DESIGN PATTERNS

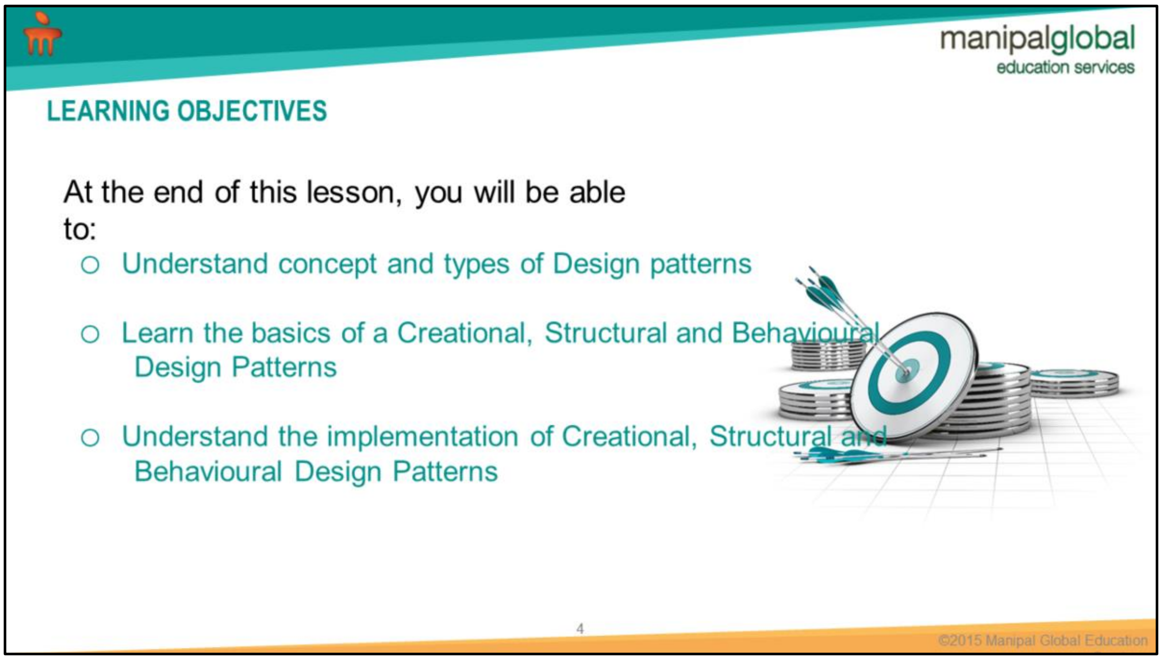
manipalglobal
education services

INTRODUCTION
Design Patterns

manipalglobal
education services

OBJECTIVES
Design Patterns

3

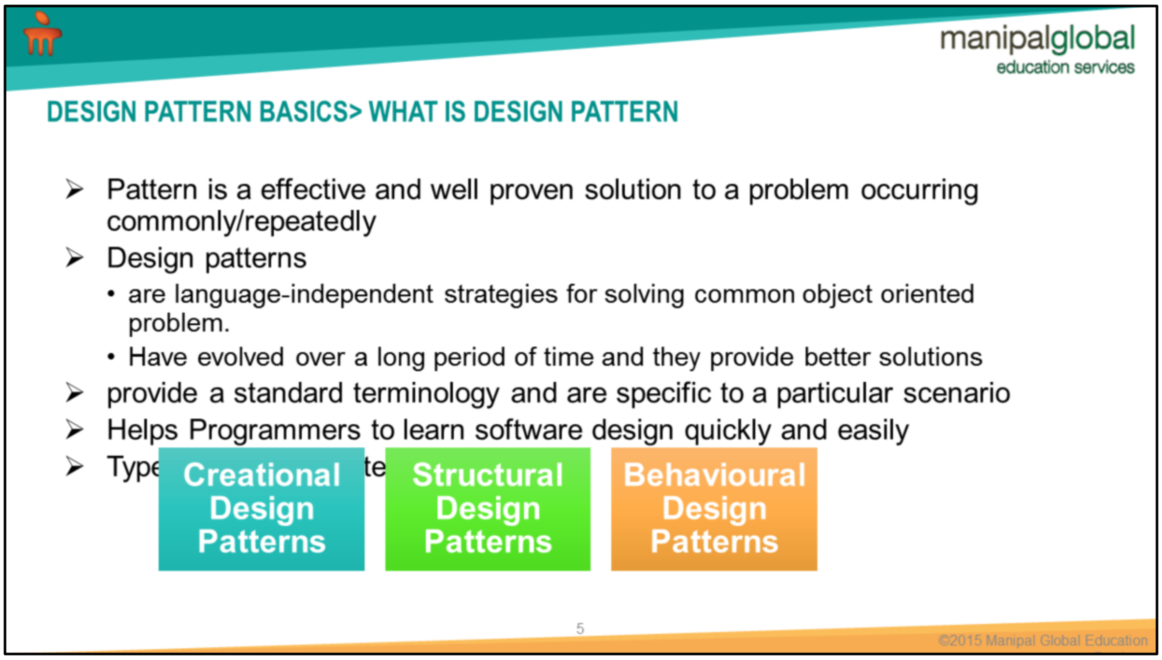


manipalglobal
education services

LEARNING OBJECTIVES

At the end of this lesson, you will be able to:

- Understand concept and types of Design patterns
- Learn the basics of a Creational, Structural and Behavioural Design Patterns
- Understand the implementation of Creational, Structural and Behavioural Design Patterns



manipalglobal
education services

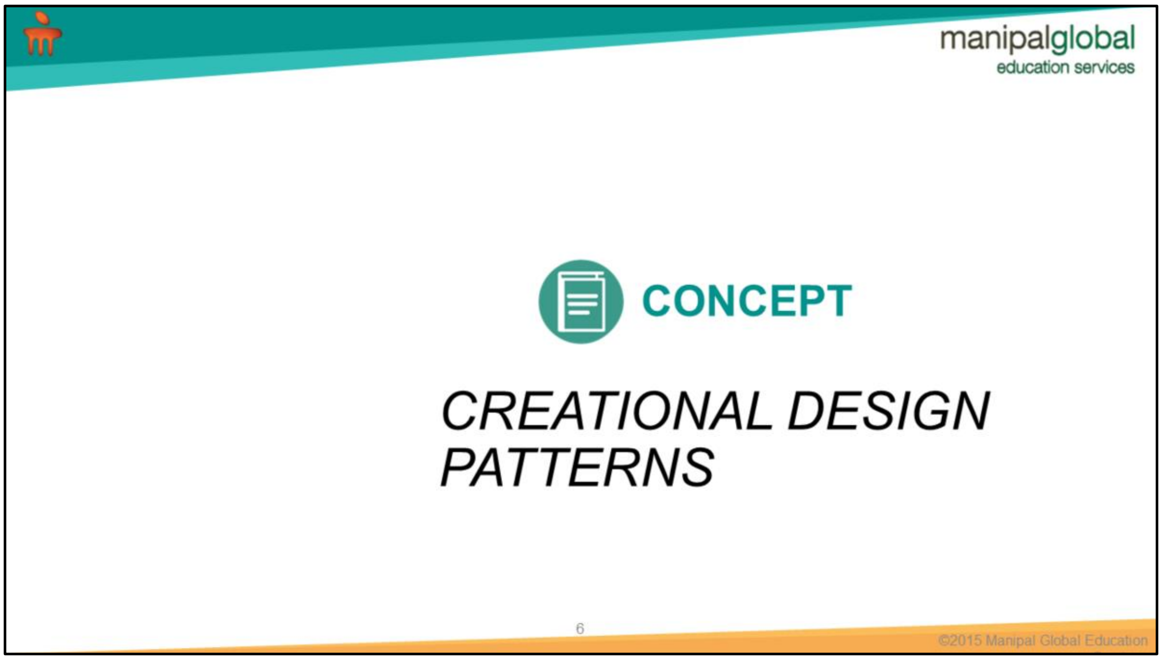
DESIGN PATTERN BASICS> WHAT IS DESIGN PATTERN

