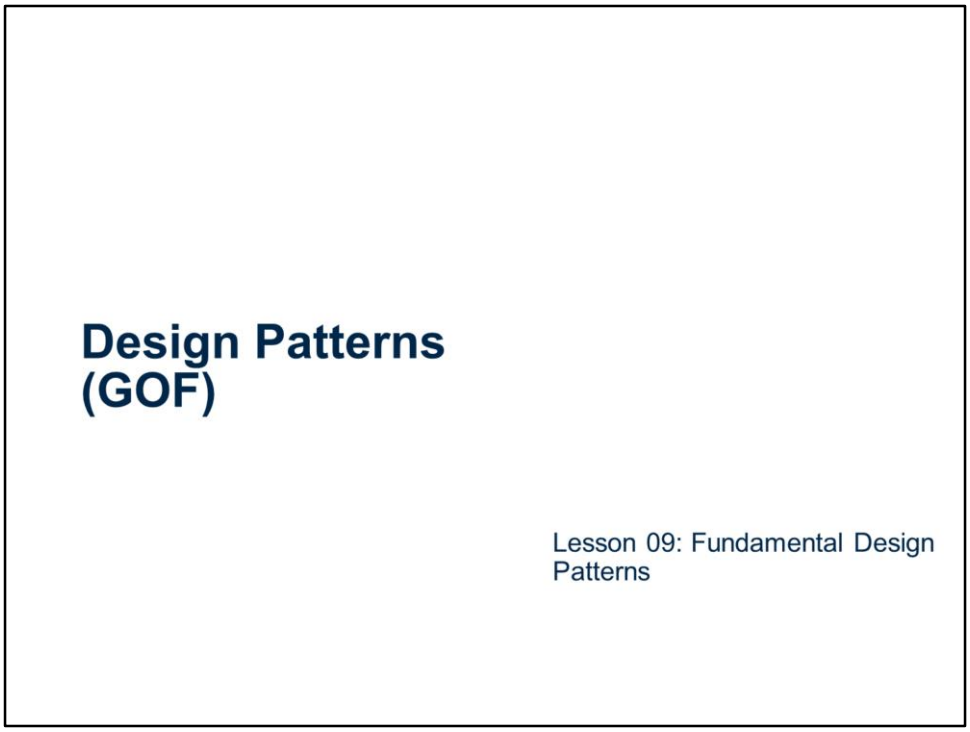
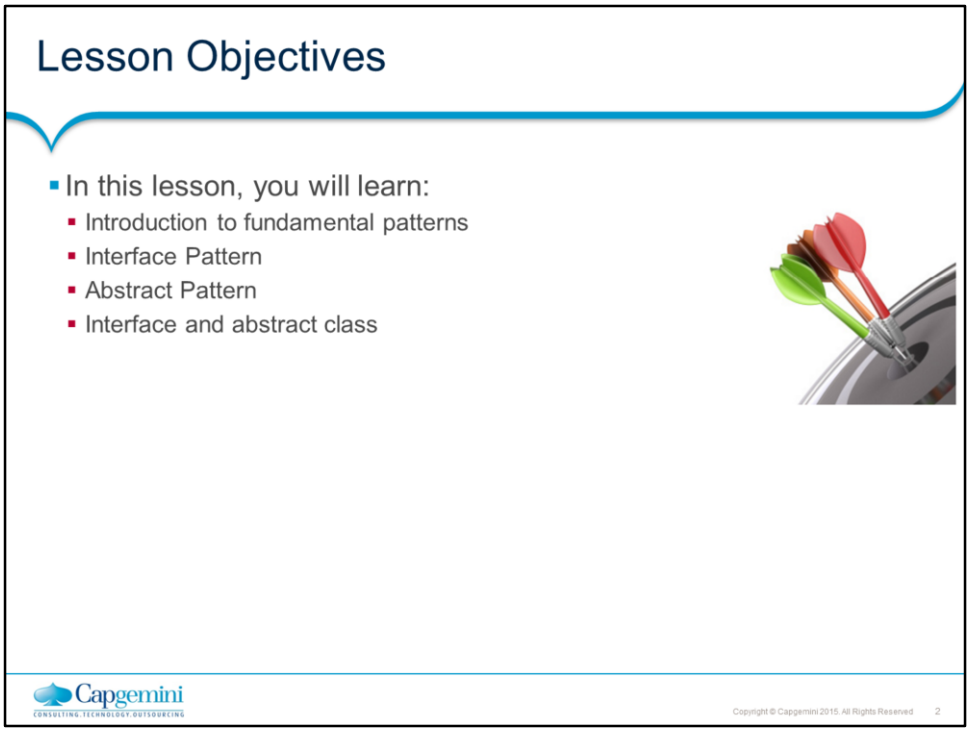
Instructor Notes:



Instructor Notes:

This lesson gives an Introduction on Fundamental Design Patterns

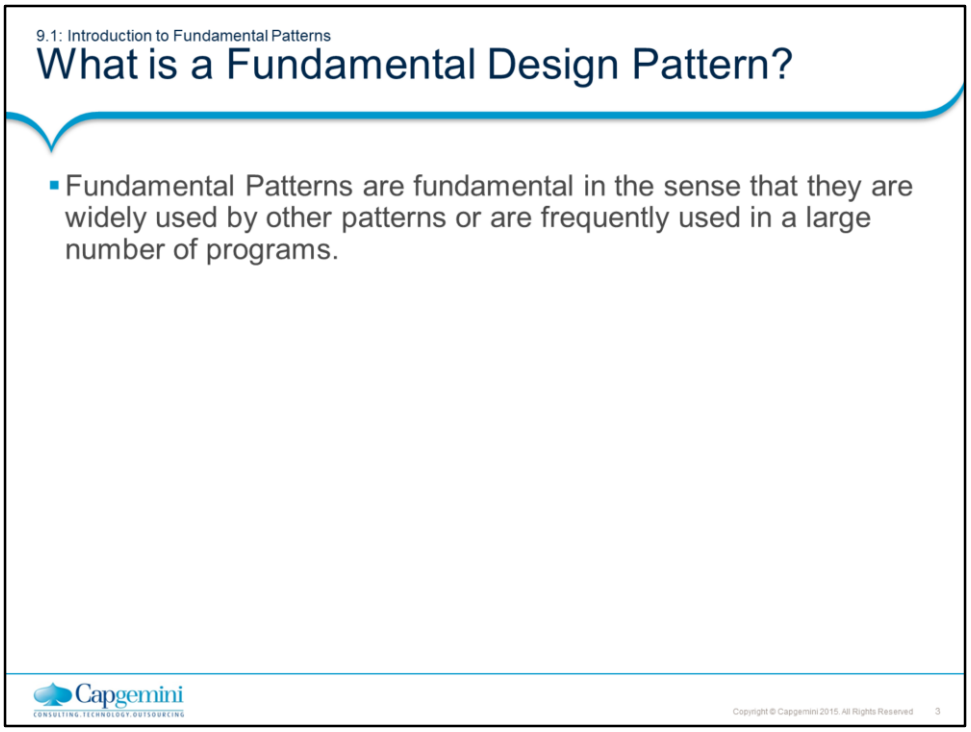


Lesson Objectives:
This lesson introduces GOF (Gang of Four) Design Patterns. The lesson contents are:
Lesson 09: Introduction to Fundamental Design Patterns

9.1: Introduction to Fundamental Patterns
9.2: Delegation Pattern
9.3: Interface Pattern
9.4: Abstract Pattern
9.5: Interface and Abstract Class
9.6: Introduction to Creational Patterns
9.7: Factory Method Pattern
9.8: Singleton Pattern

Instructor Notes:

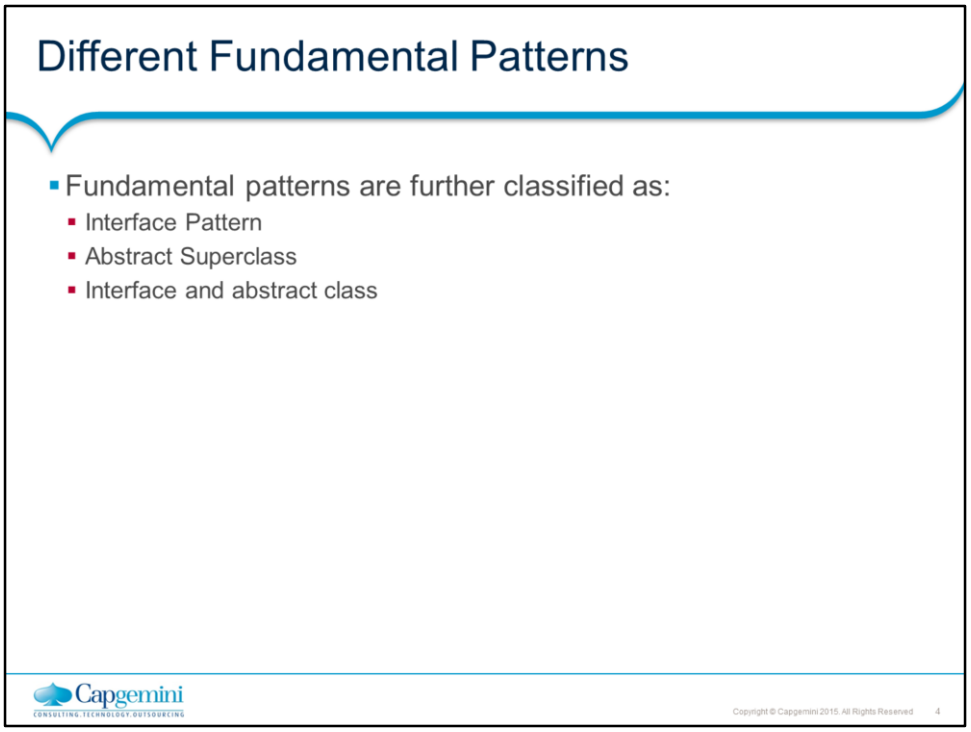
Explain the reason for including this lesson.



