الجامعة اللبنانية
Lebanese University

Graphical Interface and Application(I3305)

Chapter 1: Introduction & Design Patterns

Lebanese University

Faculty of Science - Department of Computer Science

Dr. Abed EL Safadi

Outline

- Administrative Details
- Course Description
- Introduction

Outline

- Administrative Details
- Course Description
- Introduction

Useful Info.

Instructor	Abed EL Safadi
Phone	+961 70 80 16 14 (Only WhatsApp)
Email	Abed.Safadi@ul.edu.lb

Resources

The course will use the following textbooks as references:

Program Development in Java: Abstraction, Specication, and Object-Oriented Design, by Barbara Liskov and John Guttag. Addison-Wesley.

Software Engineering for Students. A Programming Approach, by Douglas Bell (fourth edition).

Object-Oriented Software Engineering Practical software development using UML and Java, by Timothy C. Lethbridge and Robert Laganiere (second edition).

Remark : The course also uses many online materials.

Attendance

- You are expected to attend lectures.
- Some lectures are in sequence. If you skip one you are not be able to understand the lecture which follows, if you don't catch up.
- In the case of unexcused absences, you are responsible for making up lecture material that you have missed on your own.
- You are also responsible for any announcements that I make during a class when you are absent.

Outline

- Administrative Details
- Course Description
- Introduction

Course description

1- Introduction

2- Design Patterns

- Creational Patterns
- Structural Patterns
- Behavioral Patterns

3- JavaFX

4- MVC

5- Interconnection with a Database

Outline

- Administrative Details
- Course Description
- Introduction

Outline

- Introduction
- Creational patterns
- Structural patterns
- Behavioral patterns

What Is a Design Pattern?

Christopher Alexander[1] says

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

- Even though Alexander was talking about patterns in buildings and towns, what he says is true about object-oriented design patterns.
- When organizing a program, it is useful to understand the ways that people have organized in the past!
- Design patterns are like a bag of tricks that every competent programmer should understand.
- This doesn't mean that you use them indiscriminately
  - You can recognize situations where they might apply and then
  - Decide whether their use merited in that particular case.

1 : is an architect noted for his theories about design, and for more than 200 building projects in California, Japan, Mexico and around the world.

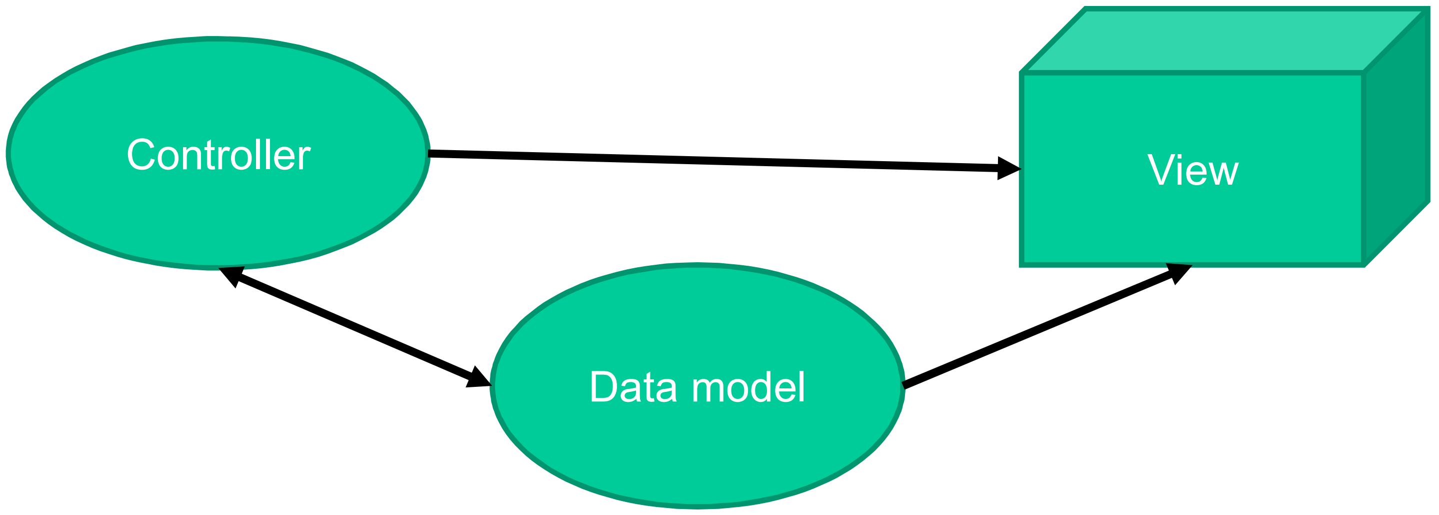
What Is a Design Pattern?

