Software architecture and design

Nguyen Manh Hung
The posts and telecommunications Institute of technology (PTIT)

SOFTWARE DESIGN PATTERNS

References

- This slide uses images and definitions from:
  - javapoint.com
  - tutorialspoint.com

Outline

- Creational patterns
  - Factory
  - Abstract factory
  - Singleton
  - Prototype
  - Builder
  - Object pool

Outline(contd)

- Structural patterns
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy

Outline(contd)

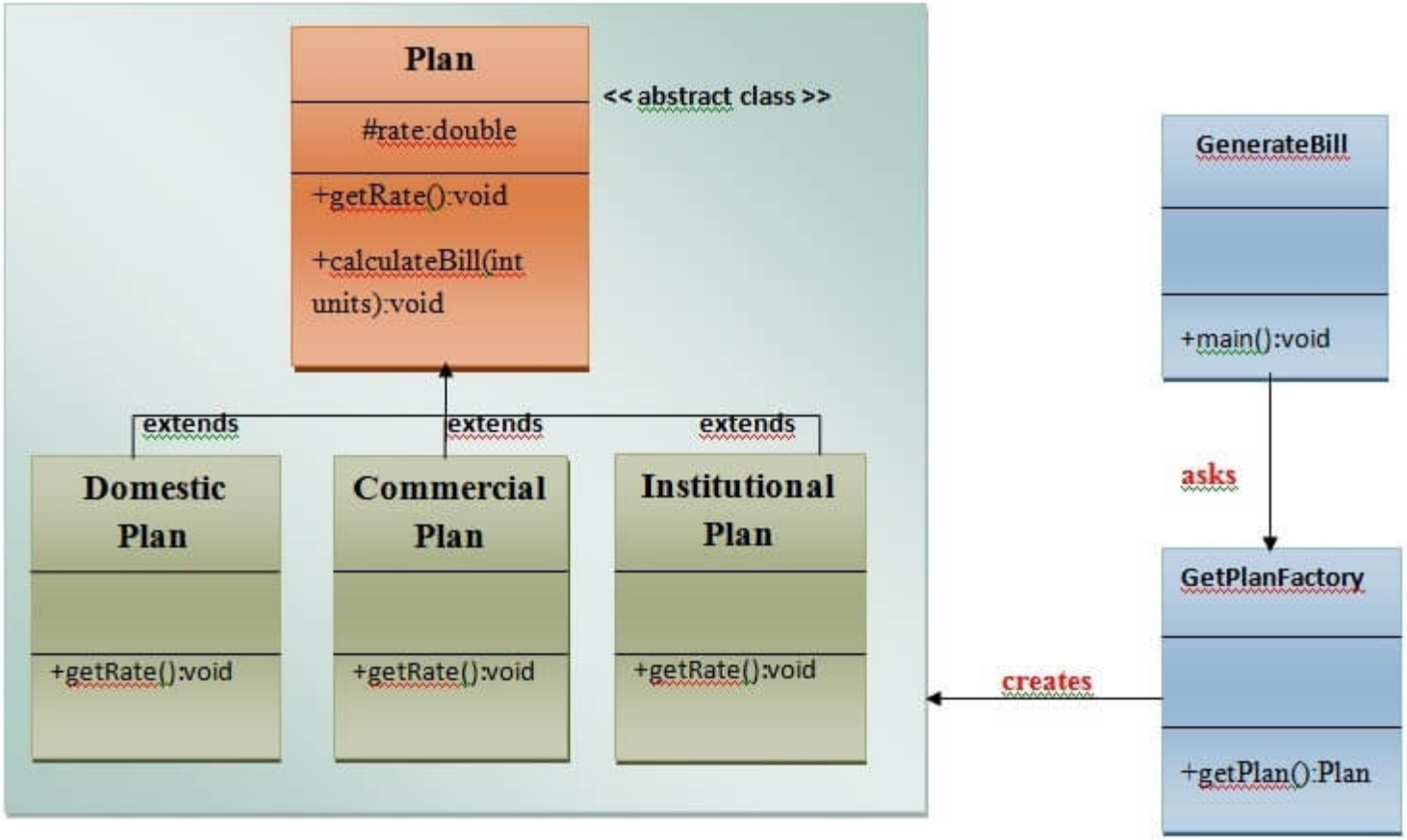
- Behavioral patterns
  - Chain of responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template