Note:
The fundamental design patterns are the building blocks of the other design patterns.

Instructor Notes:

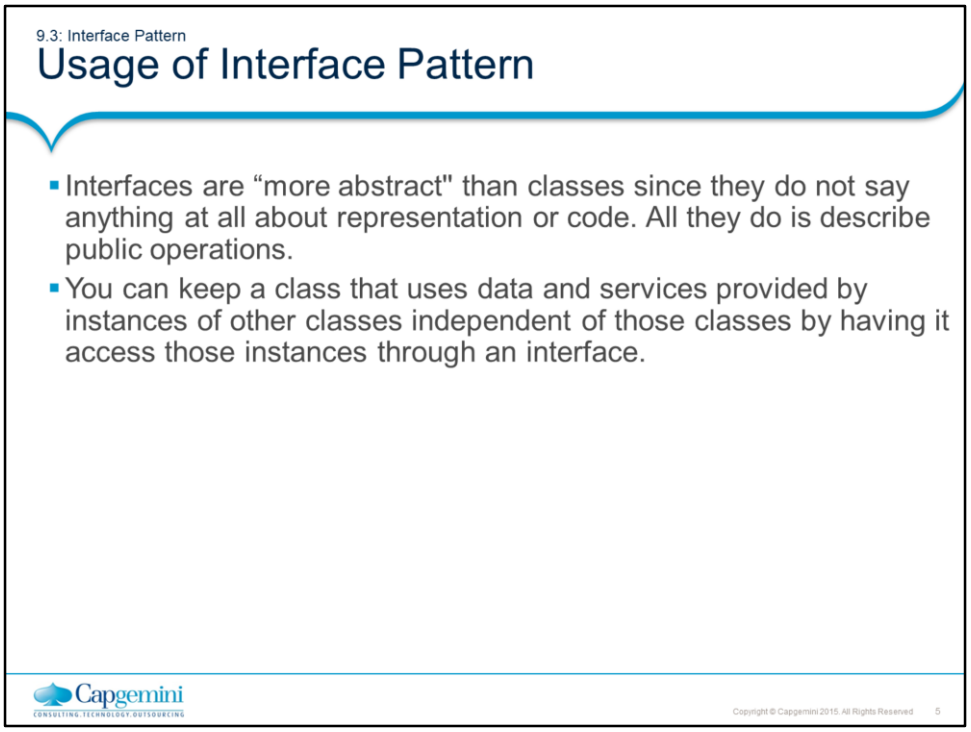
At this stage just mention the fundamental patterns and give a brief on significance of each of the Patterns.



Fundamental Design Patterns:
Delegation Pattern: It is a way of extending and reusing a class using composition rather than inheritance.
Interface Pattern: This pattern facilitates the design principle – “Program to Interface rather than Implementation”.
Abstract Superclass: It ensures consistent behavior of conceptually related classes by giving them a common abstract superclass.
Interface and abstract class: It is a combination of Interface and Abstract superclass Patterns.
Immutable Pattern: This pattern prevents the object from changing its state information after it is constructed thereby eliminating the issues related to concurrent access of the object.
Marker Interface: It is used to mark an object to be part of a particular group thereby allowing other objects to treat them in a uniform way.

Instructor Notes:

Interface Pattern supports the design principle – "Program to Interface and not to implementation"

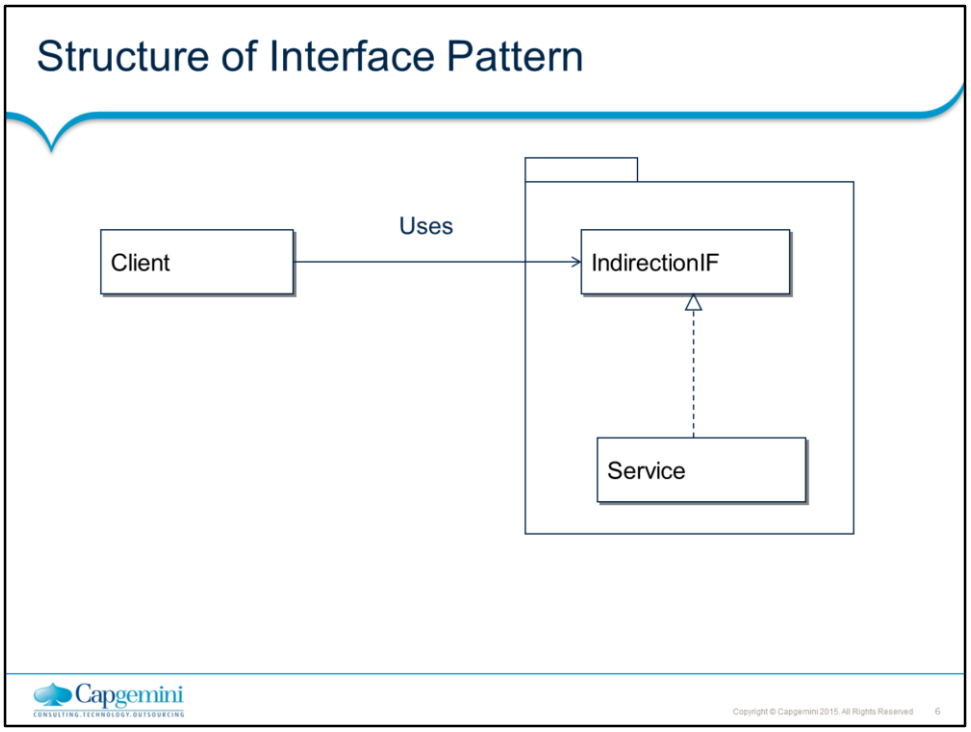


Interface Pattern:
A common problem is that you want an object to be able to use another object that provides a service without having to assume the class of the other object. The usual solution to the problem is to have the object assume that the object it uses implements a particular interface, rather than it being an instance of any particular class.
An interface encapsulates a coherent set of services and attributes, without explicitly binding this functionality to that of any particular object or code.
A class which uses other classes such as data and services should be built in such a way that the class should be independent of the kind of data and service objects it is going to deal with.

When to use:
An object relies on another object for data or services. If the object must assume that the other object upon which it relies belongs to a particular class, then the reusability of the object's class would be compromised.
You want to vary the kind of object used by other objects for a particular purpose without making the object dependent on any class other than its own.
Requirement is for generic functionality.

Instructor Notes:

This structure is very simple and must have been used by all developers at some point or the other. Check with them their experience or reason for using this pattern.



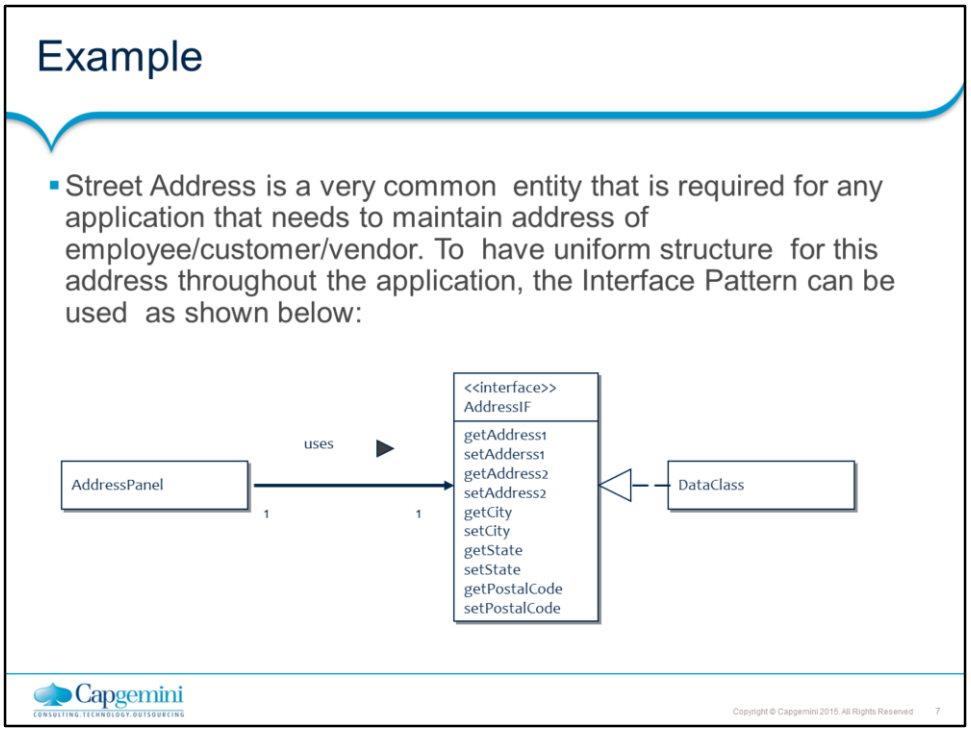
Structure of Interface Pattern:
Following are the roles played by these classes and interfaces:
Client: The Client class uses classes that implement the IndirectionIF interface.