- Pattern is a effective and well proven solution to a problem occurring commonly/repeatedly
- Design patterns
  - are language-independent strategies for solving common object oriented problem.
  - Have evolved over a long period of time and they provide better solutions
- provide a standard terminology and are specific to a particular scenario
- Helps Programmers to learn software design quickly and easily
- Type te

Creational Design Patterns

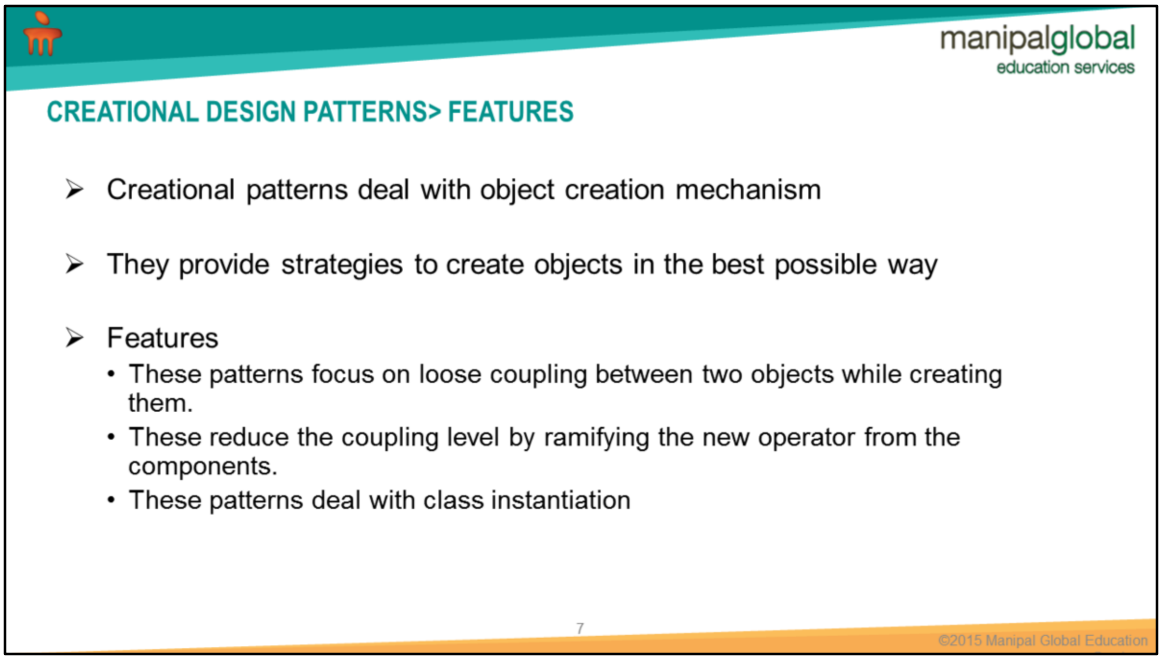
Structural Design Patterns

Behavioural Design Patterns



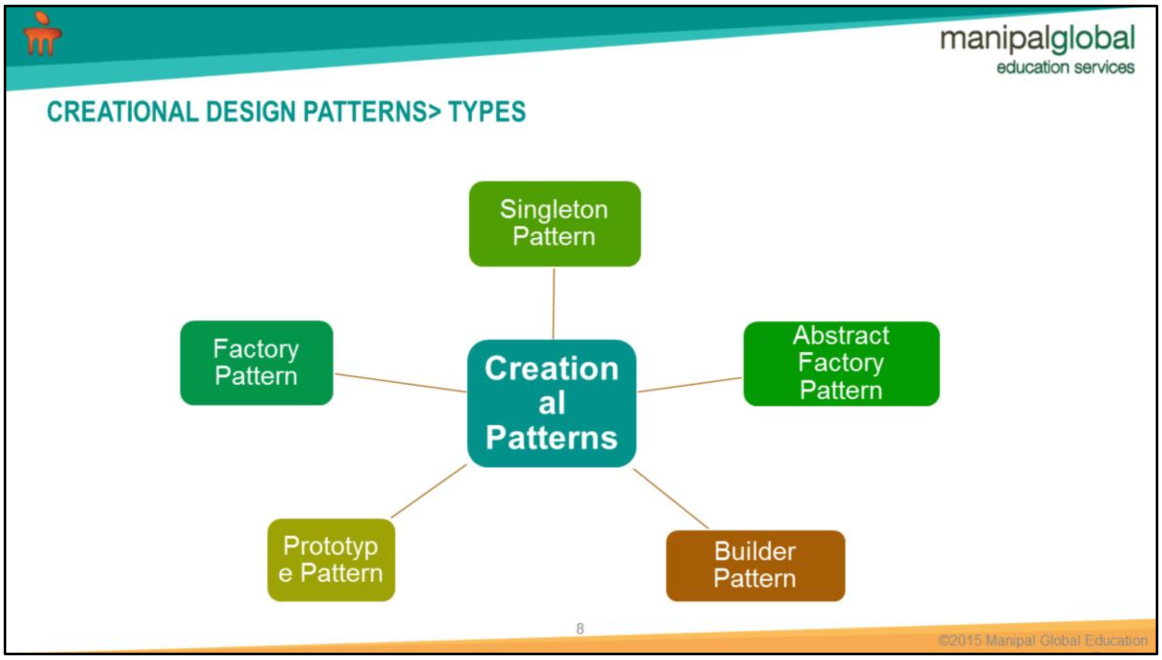
CONCEPT

CREATIONAL DESIGN PATTERNS


manipalglobal
education services

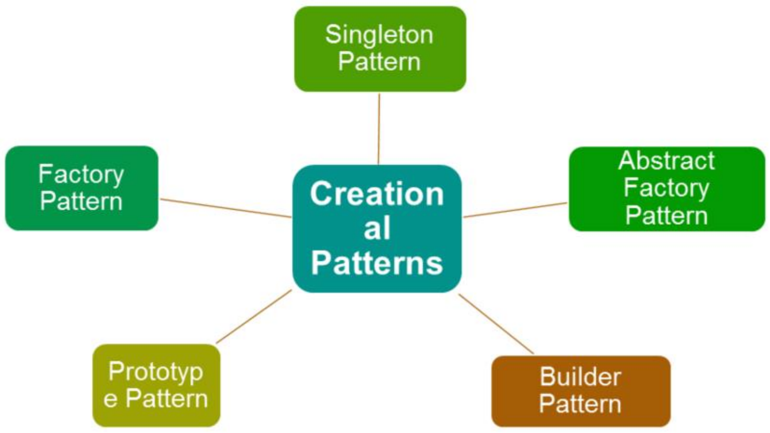
CREATIONAL DESIGN PATTERNS> FEATURES

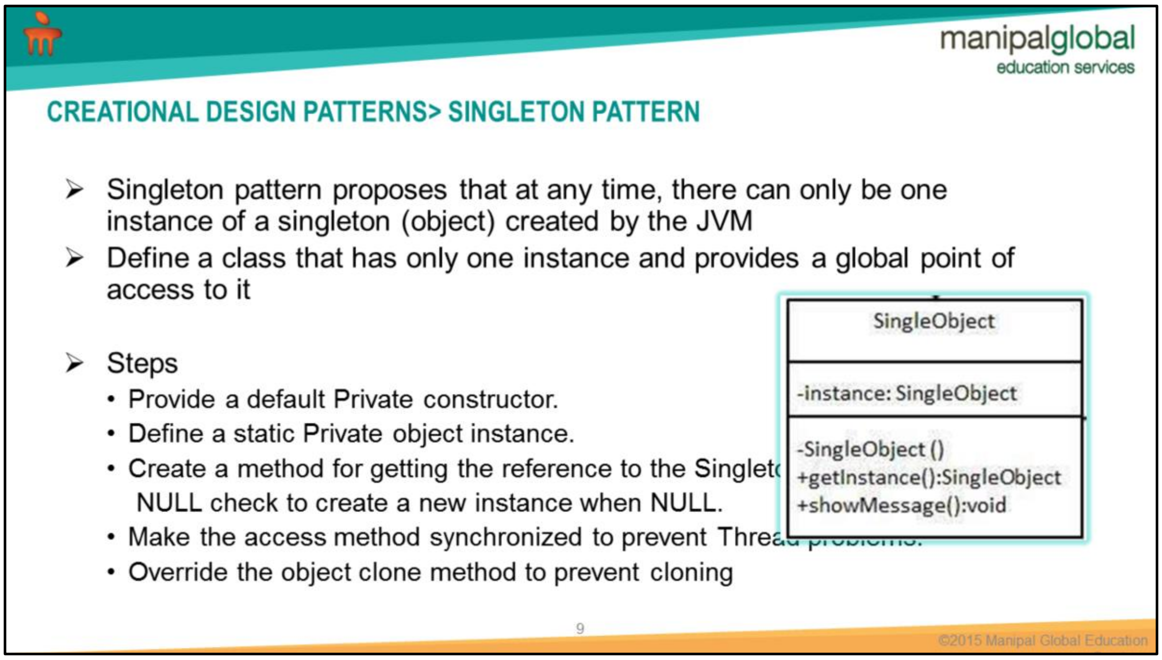
- Creational patterns deal with object creation mechanism
- They provide strategies to create objects in the best possible way
- Features
  - These patterns focus on loose coupling between two objects while creating them.
  - These reduce the coupling level by ramifying the new operator from the components.
  - These patterns deal with class instantiation



manipalglobal
education services

CREATIONAL DESIGN PATTERNS> TYPES



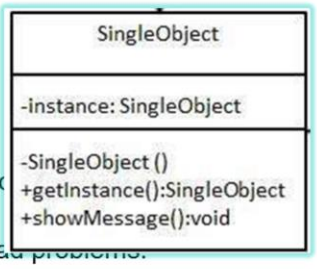


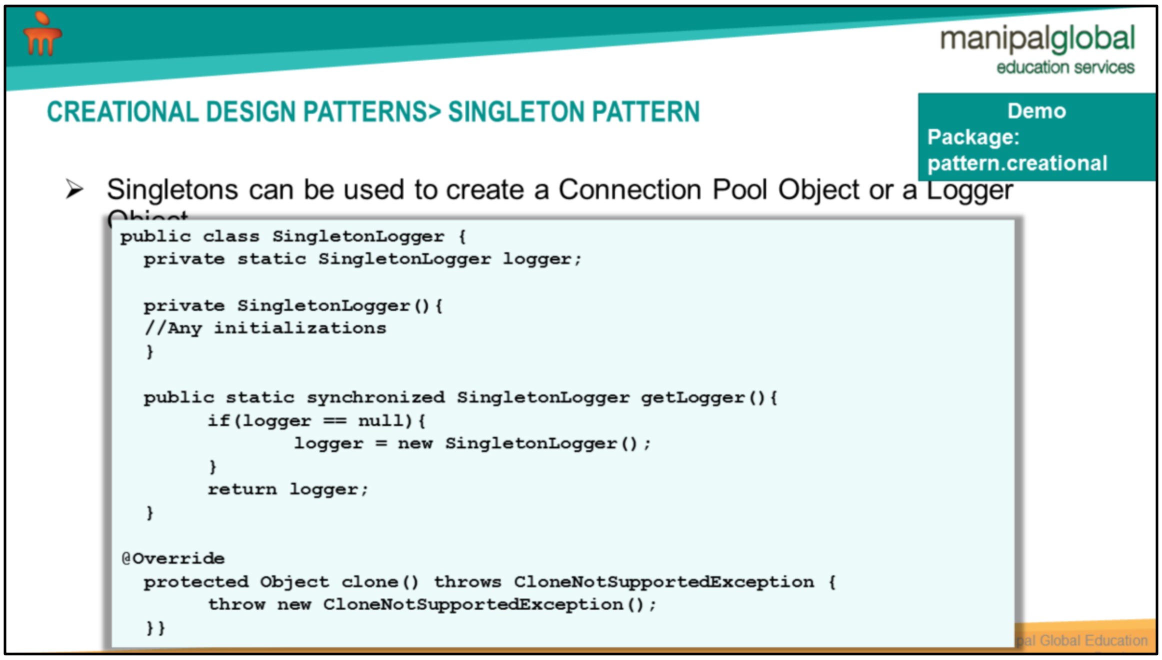
manipalglobal
education services

CREATIONAL DESIGN PATTERNS> SINGLETON PATTERN

- Singleton pattern proposes that at any time, there can only be one instance of a singleton (object) created by the JVM
