```
/*
The following is an example of a Pet Factory
implementing the singleton design pattern
*/

public class PetFactory implements Factory {

  // Stores the instance of a pet factory.
  private static PetFactory instance;

  // Private constructor for pet factory.
  private PetFactory(){}

  // Retrieves the instance for a pet factory
  public static PetFactory getInstance){
    if (this.instance == null){
        this.instance = new PetFactory();
    }
    return this.instance;
  }

  public Pet createPet(){
    // Creates pet
    // Complex
  }
}
```

The Singleton

```
/*
The following is an example of a Pet Factory
implementing the singleton design pattern
*/

public class PetFactory implements Factory {

  // Stores the instance of a pet factory.
  private static PetFactory instance;

  // Private constructor for pet factory.
  private PetFactory(){}

  // Retrieves the instance for a pet factory
  public static PetFactory getInstance){
    if (this.instance == null){
        this.instance = new PetFactory();
    }
    return this.instance;
  }

  public Pet createPet(){
    // Creates pet
    // Complex
  }
}
```

- The **singleton** is a **design pattern** where there is at most one object of a particular type throughout the entire application.
  - The first time the singleton is called (generally through a method called `getInstance`), the object is initialized; otherwise, the previously created object is used.
- Singletons are often used when a single instance of an object is desired or as a simple way to create global instances in an OOP language such as Java.
  - Database Connections
  - Authentication Clients
  - Users of a SUA
  - Factories
  - Loggers
- Singletons are particularly challenging to unit-test because there is a single global instance that is consistent throughout all tests.
  - It is hard to inject our own instance for the object to set particular parameters for testing scenarios .

Breaking Dependencies w/ Singleton