IndirectionIF: The IndirectionIF interface provides indirection that keeps the Client class independent of the class that is playing the Service role. Interfaces in this role are generally public.

Service: Classes in this role provide a service to classes in the client role. Classes in this role are ideally private to their package. Making Service classes private forces classes outside of their package to go through the interface. However, it is common to have implementations of an interface that are in different packages.

Instructor Notes:

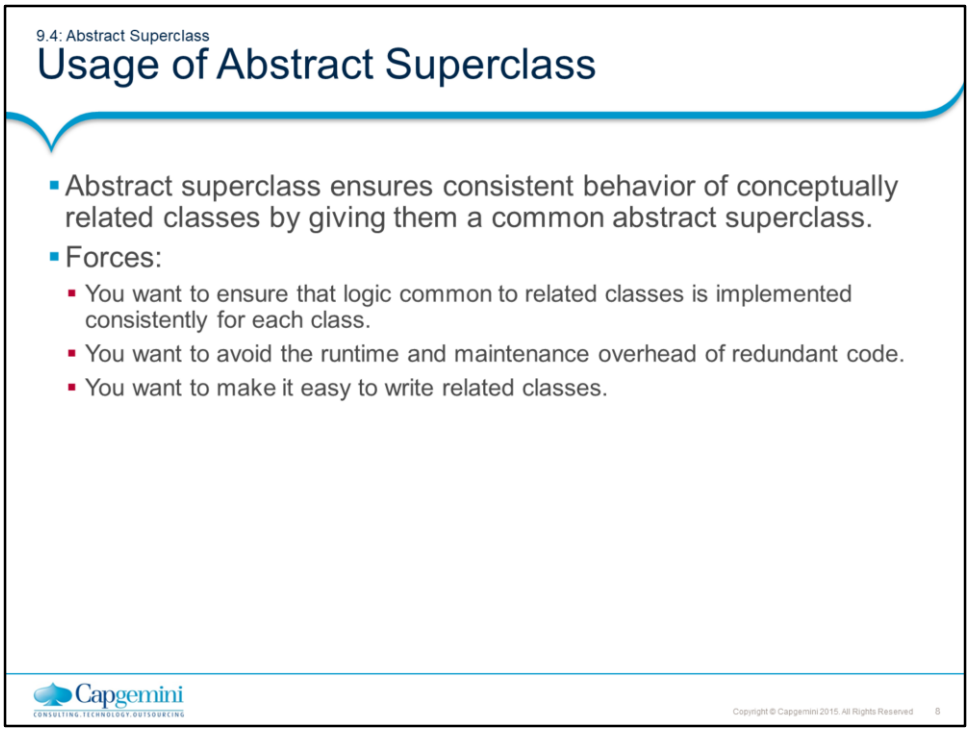
Discuss the example on the slide.



Interface Pattern:
To avoid classes having to depend on other classes because of a uses/usedby relationship, make the usage indirect through an interface.
For example, suppose you are writing an application to purchase goods for a business. Your program will need to be informed of such entities as vendors, freight companies, receiving locations, and billing locations. One thing these have in common is that they all have a street address. These street addresses will be displayed in different parts of the user interface. You will want to have a class for displaying and editing street addresses so that you can reuse it wherever there is an address in the user interface. We will call that class AddressPanel.
You want AddressPanel objects to be able to get and set address information in a separate data object. This raises the question of what instances of the AddressPanel class can assume about the class of the data objects that they will use. Clearly, you will use different classes to represent vendors, freight companies, and the like. If you program in a language that supports multiple inheritance, like C++, you can arrange for the data objects that instances of AddressPanel use to inherit from an address class in addition to the other classes they inherit from. If you program in a language like Java that uses a single inheritance object model, then you must explore other solutions.
You can solve the problem by creating an address interface. Instances of the AddressPanel class would then simply require data objects that implement the address interface. They would then be able to call the accessor methods of that object to get and set its address information. Using the indirection that the interface provides, instances of the AddressPanel class are able to call the methods of the data object without having to be aware of what class it belongs to. Here is a class diagram showing these relationships.
Here in this example we have seen that the AddressPanel class is independent or does not have to specifically assume the data object it is going to deal with. In turn, it will work upon the data object with the help of the AddressIF interface which is actually implemented by the data classes.

Instructor Notes:

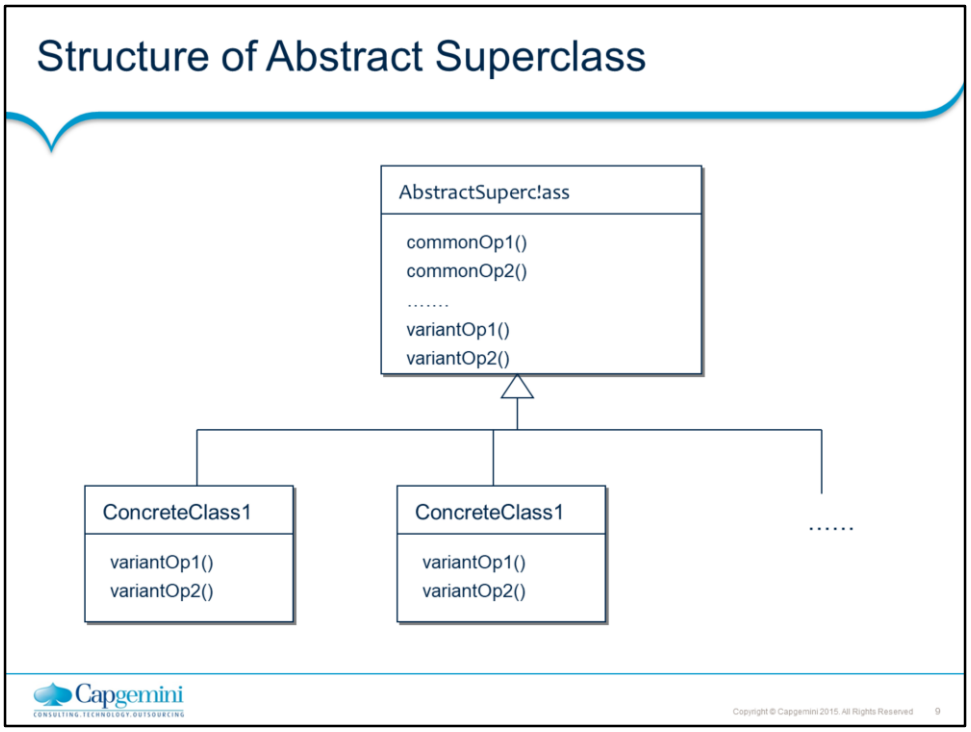
This pattern is also very obvious and developers/designers would have used it without knowing that it is a pattern.



Abstract Superclass:
The Abstract superclass helps in maintaining consistency in behavior across the sub classes.
Whatever is common is separated out in an abstract class and all the related classes are made to inherit from this abstract class so that they inherit this common behavior.

Instructor Notes:

This structure is well-known for any OO programmer. Ask them to interpret.

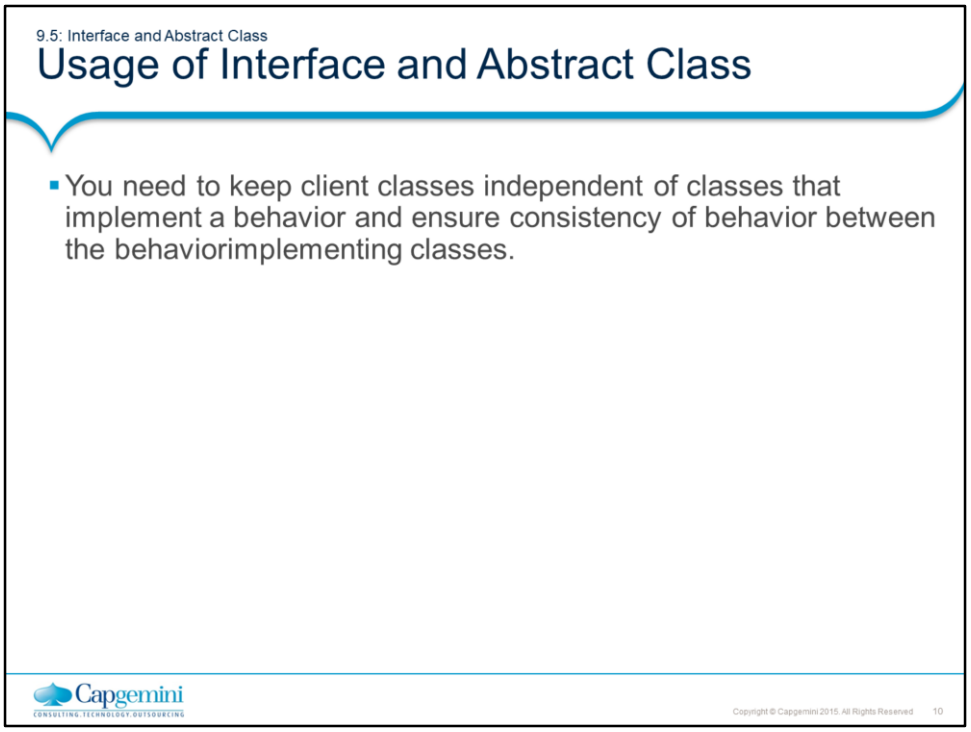


Abstract Superclass:
Organize the common behavior of related classes into an abstract superclass. To the extent possible, organize variant behavior into methods with common signatures. Declare the abstract superclass to have abstract methods with these common signatures. The above figure shows this organization.
Following are the roles that classes play in the Abstract Superc!ass pattern:

AbstractSuperclass:
A class in this role is an abstract superclass that encapsulates the common logic for related classes. The related classes extend this class so they can inherit methods from it. Methods whose signature and logic are common to the related classes are put into the superclass so their logic can be inherited by the related classes that extend the superclass. Methods with different logic but the same signature are declared in the abstract class as abstract methods, ensuring that each concrete subclass has a method with those signatures.
ConcreteClassl, ConcreteClass2, and so on:
A class in this role is a concrete class whose logic and purpose is related to other concrete classes. Methods common to these related classes are refactored into the abstract superclass. Common logic that is not encapsulated in common methods is refactored into common methods.

Instructor Notes:

This is the combination of Interface & Abstract super class patterns. The behavior that is common for subclasses goes in abstract class and the behavior that varies amongst subclasses will be declared in the Interface so that the subclass themselves can implement this varied behavior.

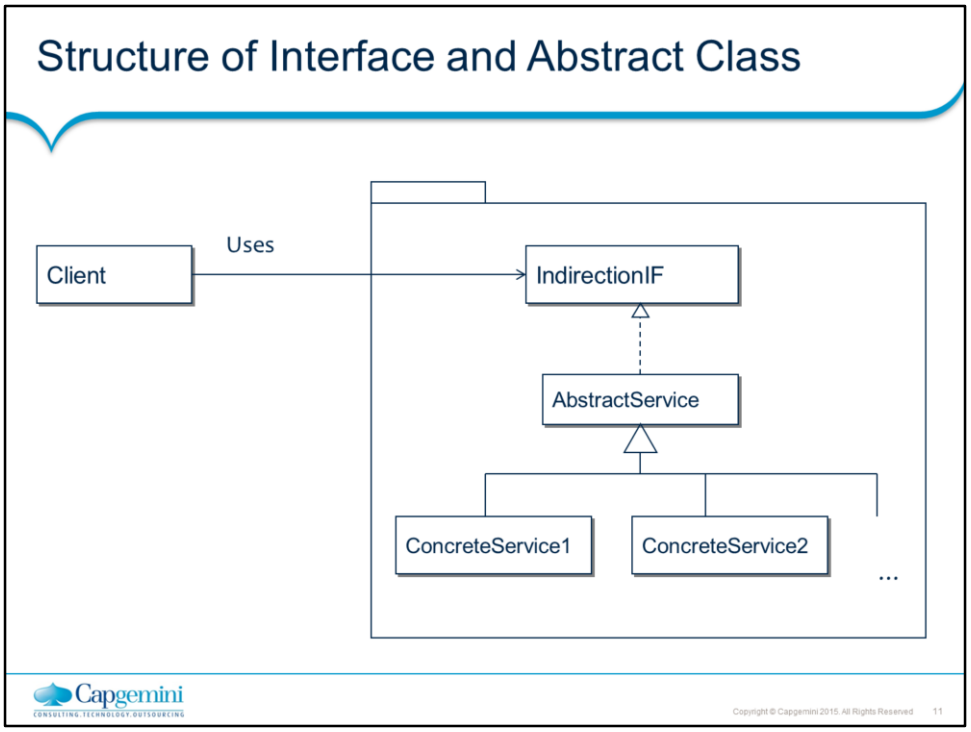


Interface and Abstract Class:
You need to keep client classes independent of classes that implement a behavior and ensure consistency of behavior between the behavior-implementing classes. Do not choose between using an interface and an abstract_ class. Have the classes implement an interface and extend an abstract class.
Forces:

Using the Interface pattern, Java interfaces can be used to hide the specific class that implements a behavior from the clients of the class. Organizing classes that provide related behaviors with a common superclass helps to ensure consistency of implementation. Through reuse, it may also reduce the effort required for implementation.

Instructor Notes:

Again a well-known pattern, though developers may be using it unknowingly

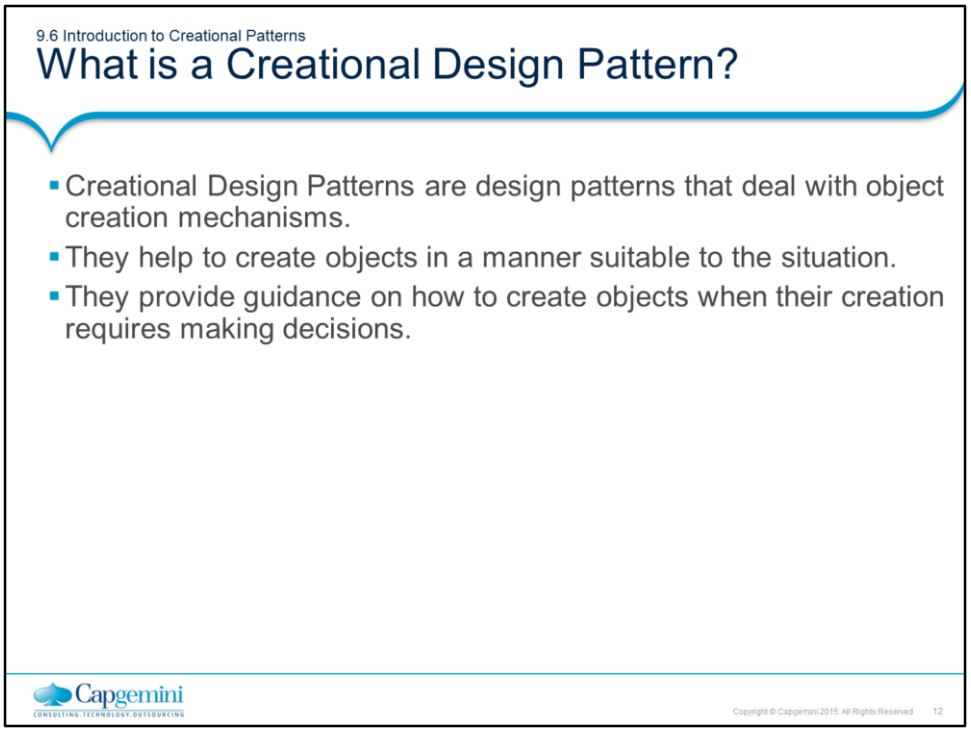


Interface and Abstract Class:
The main intention here is to hide the classes that implement some behavior making the classes private to the package by having them implement a public interface.

By isolating client from service through Interface
By isolating Abstraction from implementation through Abstract class

Instructor Notes:

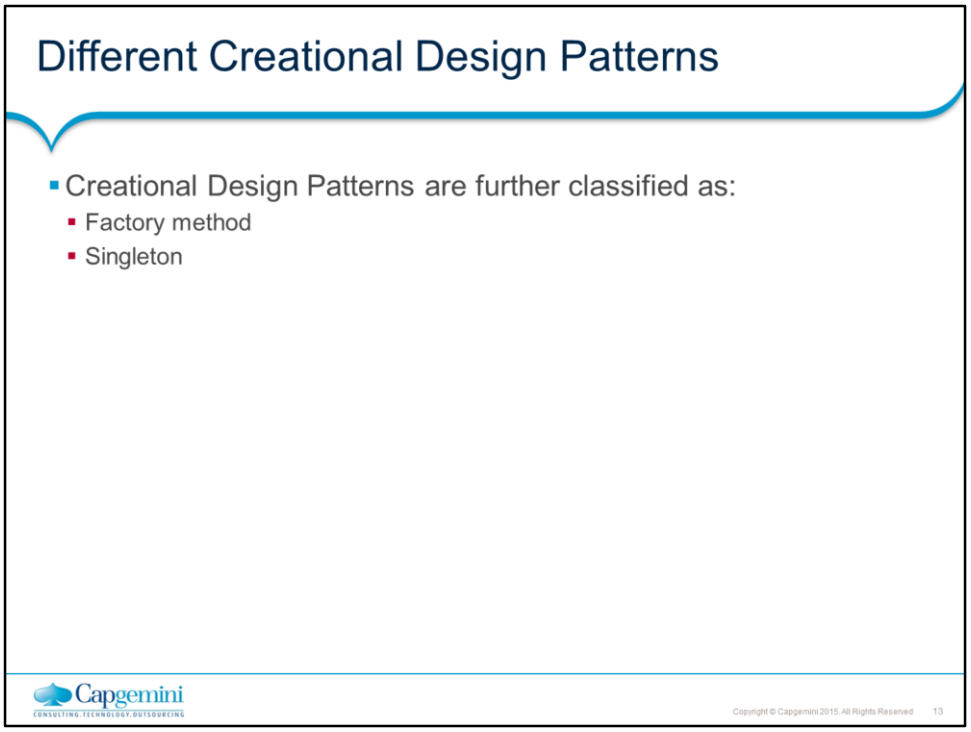
Creational Patterns are all about class instantiation. They can be further divided into class-creational pattern & object creational pattern. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.