- Define a class that has only one instance and provides a global point of access to it

- Steps
  - Provide a default Private constructor.
  - Define a static Private object instance.
  - Create a method for getting the reference to the Singleto NULL check to create a new instance when NULL.
  - Make the access method synchronized to prevent Thread problems.
  - Override the object clone method to prevent cloning





manipalglobal
education services

CREATIONAL DESIGN PATTERNS> SINGLETON PATTERN

Demo
Package: pattern.creational

- Singletons can be used to create a Connection Pool Object or a Logger Object

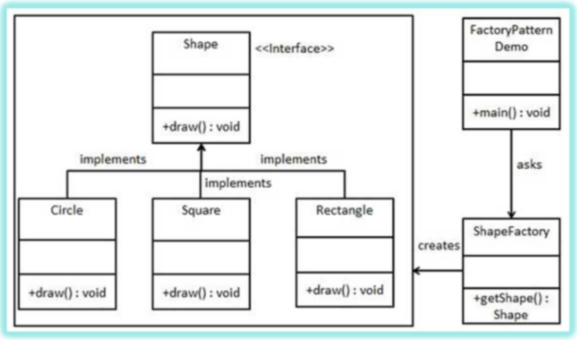
```
public class SingletonLogger {
  private static SingletonLogger logger;

  private SingletonLogger(){
  //Any initializations
  }

  public static synchronized SingletonLogger getLogger(){
        if(logger == null){
               logger = new SingletonLogger();
        }
        return logger;
  }

@Override
  protected Object clone() throws CloneNotSupportedException {
        throw new CloneNotSupportedException();
  }}
```

manipalglobal
education services

CREATIONAL DESIGN PATTERNS> FACTORY PATTERN

Demo Package: pattern.creational.Factory

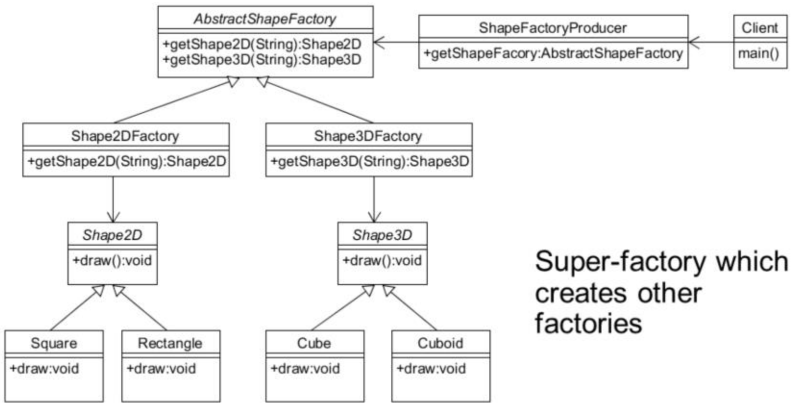
- Objects are created without exposing the creation logic to the client
- Newly created objects are referred using a common interface
- Objects of the subclasses of an inheritance hierarchy are created based on parameters provided to the Factory Class
- Increased level of encapsulation while creating objects