Factory

- Introduction
  - define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.
  - subclasses are responsible to create the instance of the class.
- Advantage
  - allows the sub-classes to choose the type of objects to create
  - promotes the loose-coupling by eliminating the need to bind application-specific classes into the code.
- Usage when
  - a class doesn't know what sub-classes will be required to create
  - a class wants that its sub-classes specify the objects to be created
  - the parent classes choose the creation of objects to its sub-classes

Factory(contd)

- Example:

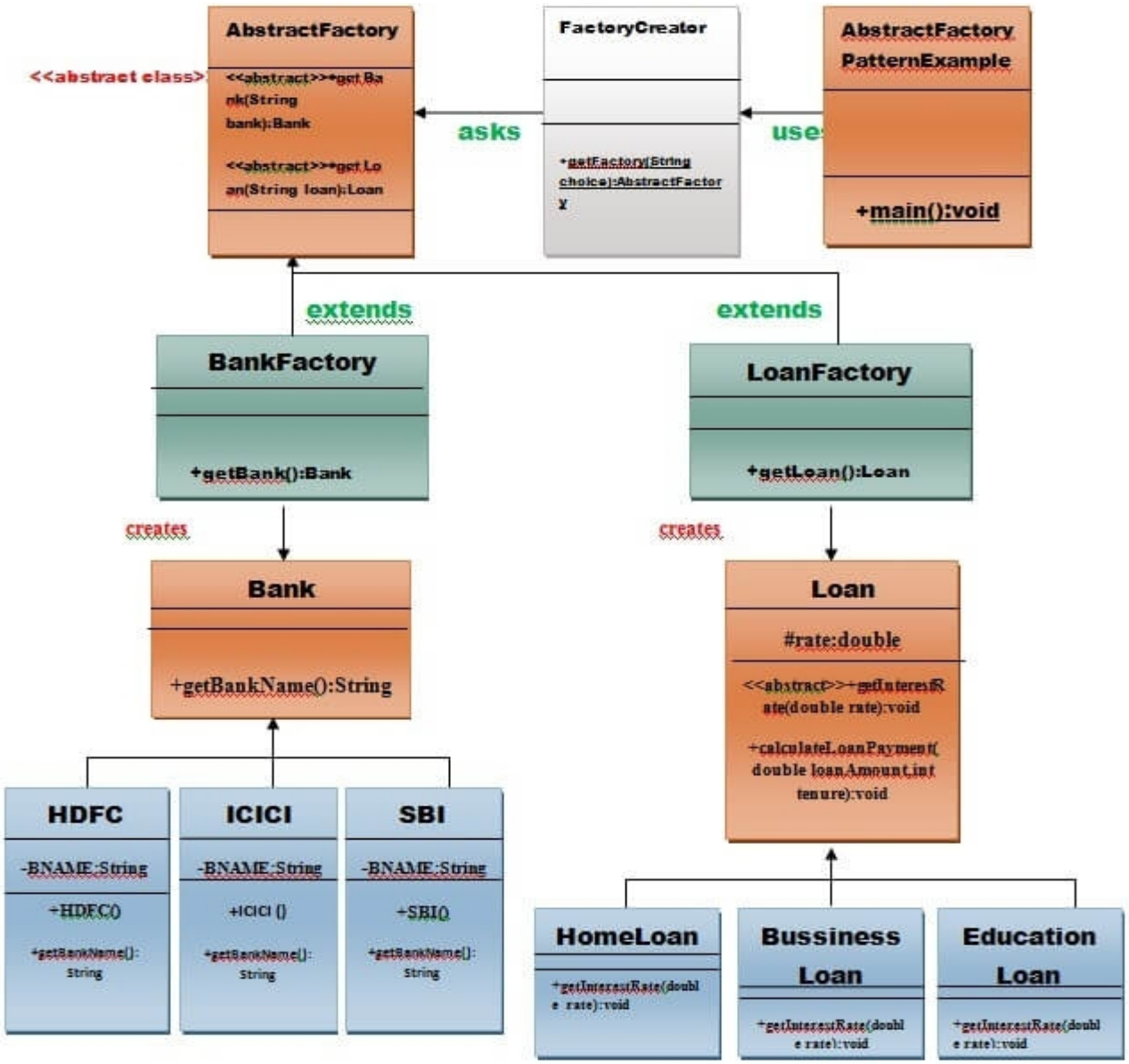


Abstract Factory

- Introduction
  - define an interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes
- Advantage
  - isolates the client code from concrete (implementation) classes
  - eases the exchanging of object families
  - promotes consistency among objects
- Usage when
  - the system needs to be independent of how its object are created, composed, and represented.
  - the family of related objects has to be used together
  - want to provide a library of objects that does not show implementations and only reveals interfaces
  - the system needs to be configured with one of a multiple family of objects.