Design patterns are just convenient ways of reusing object-oriented code between projects and between programmers.

What Is a Design Pattern?

- One of the frequently cited frameworks was the **Model-View-Controller** framework, which divided the user interface problem into three parts.

1. The parts were referred to as a *data model* which contain the computational parts of the program,
2. the *view*, which presented the user interface,
3. and the *controller*, which interacted between the user and the view.



What Is a Design Pattern?

- Communication between the user, the GUI and the data should be carefully controlled and this separation of functions accomplished that very nicely.
- Three objects talking to each other using this restrained set of connections is an example of a powerful *design pattern*.

Defining Design Patterns

3 types

1. ***Creational patterns*** are ones that create objects for you, rather than having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case.
2. ***Structural patterns*** help you compose groups of objects into larger structures, such as complex user interfaces or accounting data.
3. ***Behavioral patterns*** help you define the communication between objects in your system and how the flow is controlled in a complex program.

Object Oriented Approaches

- **The fundamental reason for using various design patterns is to keep classes separated and prevent them from having to know too much about one another.**
- There are a number of strategies that OO programmers use to achieve this separation, among them *encapsulation* and *inheritance*.

Design Patterns Are Not About Design

- Design patterns are not about designs such as linked lists and hash tables that can be encoded in classes and reused as is.
- Design patterns are not complex, domain-specific designs for an entire application or subsystem.
- Design patterns are descriptions of ***communicating objects and classes that are customized to solve a general design problem in a particular context.***

Design Pattern Catalog

- Creational patterns: concern the process of object creation
  - Factory
  - Singleton
- Structural patterns: deal with the composition of classes or objects
  - Flyweight
  - Bridge
  - Composite
  - Adaptor
  - Proxy
  - Decorator
- Behavioral patterns: characterize the ways in which classes or objects interact and distribute responsibility.
  - State
  - Iterator
  - Visitor
  - Observer

Outline

- Introduction
- Creational patterns
- Structural patterns
- Behavioral patterns

Factory Pattern

The Factory Pattern is used to create different objects from a **factory**

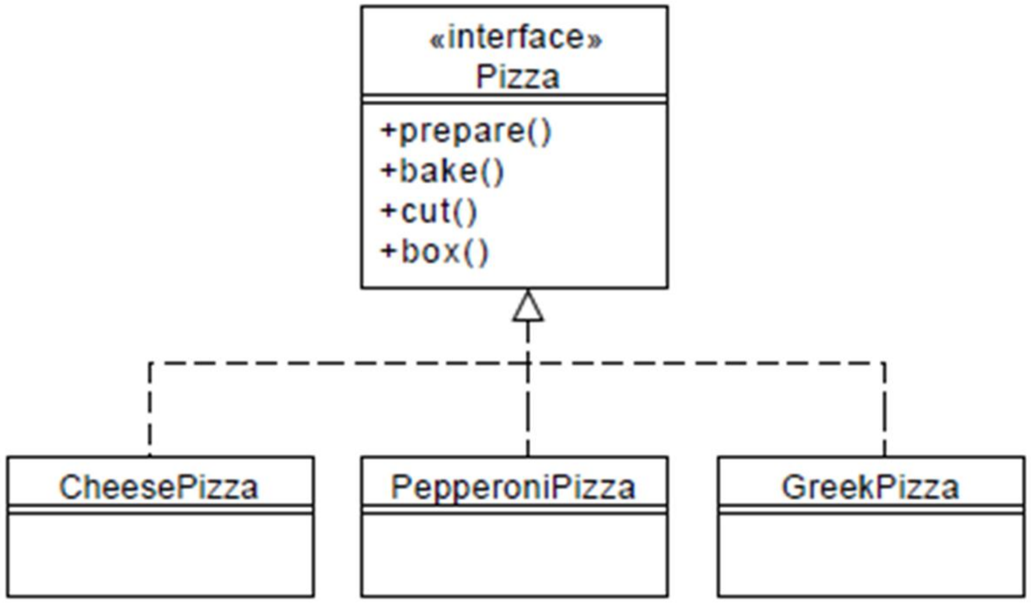
If we have a super class and *n* sub-classes, and based on input parameters, we have to return the object of one of the sub-classes, we use a factory pattern.