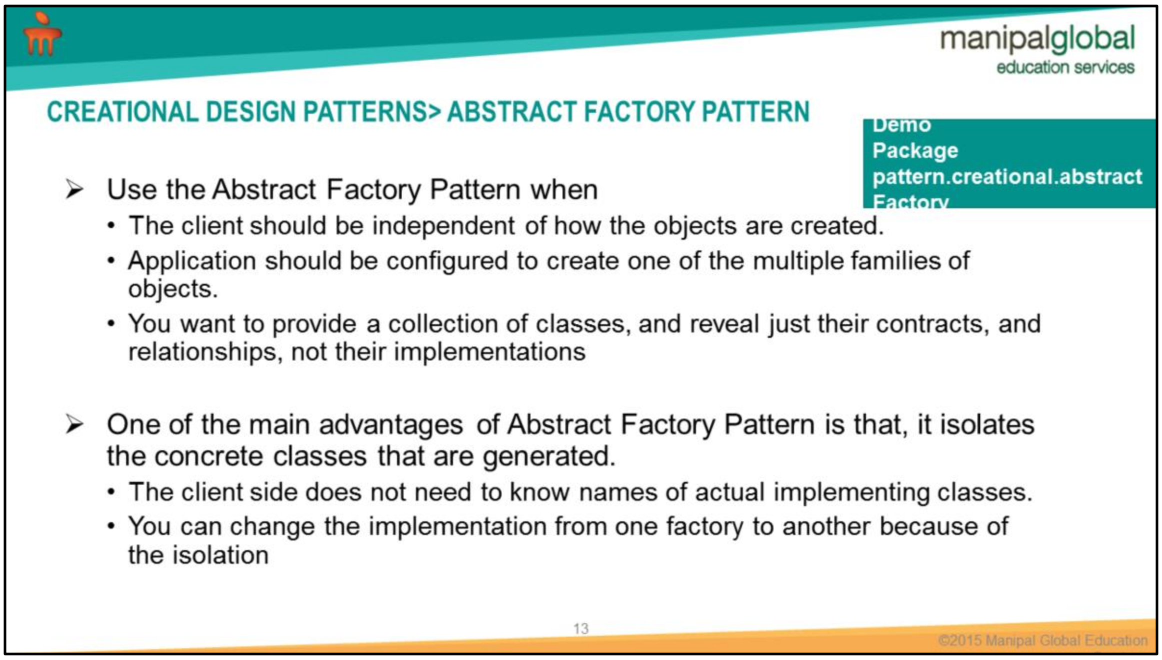
manipalglobal
education services

CREATIONAL DESIGN PATTERNS> ABSTRACT FACTORY PATTERN

- This pattern is one level of abstraction higher than the factory pattern. The abstract factory



Super-factory which creates other factories



manipalglobal
education services

CREATIONAL DESIGN PATTERNS> ABSTRACT FACTORY PATTERN

Demo
Package
pattern.creational.abstract
Factory

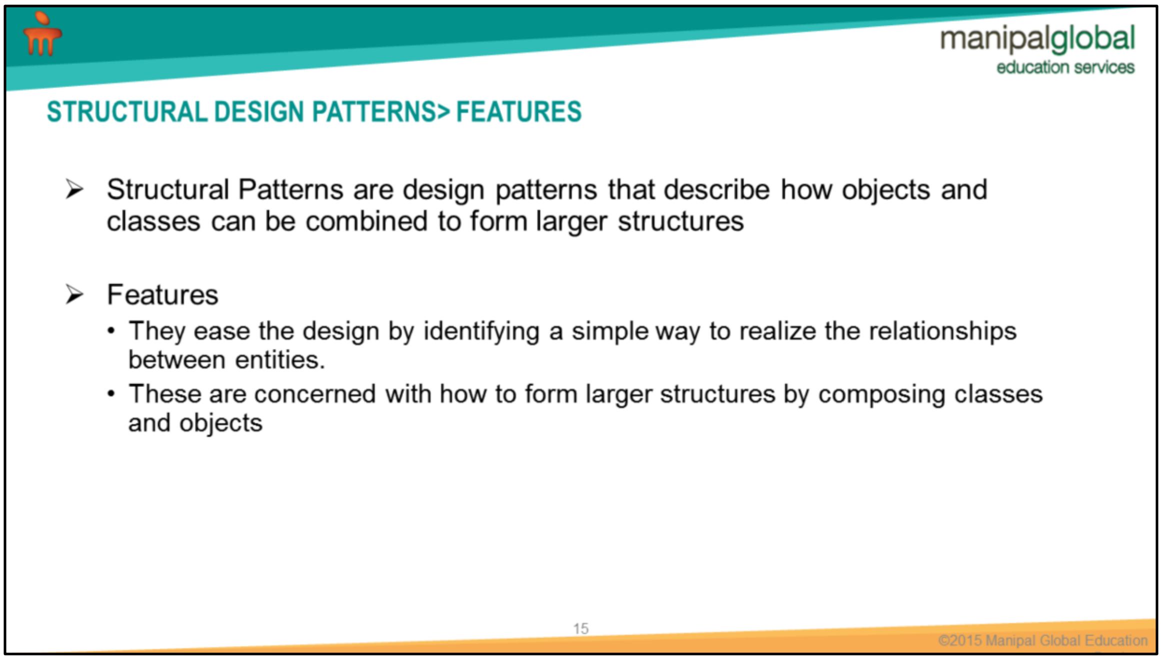
- Use the Abstract Factory Pattern when
  - The client should be independent of how the objects are created.
  - Application should be configured to create one of the multiple families of objects.
  - You want to provide a collection of classes, and reveal just their contracts, and relationships, not their implementations
- One of the main advantages of Abstract Factory Pattern is that, it isolates the concrete classes that are generated.
  - The client side does not need to know names of actual implementing classes.
  - You can change the implementation from one factory to another because of the isolation