Introduction to Creational Patterns:
Creational Patterns provide guidelines on creation, configuration, and initialization for objects.
“Decisions typically involve dynamically deciding which class to instantiate or which objects an object will delegate responsibility to.”

Instructor Notes:

At this stage just mention the list of creational Patterns and explain signification of each of them.



Different Creational Patterns:
Creational Design Patterns are further classified as:

Factory Method: It creates objects without specifying the exact object to create.

Abstract Factory: It groups object factories that have a common theme.

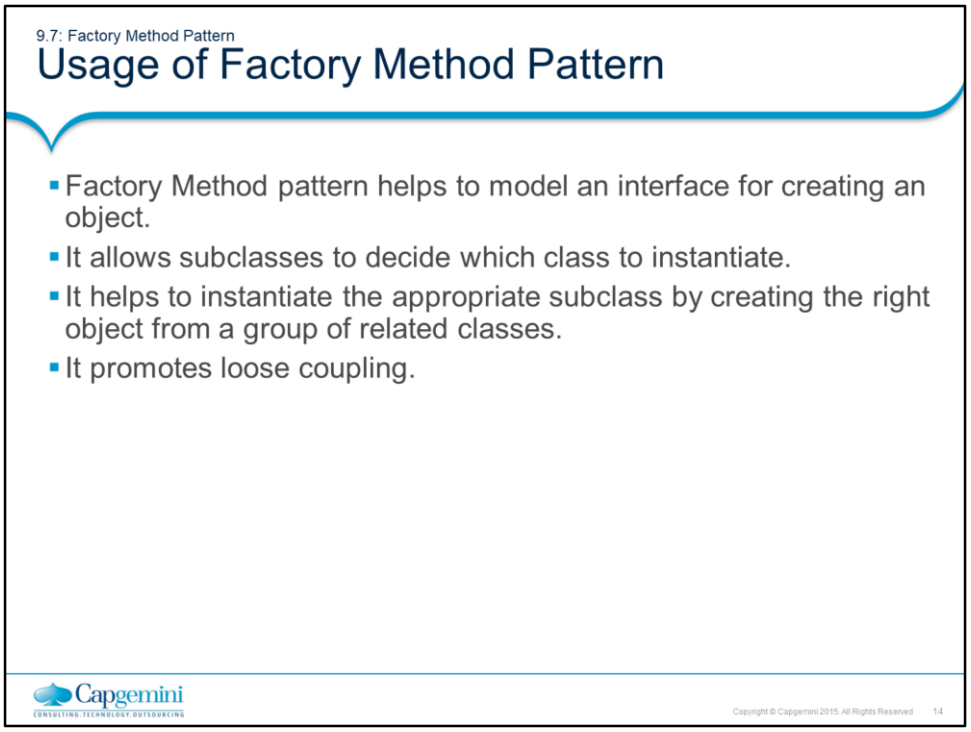
Prototype: It creates objects by cloning an existing object.

Builder: It constructs complex objects by separating construction and representation.

Singleton: It restricts object creation for a class to only one instance.

Instructor Notes:

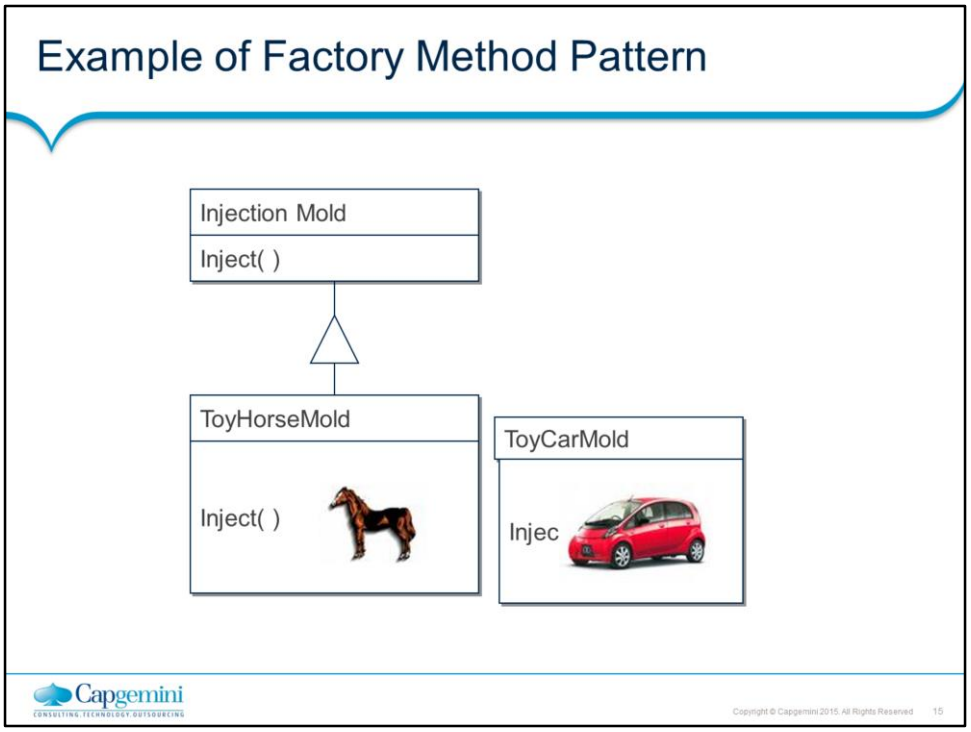
The factory method pattern delegates the work of object creation to derived classes of the interface.



Factory Method:
You can define an interface for creating an object, however, let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
It is also called as a Factory Pattern since it is responsible for “manufacturing” an Object.
The Factory Method pattern is an object-oriented design pattern. Like other creational patterns, it deals with the problem of creating objects (products) without specifying the exact class of object that will be created. Factory Method handles this problem by defining a separate method for creating the objects, which can then be overridden by subclasses to specify the derived type of the product that will be created.
They promote loose coupling by eliminating the need to bind application specific classes into the code.

Instructor Notes:

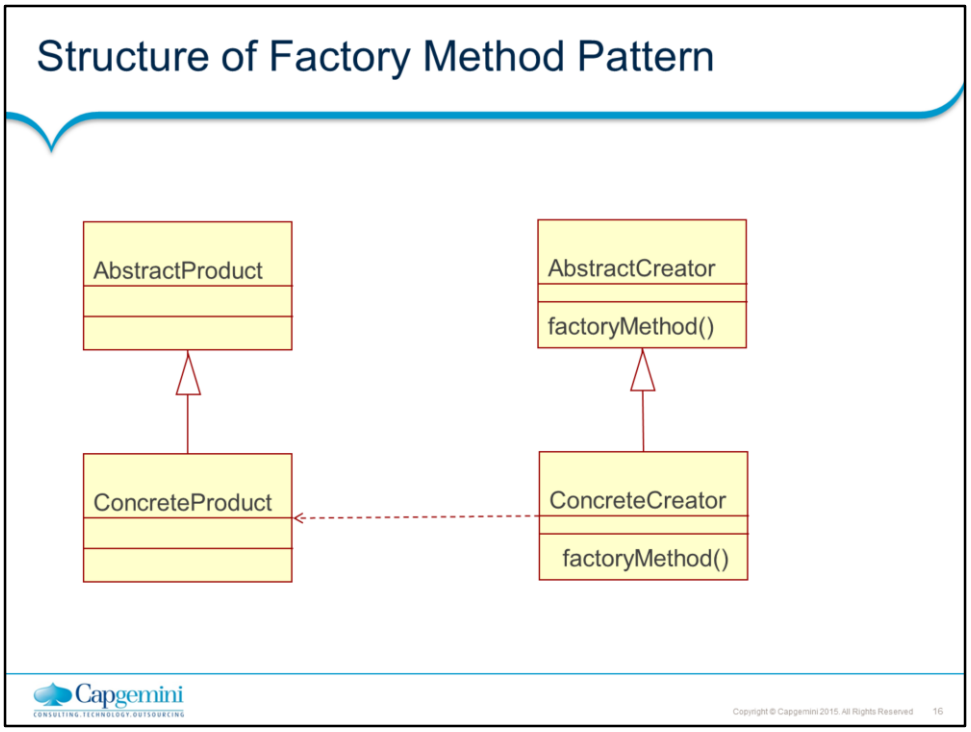
Discuss this real-life example with participants. Ask them to share their experiences.



Example of Factory Method Pattern:
The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate.
Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The class of toy (car, action figure, etc.) is determined by the mold.

