  private static Singleton instance;
  private Singleton() {}
  public static Singleton getInstance() {
    if (instance == null) instance = new Singleton();
    return instance;
  }
}
```

Recap: Static Dispatch

- `static` variables are class variables, access `ClassName.variableName`
- `static` methods
  - class methods
  - can only access static variables and other static methods
- `static` blocks
  - executed when class is loaded
  - create static resources
  - can only access static variables
- `static` inner class

```
public class Demo {
  // static variable
  public static int count;

  // static method
  public static int getCount() {
    return count;
  }

  // static initializer block
  static {
    count = 4;
  }

  // static inner class
  static class Inner {}
}
```

Eager Singleton with Exception Handling

```
public class Singleton {
  private static final Singleton instance = new Singleton();
}
```

- What if loading the singleton may raise an exception?

▶ Solution

Singleton in Multi-Threaded Applications

- Must execute `instance == null` check and instantiation `instance = new Singleton()` atomically

```
public class Singleton {
  private static Singleton instance;
  public static synchronized Singleton getInstance() {
    if (instance == null) instance = new Singleton();
    return instance;
  }
  private Singleton() {}
}
```

What can be improved about this implementation?

▶ Solution

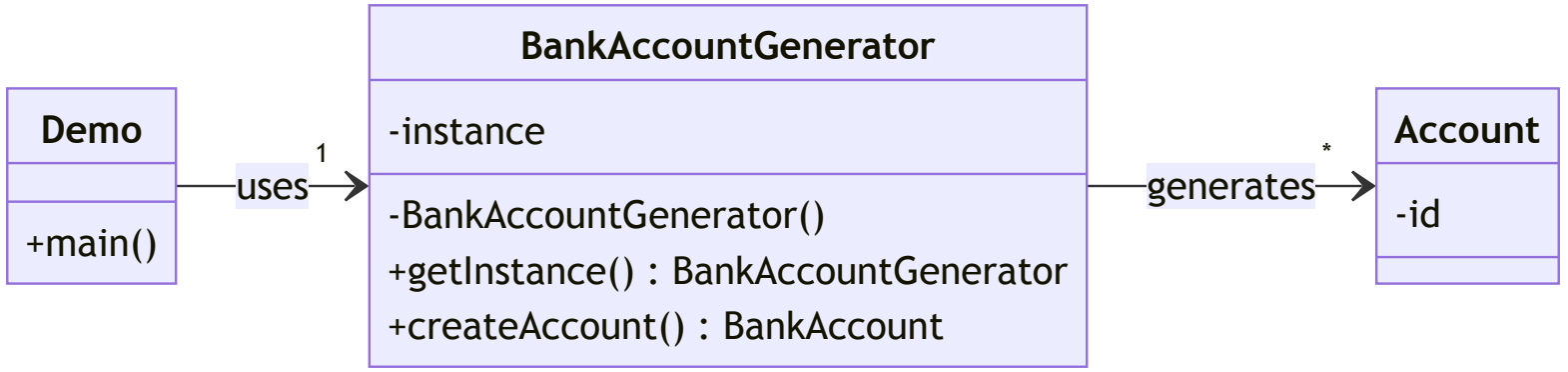
Singleton: Bill Pugh Implementation

- Thread-safe implementation of singleton without explicit synchronization
- Specific to the Java memory model and class initialization
- "Initialization-on-demand holder"
- Exploits that, in Java, `static` inner classes are only loaded when accessed

```
public class Singleton {
  private static class SingletonHolder {
    static final Singleton instance = new Singleton();
  }
  public static Singleton getInstance() {
    return SingletonHolder.instance;
  }
  private Singleton() {}

}
```

Singleton Example: Bank Account




```java
public class BankAccountGenerator {
  private static class SingletonHolder {
    static final BankAccountGenerator instance = new BankAccountGenerator();
  }
  public static BankAccountGenerator getInstance() {
    return SingletonHolder.instance;
  }

  private long nextId = 1L;
  private BankAccountGenerator() {}
  public BankAccount createAccount() {
    // create and return a new account (not thread-safe!)
    return new Account(nextId++);
  }
}
```

```java
public class Demo {
  public static void main(String[] args) {
    BankAccountGenerator g = BankAccountGenerator.getInstance();
    BankAccountGenerator h = BankAccountGenerator.getInstance();
    assert g==h;
    Account a = g.createAccount();
    Account b = g.createAccount();
  }
}
```

Singleton Variations

- Singleton registry provides
  - Instances of subclasses of a singleton
  - Multiple different singletons (e.g., named)
- Singleton destroyer
  - Languages without garbage collection (e.g., C++) must destroy the singleton instance at the end of the program

Discussion

Benefits

- Instantiation and initialization restricted
- Easy global access to single instance
- Lazy initialization possible
- Singletons are not evil

Drawbacks

- Introduces global state
- Can increase coupling
- Requires special treatment in multi-threaded applications
- Can makes unit testing difficult
- Use your Singletons wisely