13

manipalglobal
education services

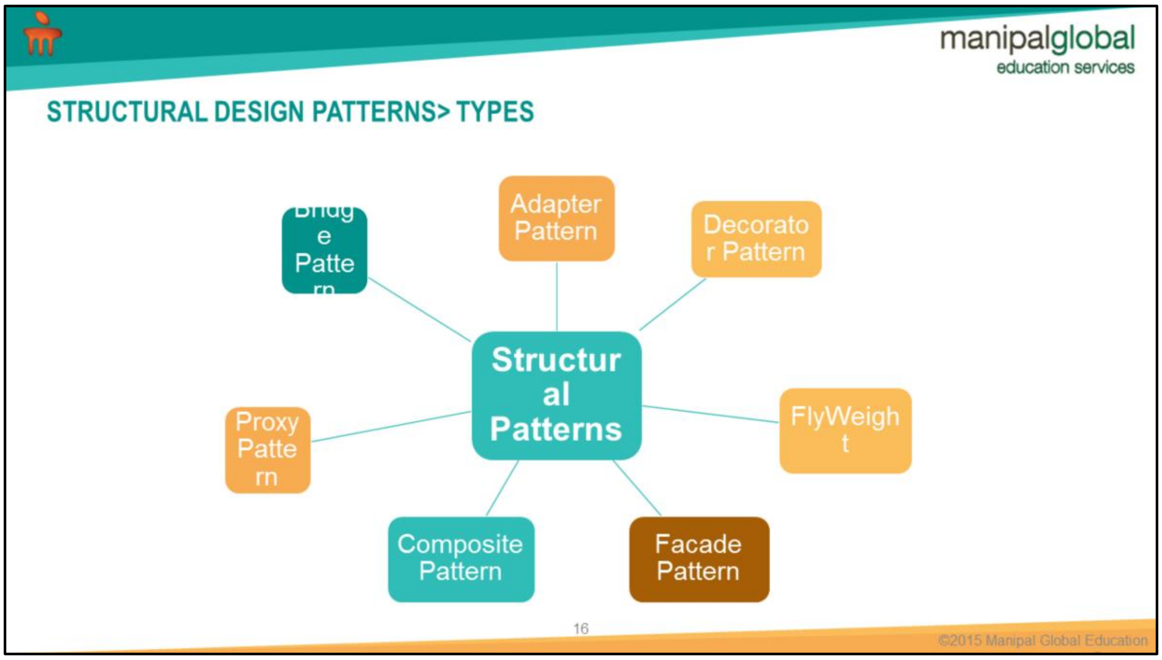
CONCEPT

STRUCTURAL DESIGN PATTERNS

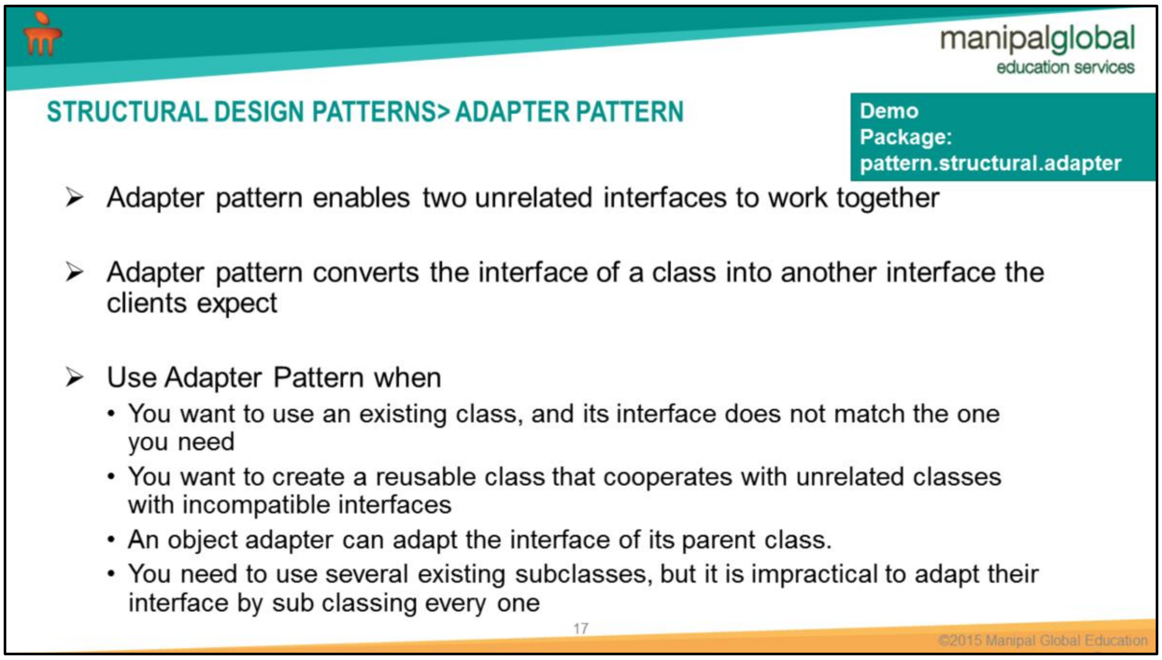

manipalglobal
education services

STRUCTURAL DESIGN PATTERNS> FEATURES

- Structural Patterns are design patterns that describe how objects and classes can be combined to form larger structures
- Features
  - They ease the design by identifying a simple way to realize the relationships between entities.
  - These are concerned with how to form larger structures by composing classes and objects



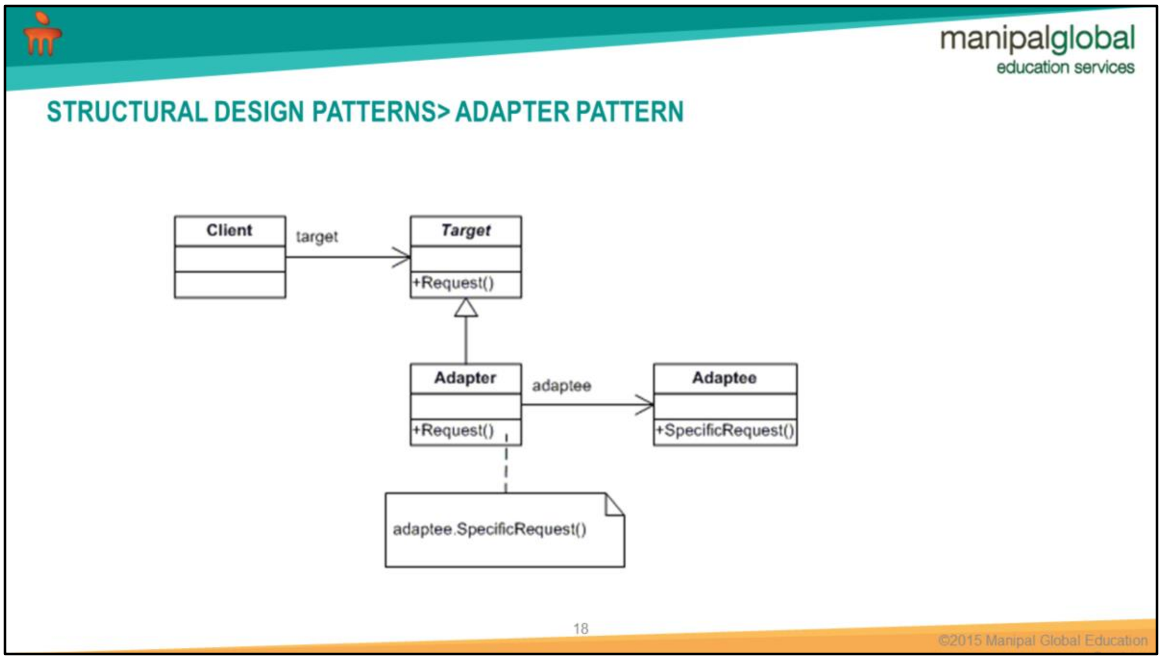
STRUCTURAL DESIGN PATTERNS> TYPES


manipalglobal
education services

STRUCTURAL DESIGN PATTERNS> ADAPTER PATTERN

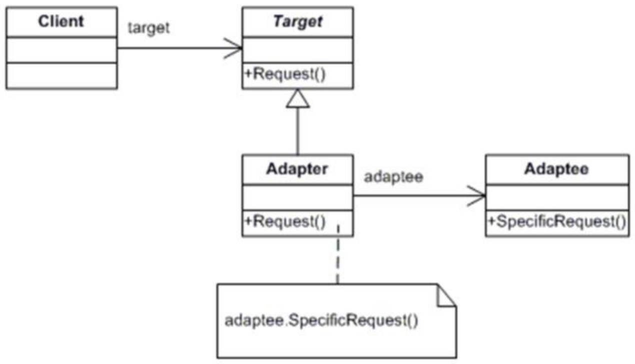
Demo Package: pattern.structural.adapter

- Adapter pattern enables two unrelated interfaces to work together
- Adapter pattern converts the interface of a class into another interface the clients expect
- Use Adapter Pattern when
  - You want to use an existing class, and its interface does not match the one you need
  - You want to create a reusable class that cooperates with unrelated classes with incompatible interfaces
  - An object adapter can adapt the interface of its parent class.
  - You need to use several existing subclasses, but it is impractical to adapt their interface by sub classing every one