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

So instead of having object creation code on client side we encapsulate inside a Factory method

Example

```
public interface Pizza {
   public void prepare();
   public void bake();
   public void cut();
   public void box();
}
```



Example: Bad Solution

```java
public class PizzaStore {

   Pizza orderPizza(String type) {
      Pizza pizza = null;
      if(type.equals("cheese"))
         pizza = new CheesePizza();
      else if(type.equals("greek"))
         pizza = new GreekPizza();
      else if(type.equals("pepporoni"))
         pizza = new PepperoniPizza();

      // Excellent example of "coding to an interface"
      pizza.prepare();
      pizza.bake();
      pizza.cut();
      pizza.box();

      return pizza;
   }
}
```

Example: Solution Using Factory Pattern

```
public class SimplePizzaFactory {

   public static Pizza createPizza(String type) {
      if(type.equals("cheese"))
         return new CheesePizza();
      else if(type.equals("greek"))
         return new GreekPizza();
      else if(type.equals("pepperoni"))
         return new PepperoniPizza();

      return null;
   }
}
```

```
public class PizzaStore {

   Pizza orderPizza(String type) {
      Pizza pizza = SimplePizzaFactory.createPizza(type);
      pizza.prepare();
      pizza.bake();
      pizza.cut();
      pizza.box();

      return pizza;
   }
}
```

Factory Pattern and Factory Method

Example

Pizza store example evolves to include:

- The notion of different franchises that exist in different parts of the country (Beirut, Tripoli, Saida)
- Each franchise will need its own factory to create pizzas that match the proclivities of the locals
- However, we want to retain the preparation process that has made PizzaStore such a great success

Bad Solution

```java
public abstract class PizzaStore {
   Pizza orderPizza(String branch, String type) {
      Pizza pizza = ComplexPizzaFactory.createPizza(branch, type);
      pizza.prepare();
      pizza.bake();
      pizza.cut();
      pizza.box();
      return pizza;
   }
}
```

```java
public class ComplexPizzaFactory {
   public static Pizza createPizza(String branch, String type) {
      if(branch.equals("beirut") && type.equals("cheese"))
         return new BeirutCheesePizza();
      else if(branch.equals("beirut") && type.equals("greek"))
         return new GreekBeirutPizza();
      ...
   }
}
```

- New branch ➔ update createPizza method
- Assume that factory of each branch contains different additional work to do (variables, methods)!
- How you can make the solution more clean (e.g., easy to add new branches without modifying existing methods)

Good Solution: Factory Method

```java
public abstract class PizzaStore {

   protected abstract Pizza createPizza(String type);

   Pizza orderPizza(String type) {
      Pizza pizza = createPizza(type);

      pizza.prepare();
      pizza.bake();
      pizza.cut();
      pizza.box();
      return pizza;
   }
}
```