Instructor Notes:

Factory methods eliminate the need to bind application – specific classes into your code. The code only deals with the Product interface; therefore it can work with any user-defined ConcreteProduct classes.



Structure of Factory Method Pattern:
Structure Elements:
AbstractProduct: It defines the interface of objects that the factory method creates.
ConcreteProduct: It implements/extends the AbstractProduct interface/class.
AbstractCreator:

- It declares the factory method, which returns an object of type AbstractProduct.
- It may also define a default implementation of the factory method that returns a default ConcreteProduct object.
- It may call the factory method to create a Product object.

ConcreteCreator: It overrides the factory method to return an instance of a ConcreteProduct.

Instructor Notes:

Explain the problem statement and ask the participants to come up with a solution without applying any pattern.

Example of Factory Method Pattern

- Suppose we have an entry form and we want to allow the user to enter his/her name either as “first name last name” or as “last name, first name”. Name order is always decided by whether comma is in between the last and first name.

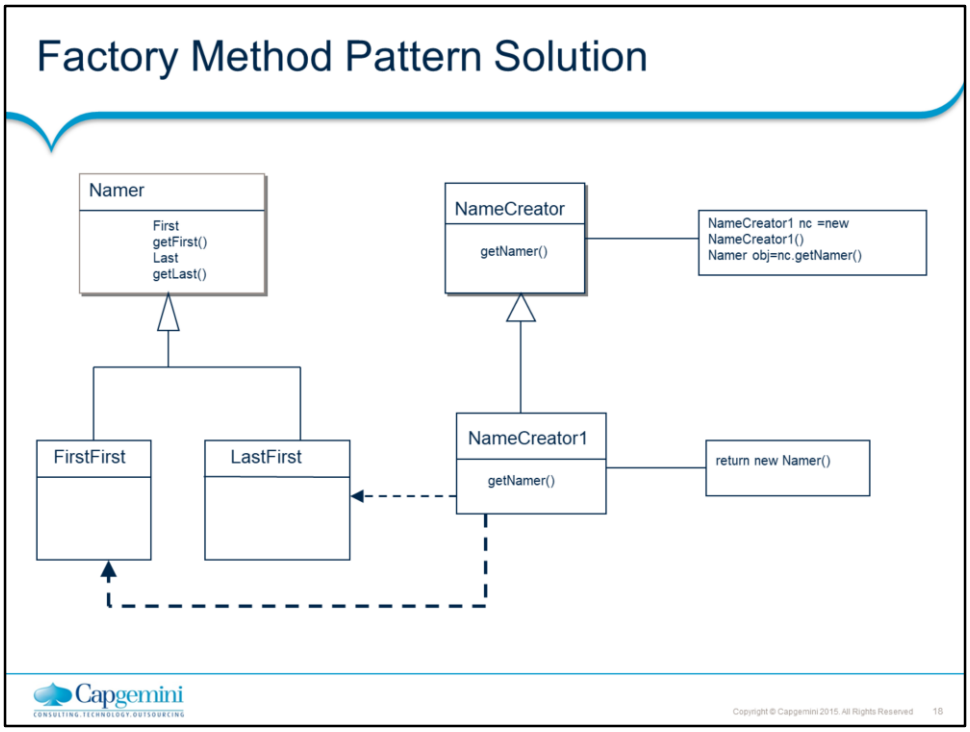
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 17

What will be your approach?

Instructor Notes:

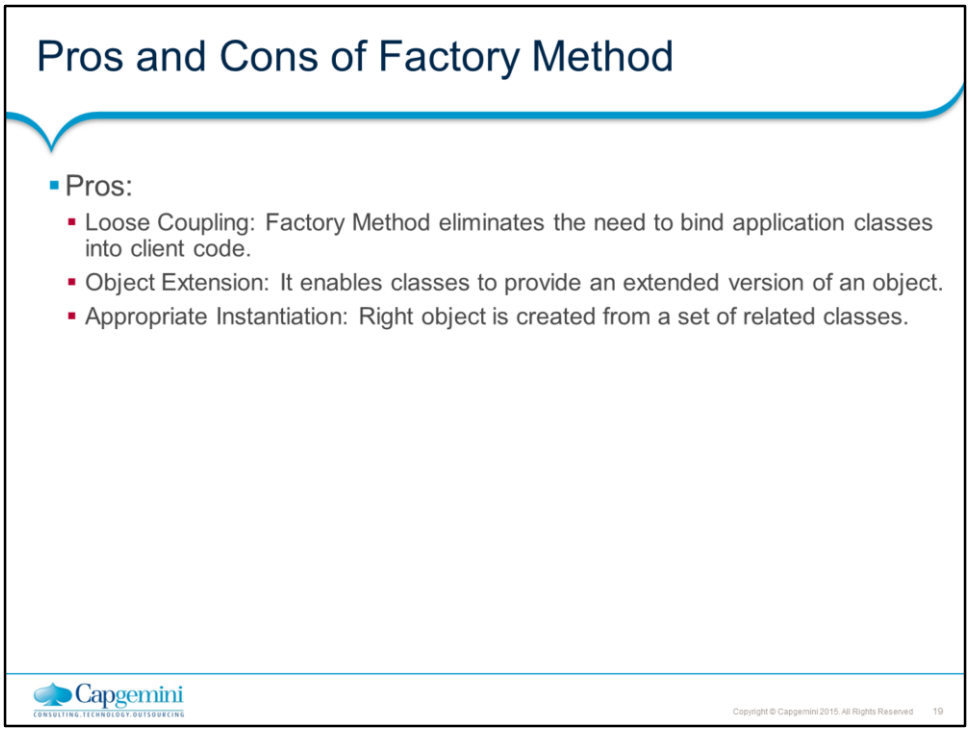
Discuss the solution using Pattern and advantage of using the same.



Factory Method Pattern Solution:
Product: Namer class
Concrete Product: FirstFirst and LastFirst classes
Creator: NameCreator class
Concrete Creator: NameCreator1 class

Instructor Notes:

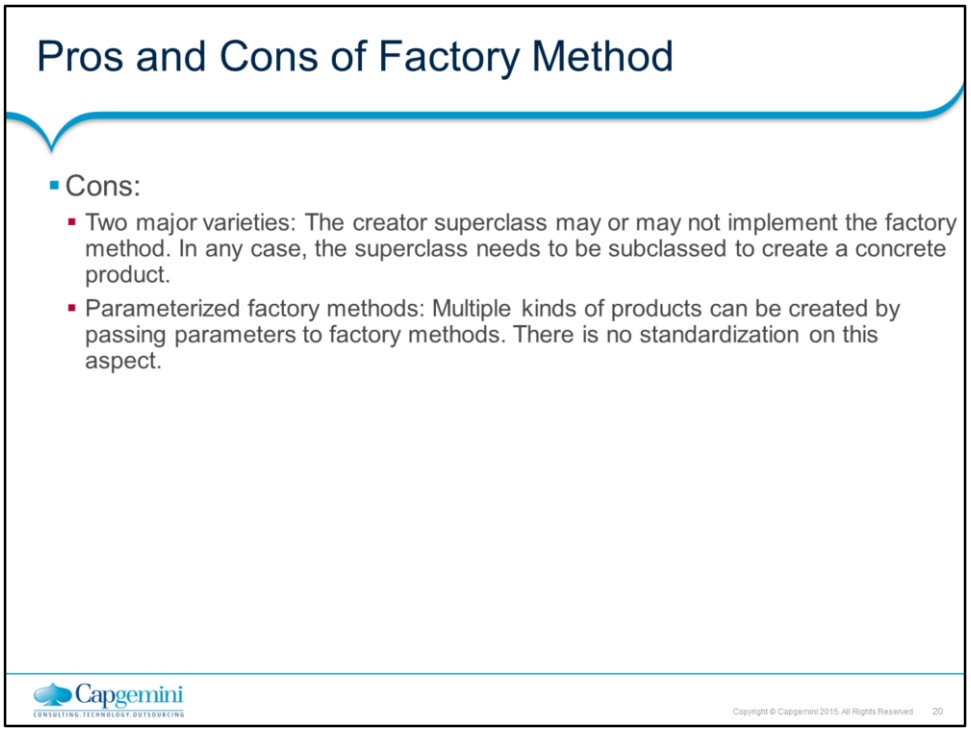
Discuss the pros of Factory method pattern.



Pros and Cons of Factory Method:
Pros:
Loose Coupling: Factory Method eliminates the needs to bind application classes into client code. The code deals with any classes that implement a certain interface. Application becomes more flexible and reusable.
Object Extension: It enables the subclasses to provide an extended version of an object.
Appropriate Instantiation: It helps to instantiate the appropriate subclass by creating the right object from a group of related classes.

Instructor Notes:

Discuss the cons of Factory method pattern.