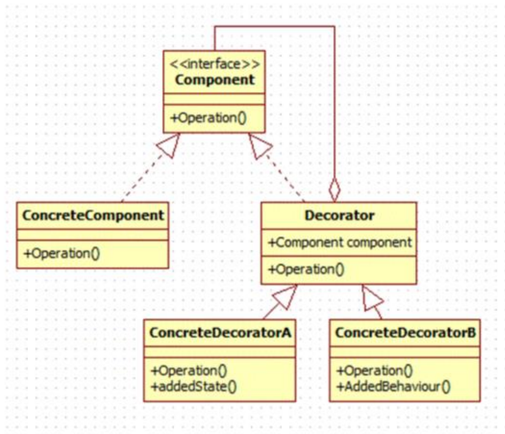
manipalglobal
education services

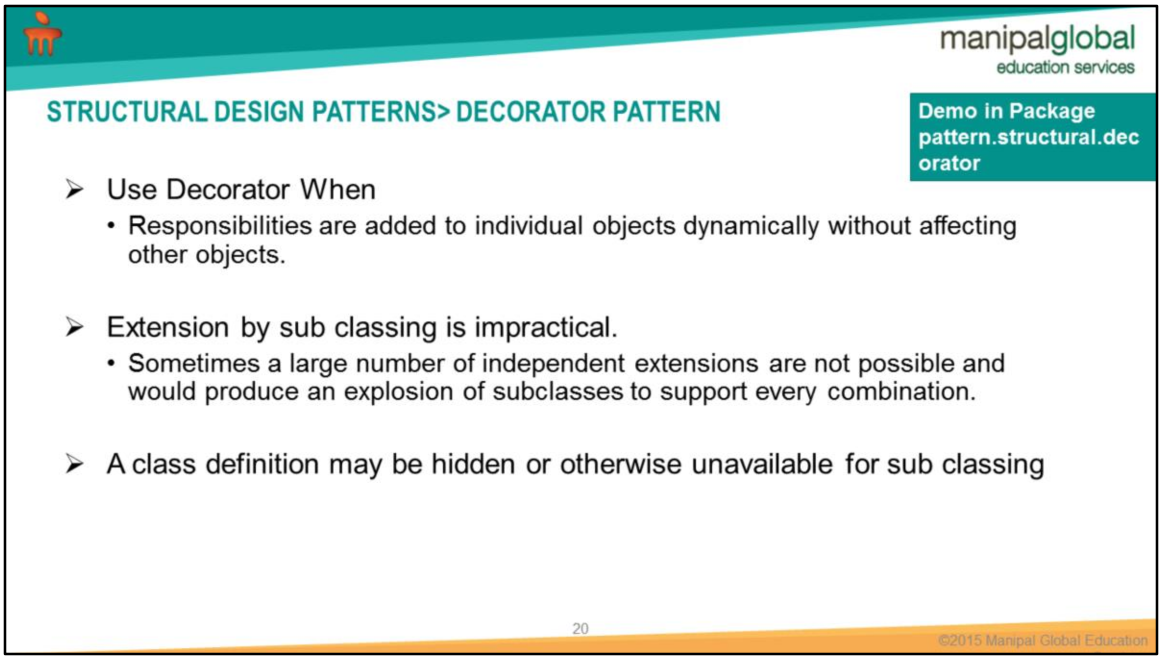
STRUCTURAL DESIGN PATTERNS> ADAPTER PATTERN



STRUCTURAL DESIGN PATTERNS> DECORATOR PATTERN

- The decorator pattern helps add behaviour or responsibilities to an object without altering its structure
- Enhances capability of Object dynamically
- This is also called "Wrapper."

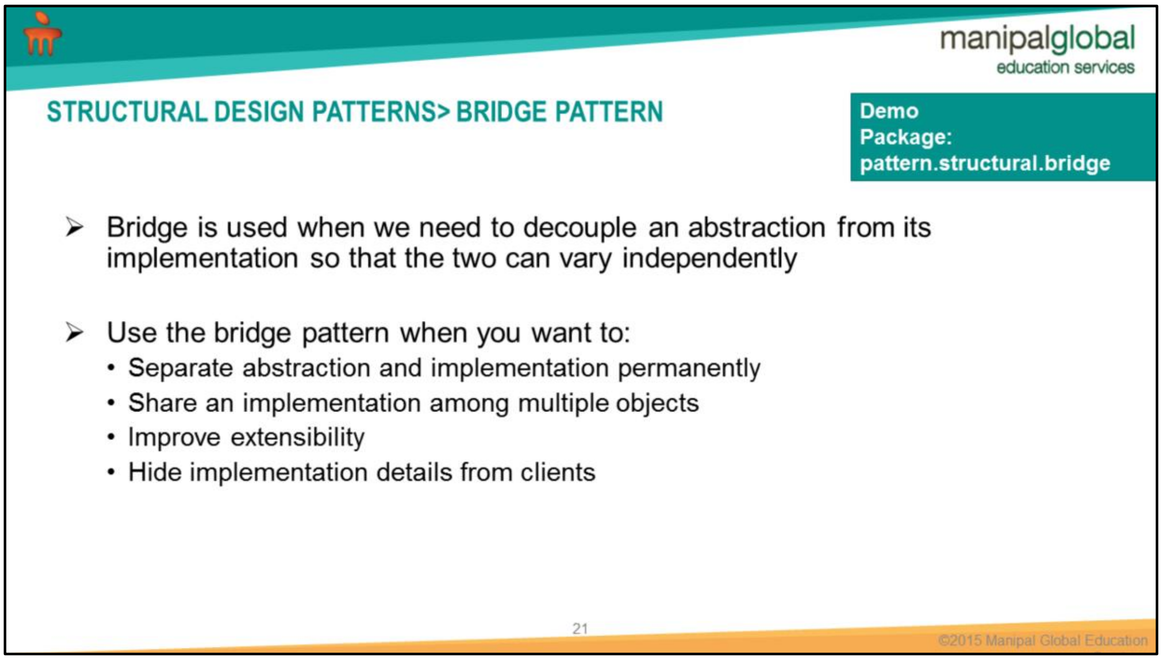




STRUCTURAL DESIGN PATTERNS> DECORATOR PATTERN

Demo in Package pattern.structural.decorator

- Use Decorator When
  - Responsibilities are added to individual objects dynamically without affecting other objects.
- Extension by sub classing is impractical.
  - Sometimes a large number of independent extensions are not possible and would produce an explosion of subclasses to support every combination.
- A class definition may be hidden or otherwise unavailable for sub classing



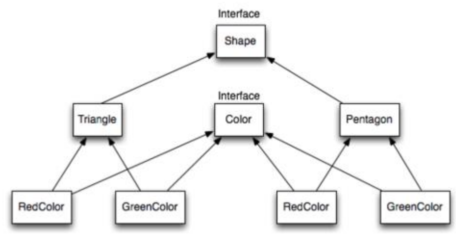
manipalglobal
education services

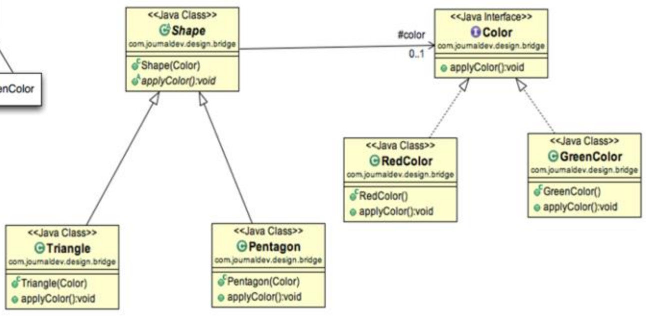
STRUCTURAL DESIGN PATTERNS> BRIDGE PATTERN