Abstract Factory(contd)

- Example:

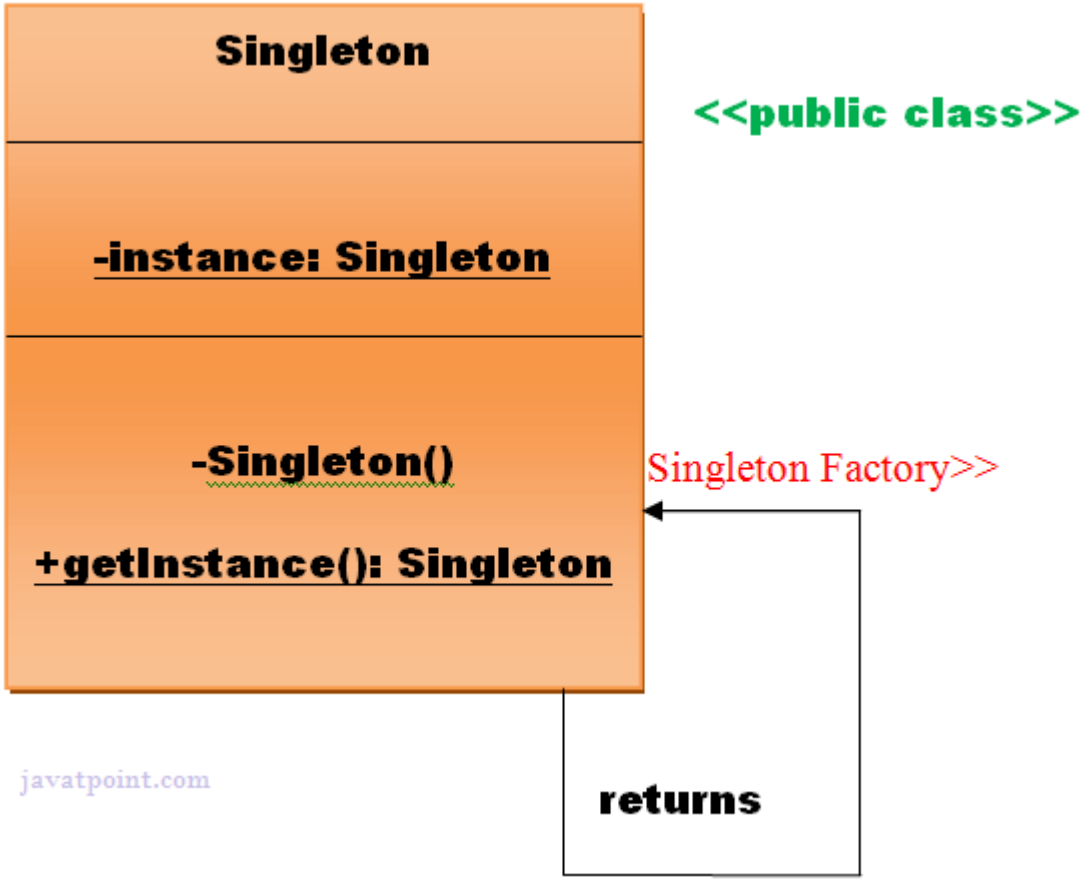


Singleton

- Introduction
  - define a class that has only one instance and provides a global point of access to it
- Advantage
  - Saves memory because object is not created at each request. Only single instance is reused again and again
- Usage when
  - mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc.