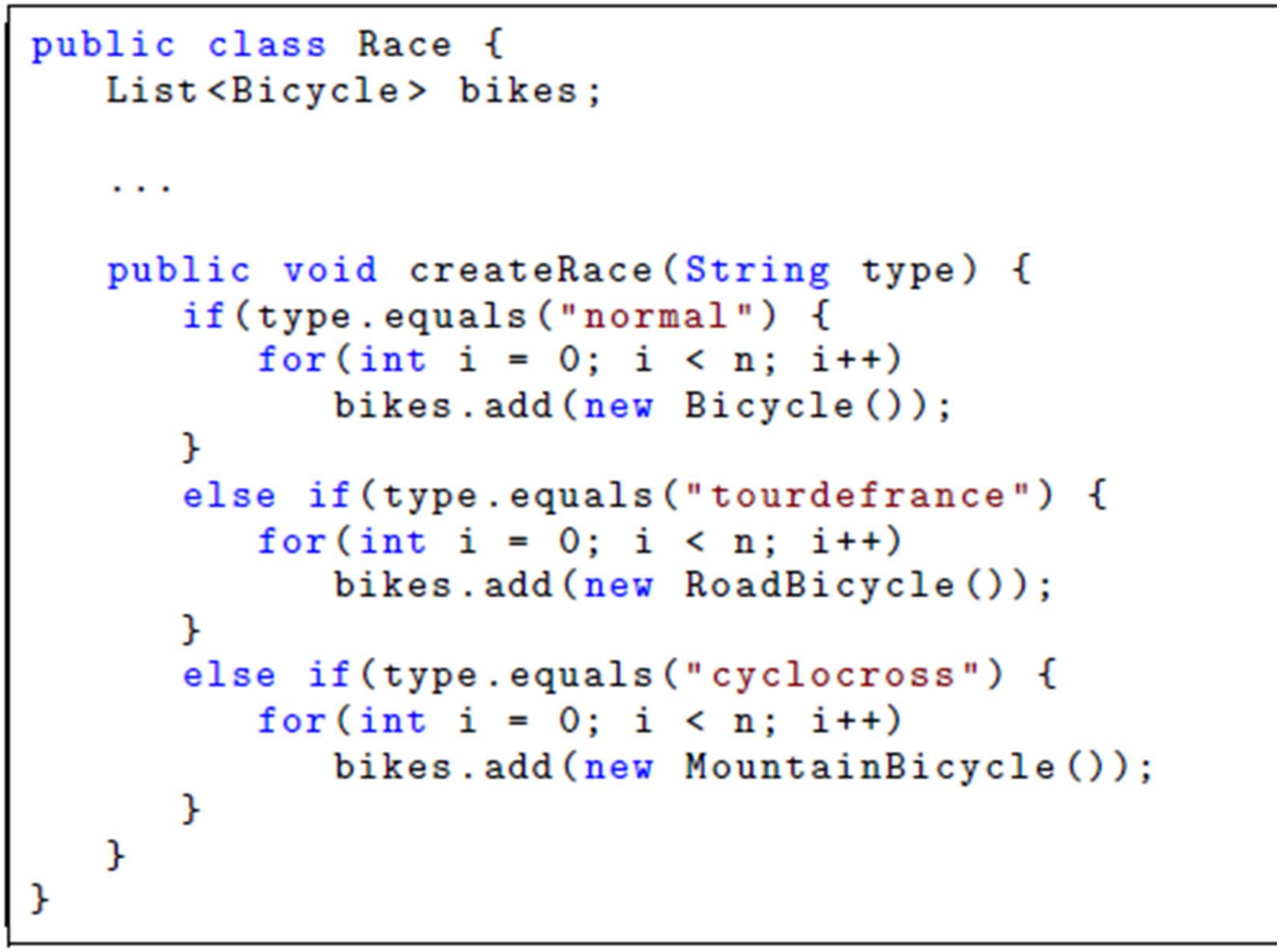
```java
public class BeirutPizzaStore extends PizzaStore {

   public Pizza createPizza(String type) {
      if(type.equals("cheese"))
         return new BeirutCheesePizza();
      else if(type.equals("greek"))
         return new BeirutGreekPizza();
      else if(type.equals("pepperoni"))
         return new BeirutPepperoniPizza();
      return null;
   }
}
```

Example 2: Race

Suppose you are writing a class to represent a bicycle race, a race consists of many bicycle, i.e (bicycle for normal race, Road bicycle for the tourdeFrance race and Mountain Bicycle for Cyclocross race)