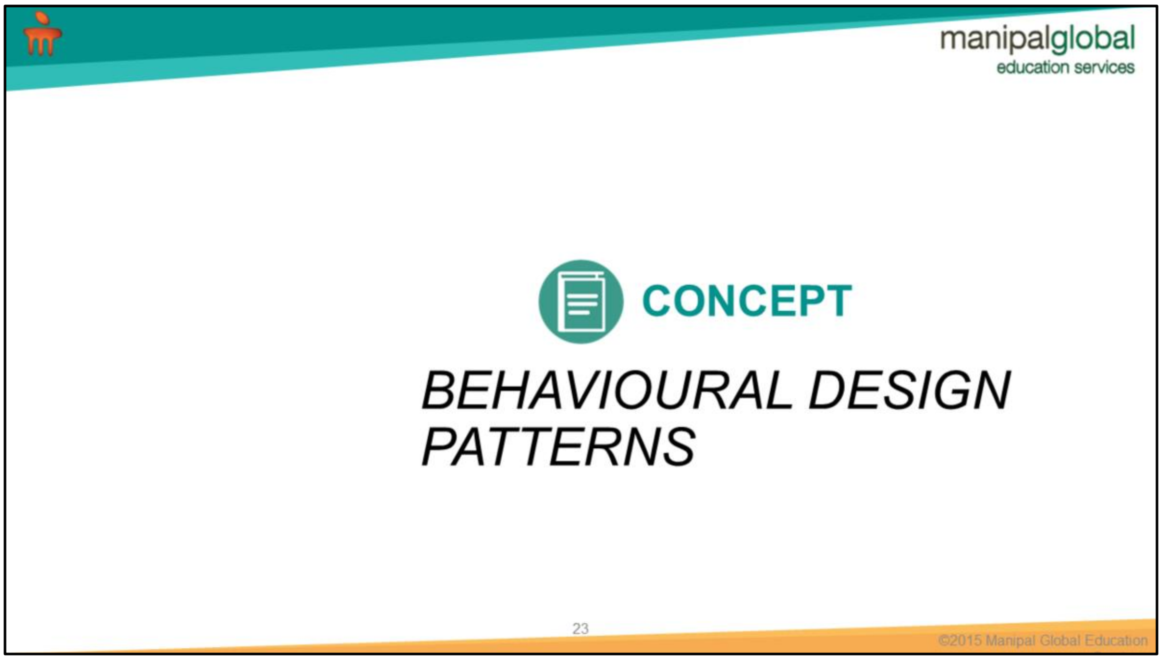
Demo Package: pattern.structural.bridge

- Bridge is used when we need to decouple an abstraction from its implementation so that the two can vary independently
- Use the bridge pattern when you want to:
  - Separate abstraction and implementation permanently
  - Share an implementation among multiple objects
  - Improve extensibility
  - Hide implementation details from clients

manipalglobal
education services

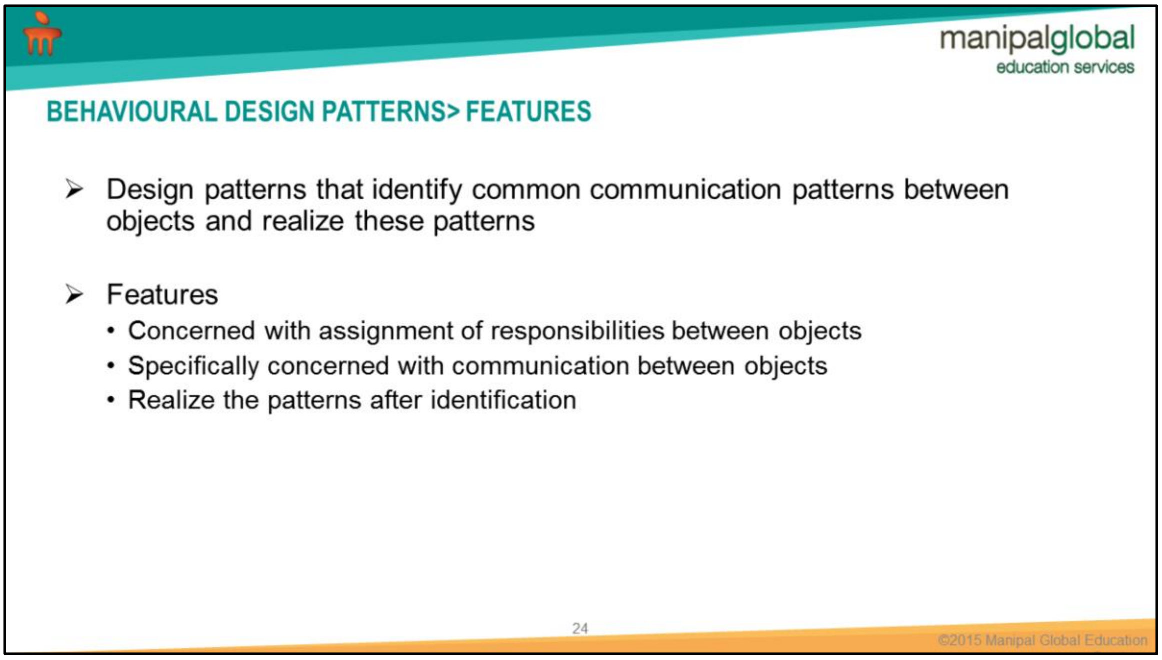






CONCEPT

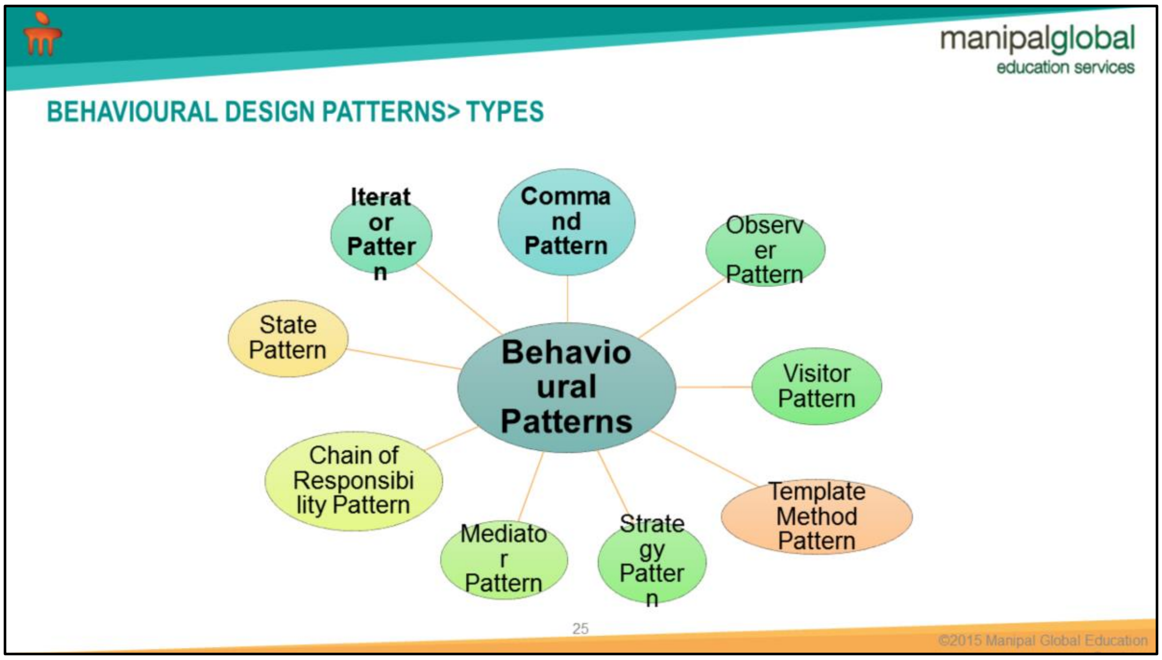
BEHAVIOURAL DESIGN PATTERNS


manipalglobal
education services

BEHAVIOURAL DESIGN PATTERNS> FEATURES

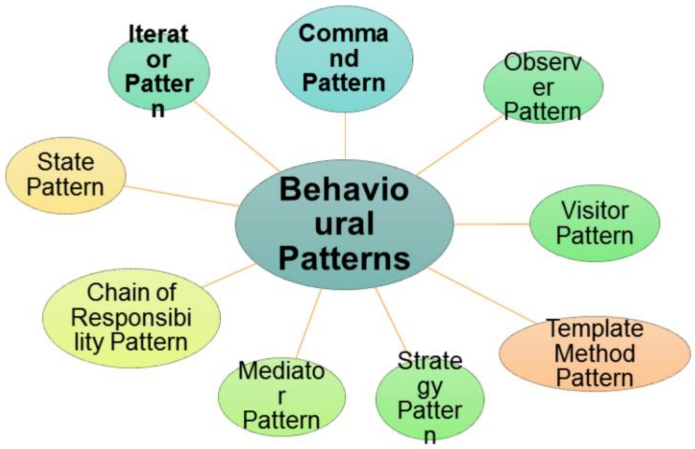
- Design patterns that identify common communication patterns between objects and realize these patterns
- Features
  - Concerned with assignment of responsibilities between objects
  - Specifically concerned with communication between objects
  - Realize the patterns after identification