Singleton(contd)

- Example:

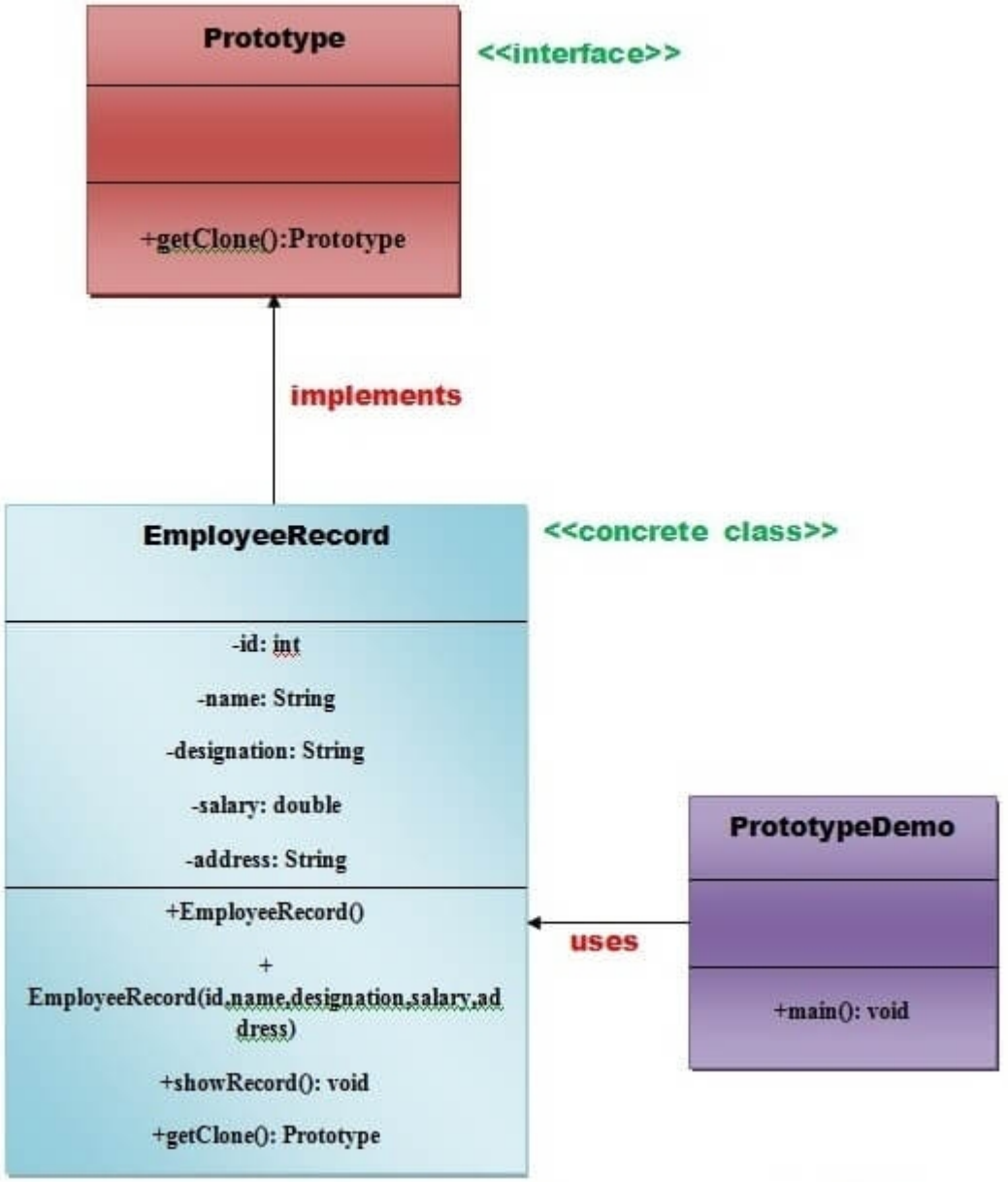


Prototype

- Introduction
  - cloning of an existing object instead of creating new one and can also be customized as per the requirement
- Advantage
  - reduces the need of sub-classing.
  - hides complexities of creating objects.
  - The clients can get new objects without knowing which type of object it will be.
  - lets you add or remove objects at runtime.
- Usage when
  - the classes are instantiated at runtime.
  - the cost of creating an object is expensive or complicated.
  - want to keep the number of classes in an application minimum.
  - the client application needs to be unaware of object creation and representation.