Bad Solution

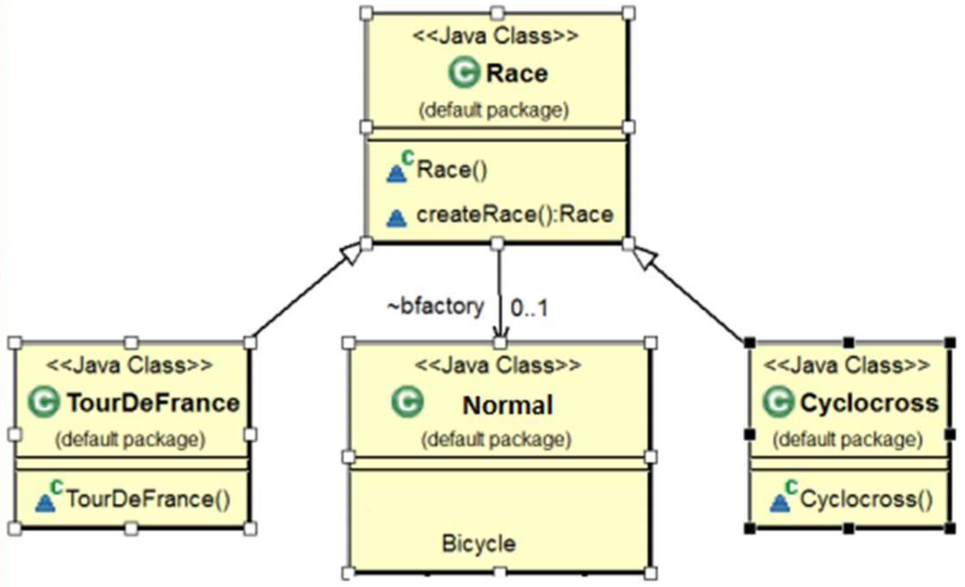


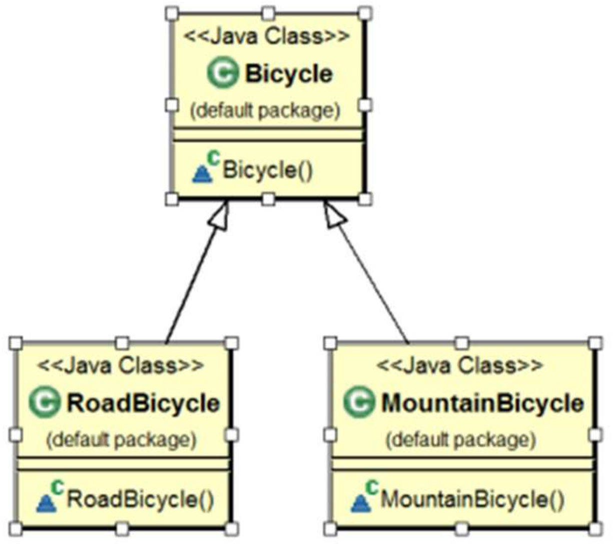
```
public class Race {
   List<Bicycle> bikes;

   ...

   public void createRace(String type) {
      if(type.equals("normal") {
         for(int i = 0; i < n; i++)
            bikes.add(new Bicycle());
      }
      else if(type.equals("tourdefrance") {
         for(int i = 0; i < n; i++)
            bikes.add(new RoadBicycle());
      }
      else if(type.equals("cyclocross") {
         for(int i = 0; i < n; i++)
            bikes.add(new MountainBicycle());
      }
   }
}
```

Example: Race - Solution Using Factory Method





Example: Race - Solution Using Factory Method

```java
public abstract class Race {
   List<Bicycle> bikes;

   // factory method
   abstract Bicycle createBicycle();

   public void createRace() {
      for(int i = 0; i < n; i++)
         bikes.add(createBicycle());
      }
      ...
   }
}
```

```java
public class NormalRace extends Race {
   @Override
   Bicycle createBicycle() {
      return new Bicycle();
   }
}
```

```java
public class TourDeFrance extends Race
   @Override
   Bicycle createBicycle() {
      return new RoadBicycle();
   }
}
```

```java
public class CycloCross extends Race {
   @Override
   Bicycle createBicycle() {
      return new MountainBicycle();
   }
}
```

```java
Race race = new TourDeFrance();
```

The Singleton pattern

Ensure a class only has one instance, and provide a global point of access to it.

A particular class should have only one instance. We will use only that instance whenever we are in need.

For example, your system can have only one window manager or a single point of access to a database engine or one RandomNumber generator, or one KeyboardReader,…

The Singleton pattern

Example :

Suppose you are a member of a football team. And in a tournament your team is going to play against another team. As per the rules of the game, the captain of each side must go for a toss to decide which side will be assigned for each team. So, if your team does not have a captain, you need to elect someone as a captain first. And at the same time, your team cannot have more than one captain.

The Singleton pattern

Example :

In this example, we have made the constructor private first, so that we cannot instantiate in normal fashion.

When we try to create an instance of the class, we are checking whether we already have one available copy.

If we do not have any such copy, we’ll create it; otherwise, we’ll simply reuse the existing copy.