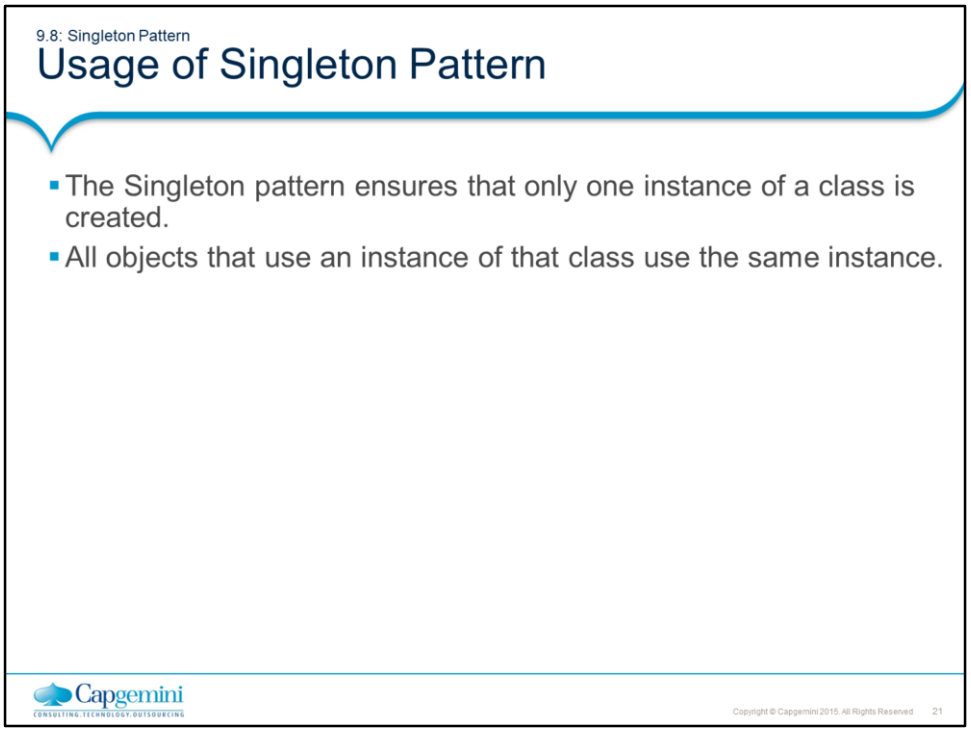
Pros and Cons of Factory Method:
Cons:
Two major varieties: The two main variations of the Factory Method pattern are: (1) the case when the Creator class is an abstract class and does not provide an implementation for the factory method it declares, (2) the case when the Creator is a concrete class and provides a default implementation for the factory method. A potential disadvantage of factory methods is that clients might have to subclass the Creator class just to create a particular ConcreteProduct object.
Parameterized factory methods: Another variation on the pattern lets the factory method create multiple kinds of products. The factory method takes a parameter that identifies the kind of object that has to be created. All objects that the factory method creates share the Product interface. In the Namer example, Namer might support different kinds of naming mechanisms. You pass getNamer an extra parameter to specify the kind of Naming mechanism.

Instructor Notes:

This is the simplest pattern to understand and the most misused pattern. The challenge is to understand where it can be applied.

A singleton is essentially a global variable. Use it when its is simpler & safer to pass an object resource as a reference to the object that needs it, rather than letting objects access the resource globally.

This pattern makes unit testing difficult , as it introduces “global state” into an application. Access to Singletons in a multithreaded environment must be synchronized.



Singleton Pattern:
There are cases in programming where you need to ensure that there can be one and only one instance of a class which is used by the application.
The Singleton pattern ensures that a class has only one instance, and provides a global point of access to it.
It is important for some classes to have exactly one instance.

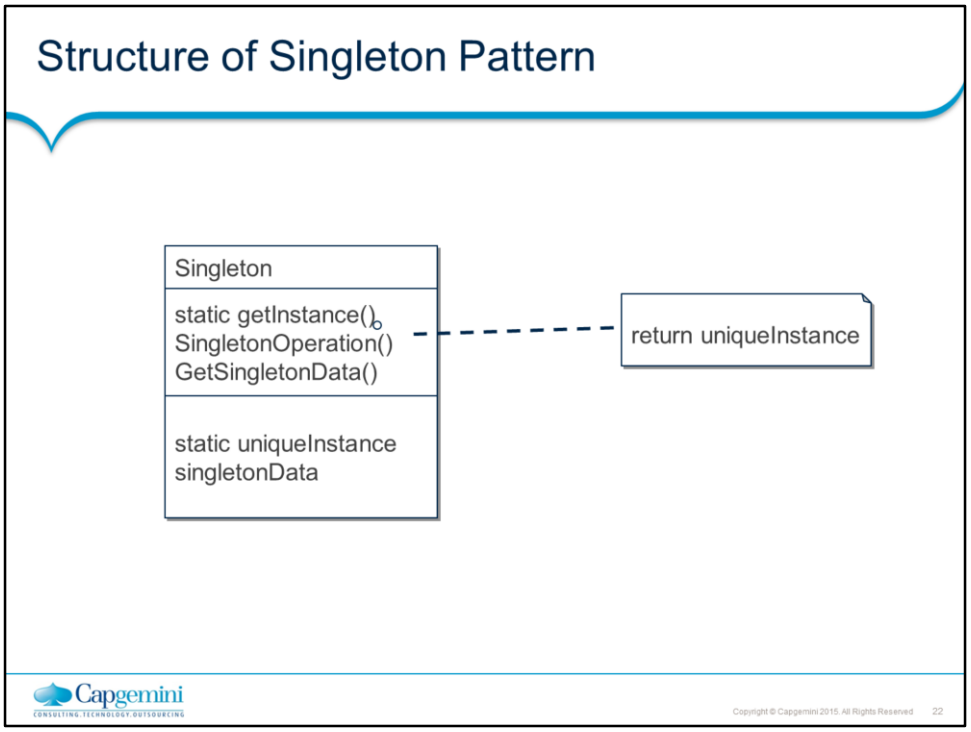
- Although there can be many printers in a system, there should be only one printer spooler.
- There should be only one File system and one Window manager.
- A digital filter will have one A/D converter.
- An accounting system will be dedicated to serving one company.

How do we ensure that a class has only one instance and that the instance is easily accessible?

- A global variable makes an object accessible. However, it does not keep you from instantiating multiple objects.
- A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance. This is the Singleton pattern.

Instructor Notes:

Discuss the structure & implementation of the Singleton Pattern. It is one of the easiest to understand.



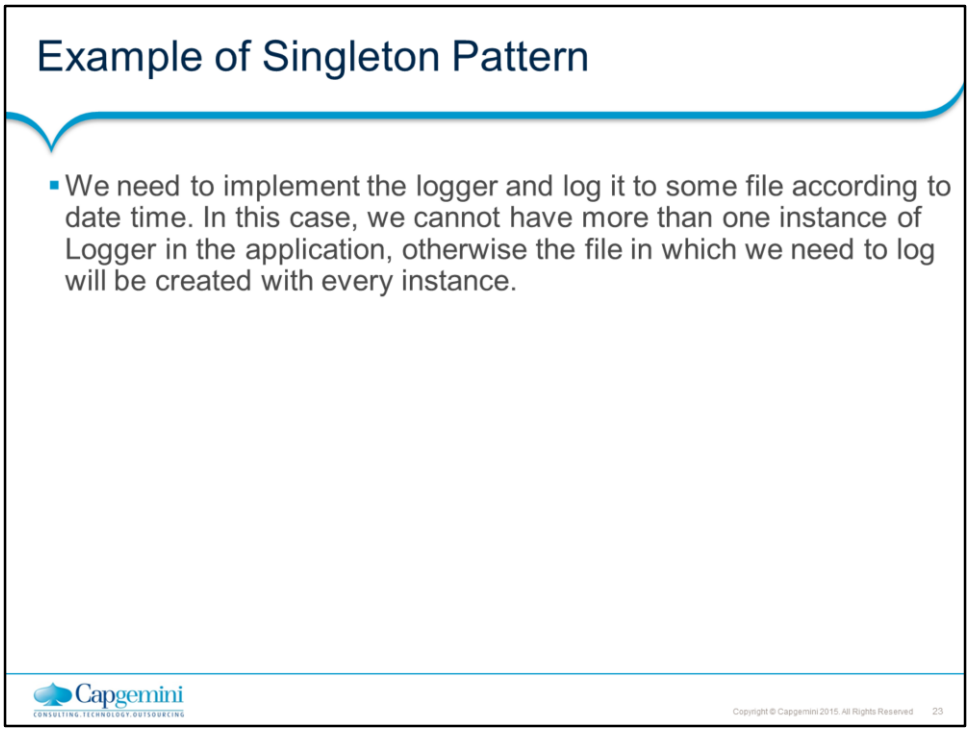
Structure of Singleton Pattern:
Structure Elements:
Singleton:

It defines a getInstance operation that lets clients access its unique instance. getInstance is a class operation (i.e. static method)
It may be responsible for creating its own unique instance.
It should have a private constructor so that the client cannot creates its instance using “new” keyword.

In case of multiple class loaders, the Singleton Implementation will be difficult as each class loader will have one instance of the Singleton class which is as good as having multiple instances of the same class.