Prototype(contd)

- Example:

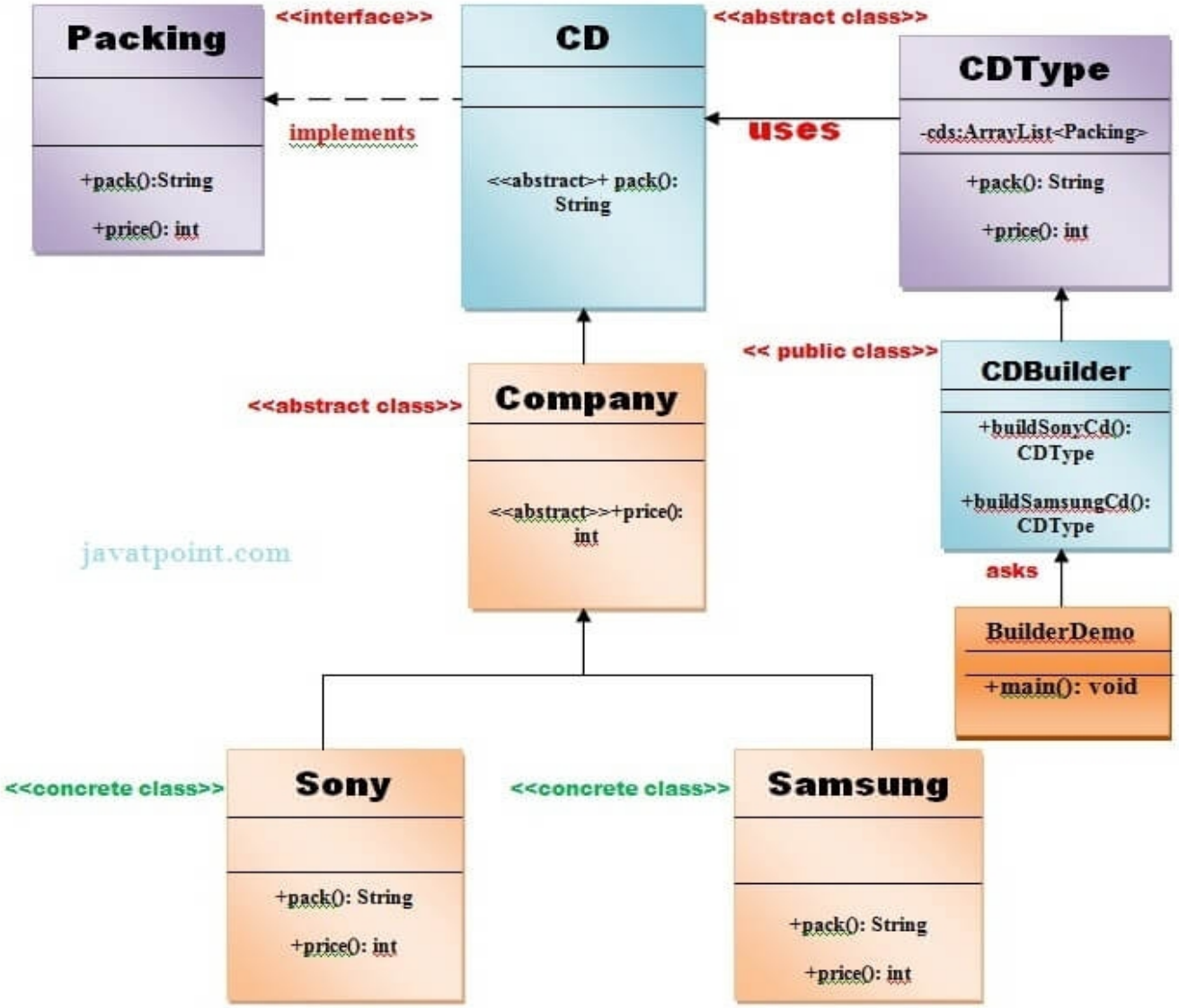


Builder

- Introduction
  - construct a complex object from simple objects using step-by-step approach
- Advantage
  - provides clear separation between the construction and representation of an object.
  - provides better control over construction process.
  - supports to change the internal representation of objects.
- Usage when
  - object can't be created in single step like in the de-serialization of a complex object

Builder(contd)

- Example:

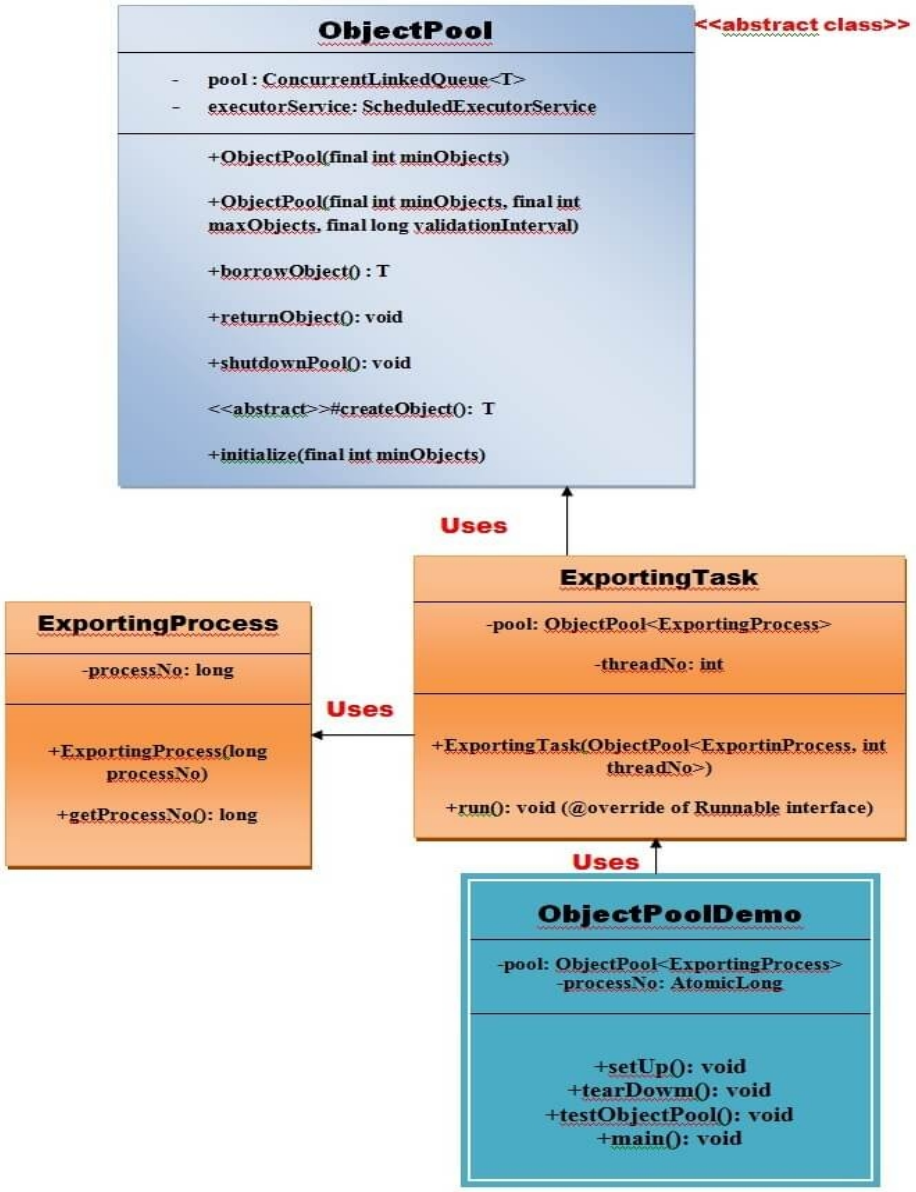


Object pool

- Introduction
  - to reuse the object that are expensive to create
  - Objects in the pool have a lifecycle: creation, validation and destroy
- Advantage
  - boosts the performance of the application significantly.
  - most effective in a situation where the rate of initializing a class instance is high.
  - manages the connections and provides a way to reuse and share them.
  - can also provide the limit for the maximum number of objects that can be created.
- Usage when
  - an application requires objects which are expensive to create.
  - there are several clients who need the same resource at different times.