The Singleton pattern-Implementation

```
class MakeACaptain
{    private static MakeACaptain _captain;
     //We make the constructor private to prevent the use of "new"
     private MakeACaptain() { }
     public static MakeACaptain getCaptain()
     {
          // Lazy initialization
          if (_captain == null)
          {
               _captain = new MakeACaptain();
               System.out.println("New Captain selected for our team");
          }
          else
          {
               System.out.print("You already have a Captain for your team.");
               System.out.println("Send him for the toss.");
          }
          return _captain;
     }
}
```

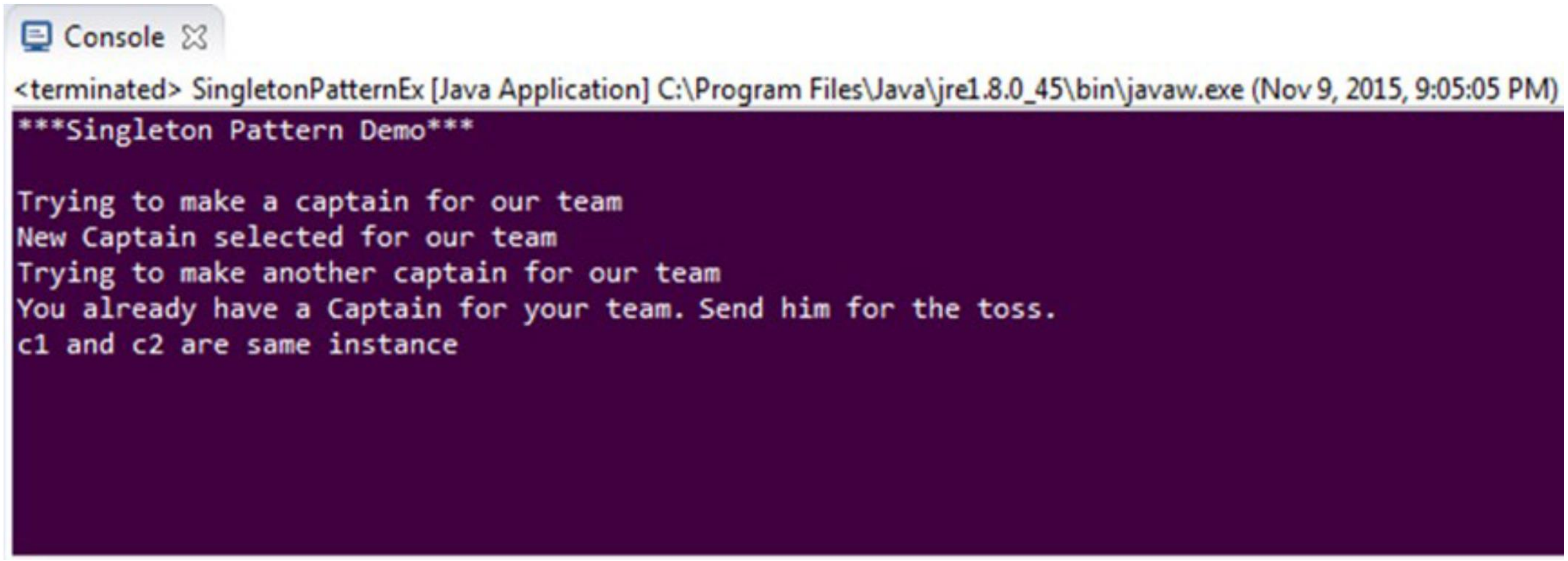
The Singleton pattern-Implementation

```
class SingletonPatternEx
{
    public static void main(String[] args)
    {
        System.out.println("***Singleton Pattern Demo***\n");

        System.out.println("Trying to make a captain for our team");
        MakeACaptain c1 = MakeACaptain.getCaptain();
        System.out.println("Trying to make another captain for our team");
        MakeACaptain c2 = MakeACaptain.getCaptain();

        if (c1 == c2)
        {
            System.out.println("c1 and c2 are same instance");
        }
    }
}
```

The Singleton pattern-Output



```
<terminated> SingletonPatternEx [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Nov 9, 2015, 9:05:05 PM)
***Singleton Pattern Demo***

Trying to make a captain for our team
New Captain selected for our team
Trying to make another captain for our team
You already have a Captain for your team. Send him for the toss.
c1 and c2 are same instance
```