Instructor Notes:

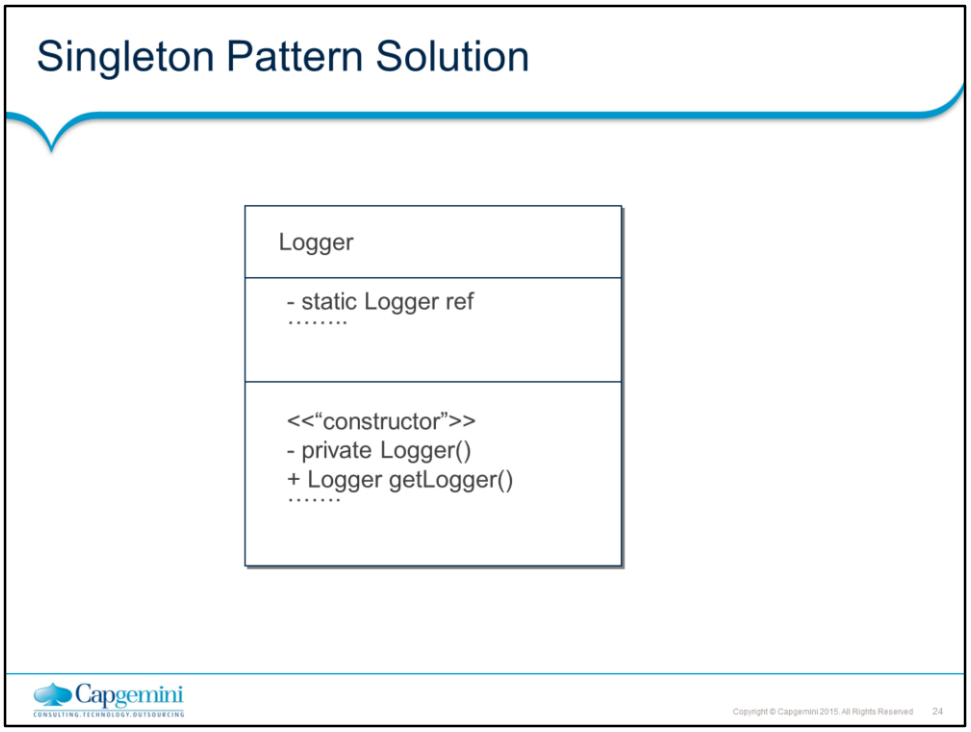
Discuss this scenario and ask the participants to suggest design solution.



How can you implement this logger?

Instructor Notes:

Discuss this scenario and ask the participants to suggest design solution.



Solution:

Singleton: Logger

Logger class needs to be a singleton. This is because application uses the single Logger to log the information. This can be achieved by restricting an application from creating multiple instances, by providing private constructor.

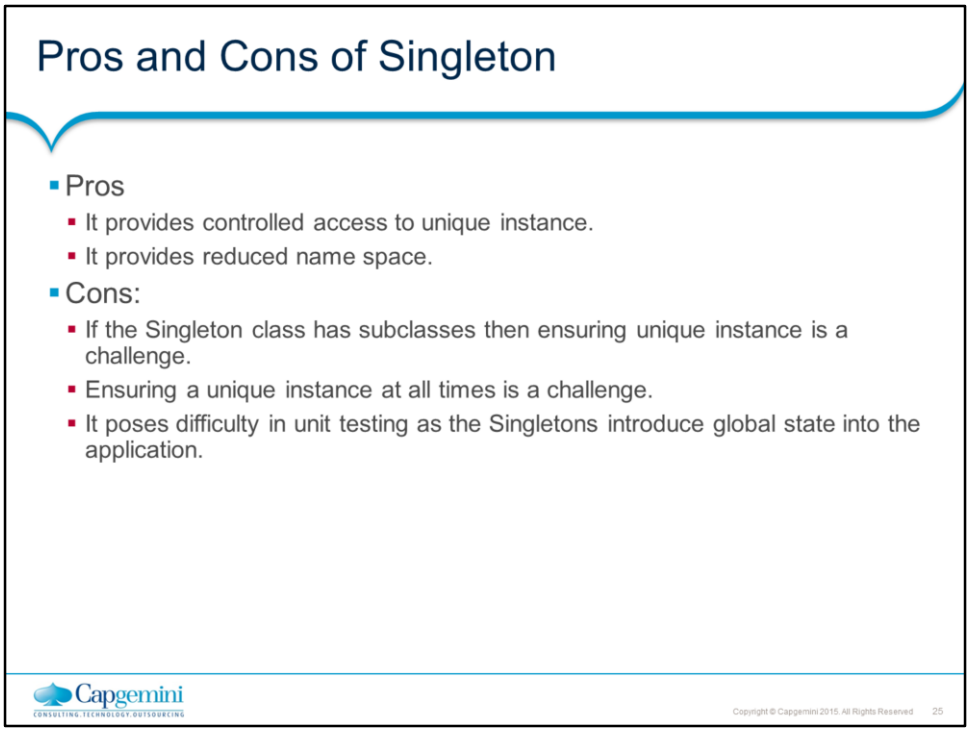
To ensure that the instance is created the first time and later the same instance is used by the application, getLogger() method is defined. You also need to provide the static instance variable of Logger class to make it global.

Instructor Notes:

As mentioned earlier this is the most misused pattern. The pattern itself has no problems, the problem is due to the way this pattern is used by the developers.
Why is Singleton should not be used:
Answer: Singleton

a. Hides dependencies
b. Hard to test
c. Hard to subclass
d. It's a lie (in case of multiple class loaders). We can have one instance per Class loader but not one instance per VM.
e. A singleton today is a multiple tomorrow.

So how do we avoid it?
Instead of using Singleton wherever you do not want the clients to create instances of a object and ensure that all clients use the same instead, in such a case create the instance using Factory Method pattern and then pass this instance to the client (object) that needs it as dependency injection.



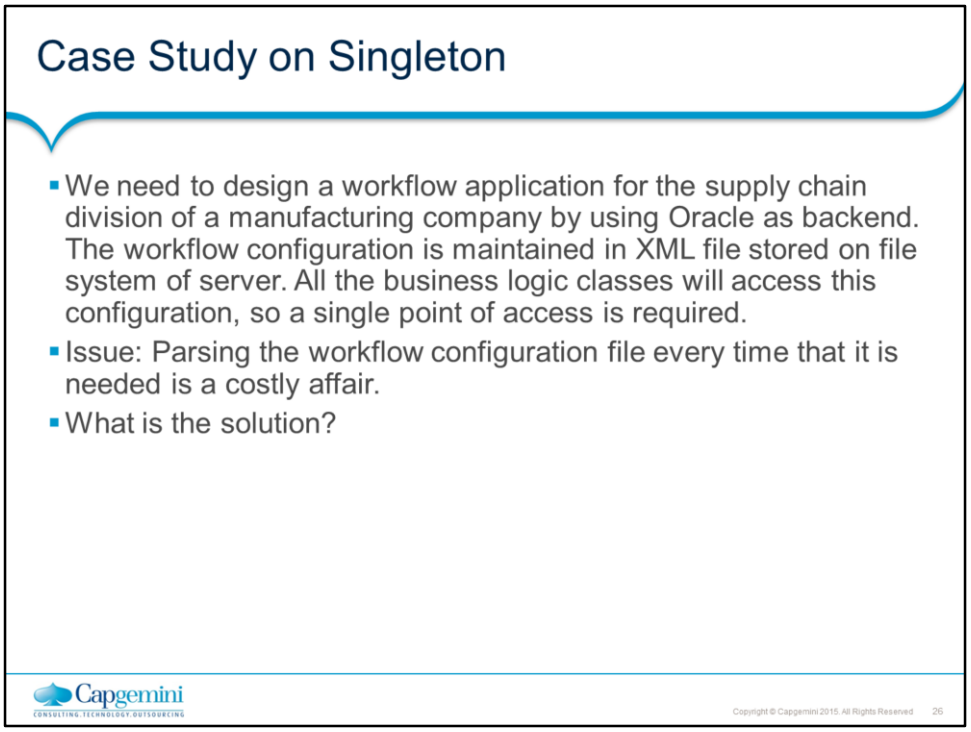
Pros and Cons of Singleton:
Pros:
Controlled access to sole instance: Since the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.

Reduced name space: The Singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables that store sole instances.

Cons:
Subclassing the Singleton class: The main issue is not in defining the subclass but ensuring its unique instance so that clients will be able to use it. In other words, the variable that refers to the singleton instance must get initialized with an instance of the subclass.
Ensuring a unique instance: In case of multiple class loaders, it is difficult to ensure single instance.
Hard to test: Singleton introduces global state into an application and hence it is difficult to unit test it. Also the hidden dependency of users on a singleton makes testing very difficult as there is no way to mock out or inject a test instance of the singleton. Also the state of the singleton affects the execution of a suite of tests such that they are not properly isolated from each other.

Instructor Notes:

Discuss case study and its solution. Also, discuss the alternative to using Singleton Pattern for this scenario.



Solution:
The configuration file can be loaded only once when the application is deployed.
This can be implemented by using the Singleton Pattern. You can create a class that will load the XML file and make use of the configuration settings in the XML file. You will have to ensure that only one instance of this class (workflow configuration class) is created using Single ton Pattern. All the business logic classes will then need to make use of this single instance.