Object pool(contd)

- Example:

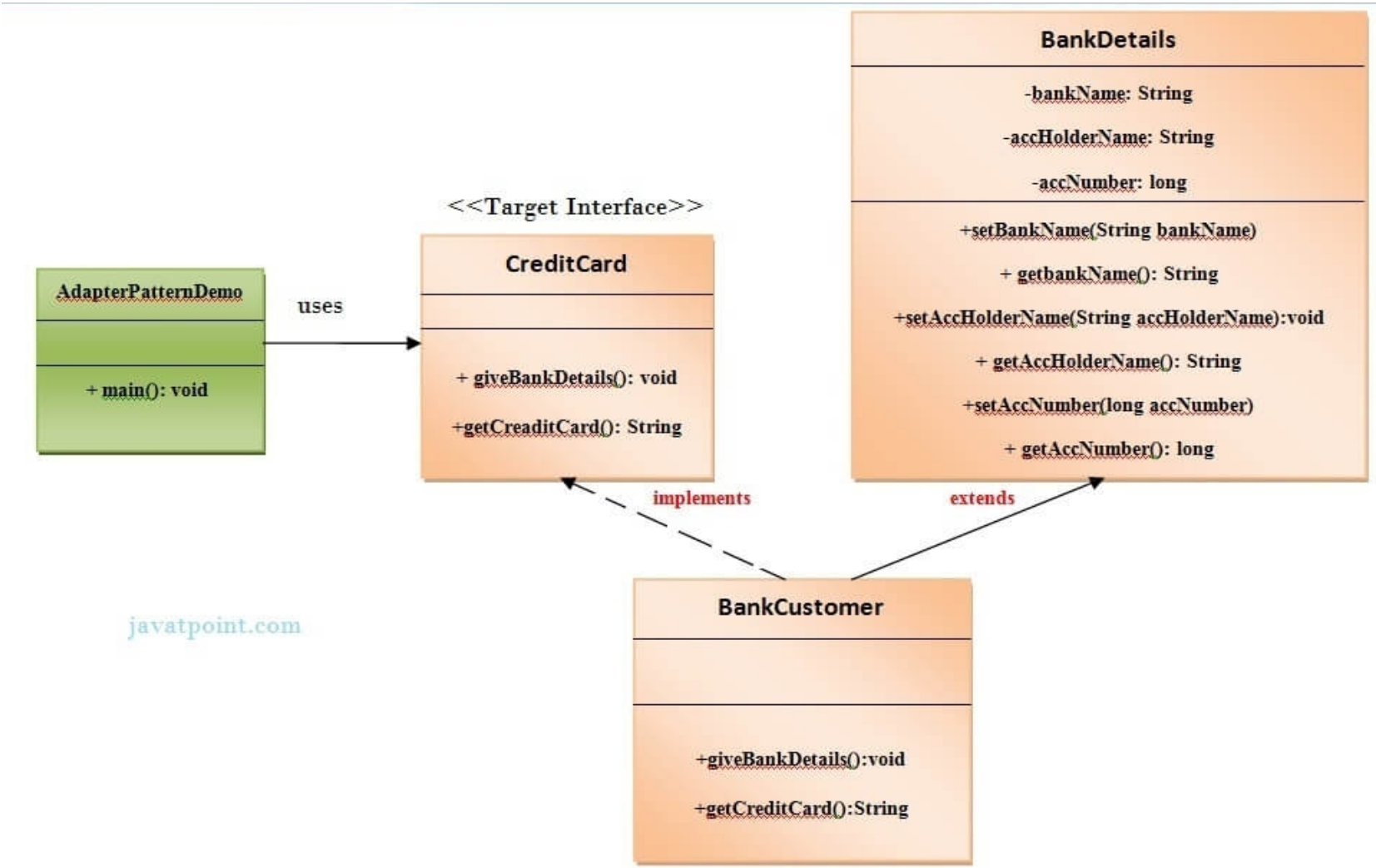


Adapter (Wrapper)

- Introduction
  - converts the interface of a class into another interface that a client wants
- Advantage
  - allows two or more previously incompatible objects to interact.
  - allows reusability of existing functionality.
- Usage when
  - an object needs to utilize an existing class with an incompatible interface.
  - want to create a reusable class that cooperates with classes which don't have compatible interfaces.
  - want to create a reusable class that cooperates with classes which don't have compatible interfaces.