24



manipalglobal
education services

BEHAVIOURAL DESIGN PATTERNS> TYPES

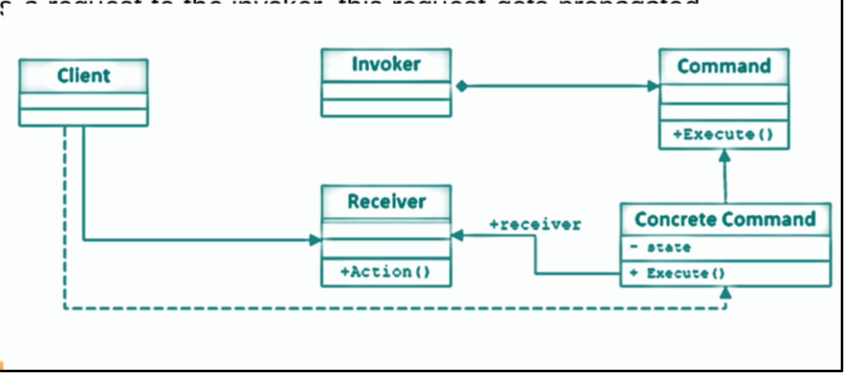


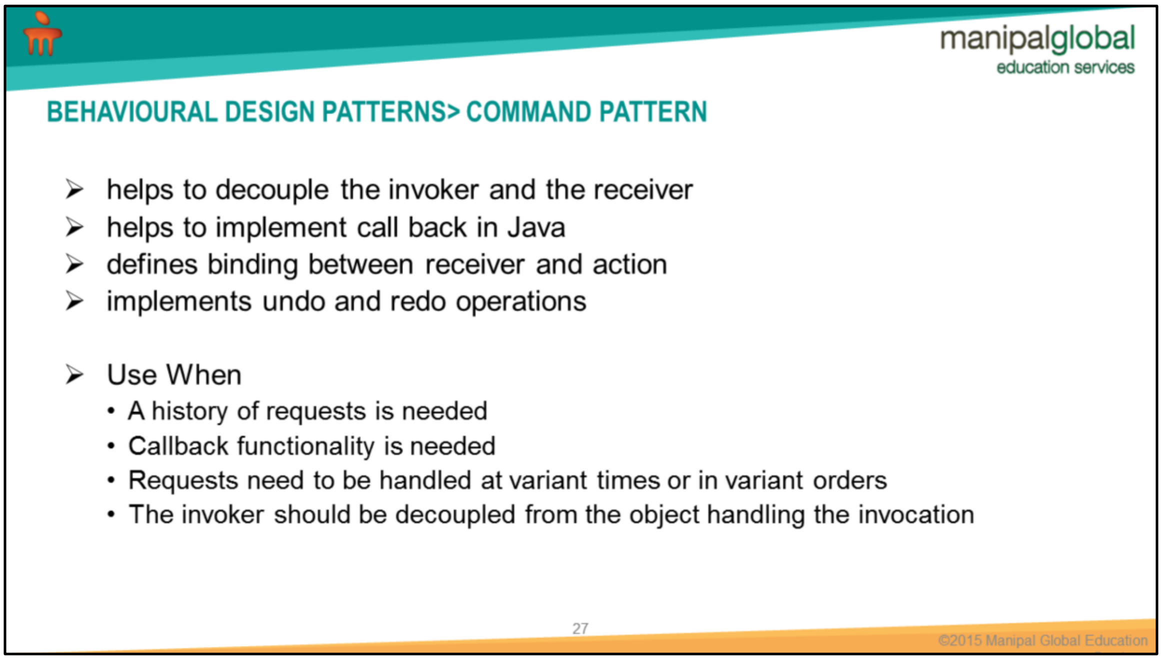
manipalglobal
education services

BEHAVIOURAL DESIGN PATTERNS> COMMAND PATTERN

Demo
Package:
pattern.behavioural.command

- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations
- The client passes a request to the invoker, this request gets propagated as a command





manipalglobal
education services

BEHAVIOURAL DESIGN PATTERNS> COMMAND PATTERN

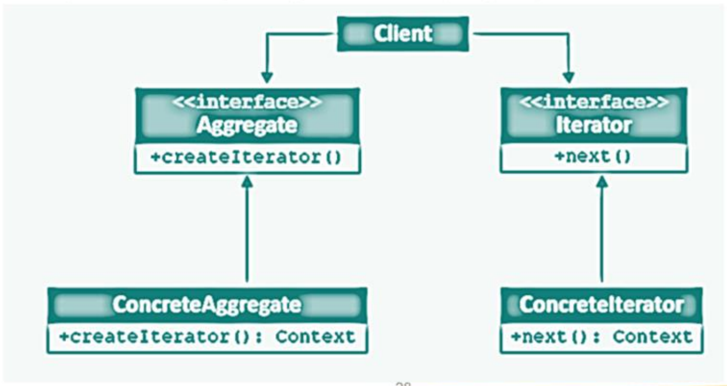
- helps to decouple the invoker and the receiver
- helps to implement call back in Java
- defines binding between receiver and action
- implements undo and redo operations

- Use When
  - A history of requests is needed
  - Callback functionality is needed
  - Requests need to be handled at variant times or in variant orders
  - The invoker should be decoupled from the object handling the invocation

manipalglobal
education services

BEHAVIOURAL DESIGN PATTERNS> ITERATOR PATTERN

- It is to provide a way to access the elements of an aggregate object sequentially, without exposing the underlying representation

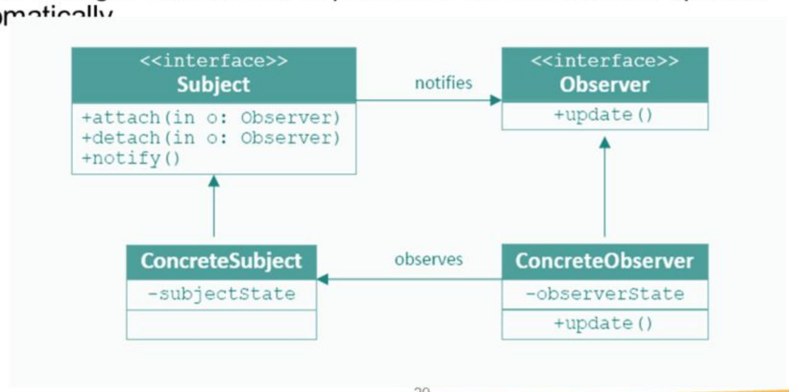


manipalglobal
education services

BEHAVIOURAL DESIGN PATTERNS> OBSERVER PATTERN

Demo Package: pattern.behavioural.observer

- Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated autom


<<interface>>
Subject
+attach(in o: Observer)
+detach(in o: Observer)
+notify()
notifies
<<interface>>
Observer
+update()
ConcreteSubject
-subjectState
observes
ConcreteObserver
-observerState
+update()