Adapter (contd)

- Example:

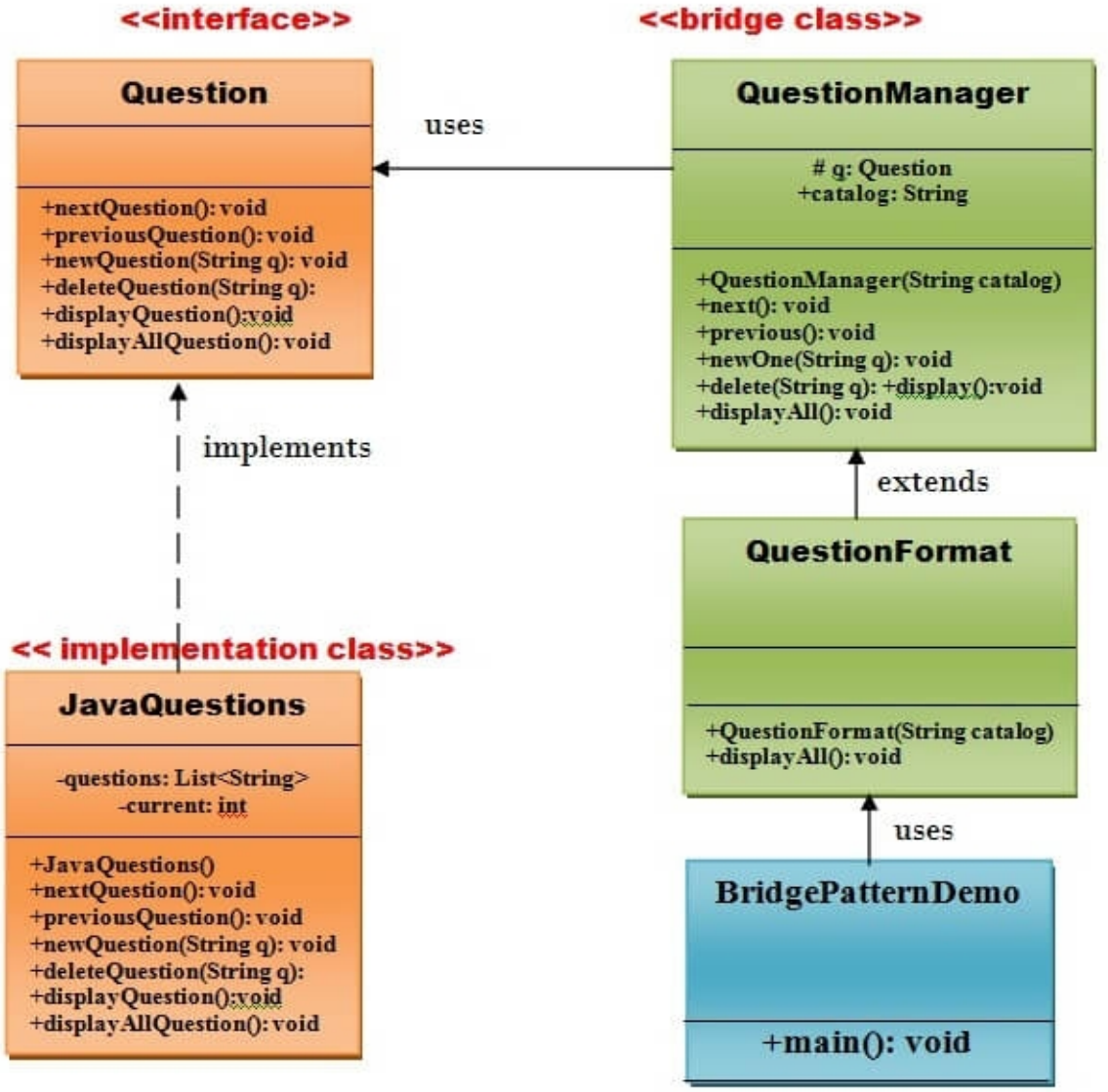


Bridge

- Introduction
  - decouple the functional abstraction from the implementation so that the two can vary independently
- Advantage
  - enables the separation of implementation from the interface.
  - improves the extensibility.
  - allows the hiding of implementation details from the client.
- Usage when
  - don't want a permanent binding between the functional abstraction and its implementation.
  - both the functional abstraction and its implementation need to extended using sub-classes.
  - mostly used in those places where changes are made in the implementation does not affect the clients.

Bridge(contd)

- Example:

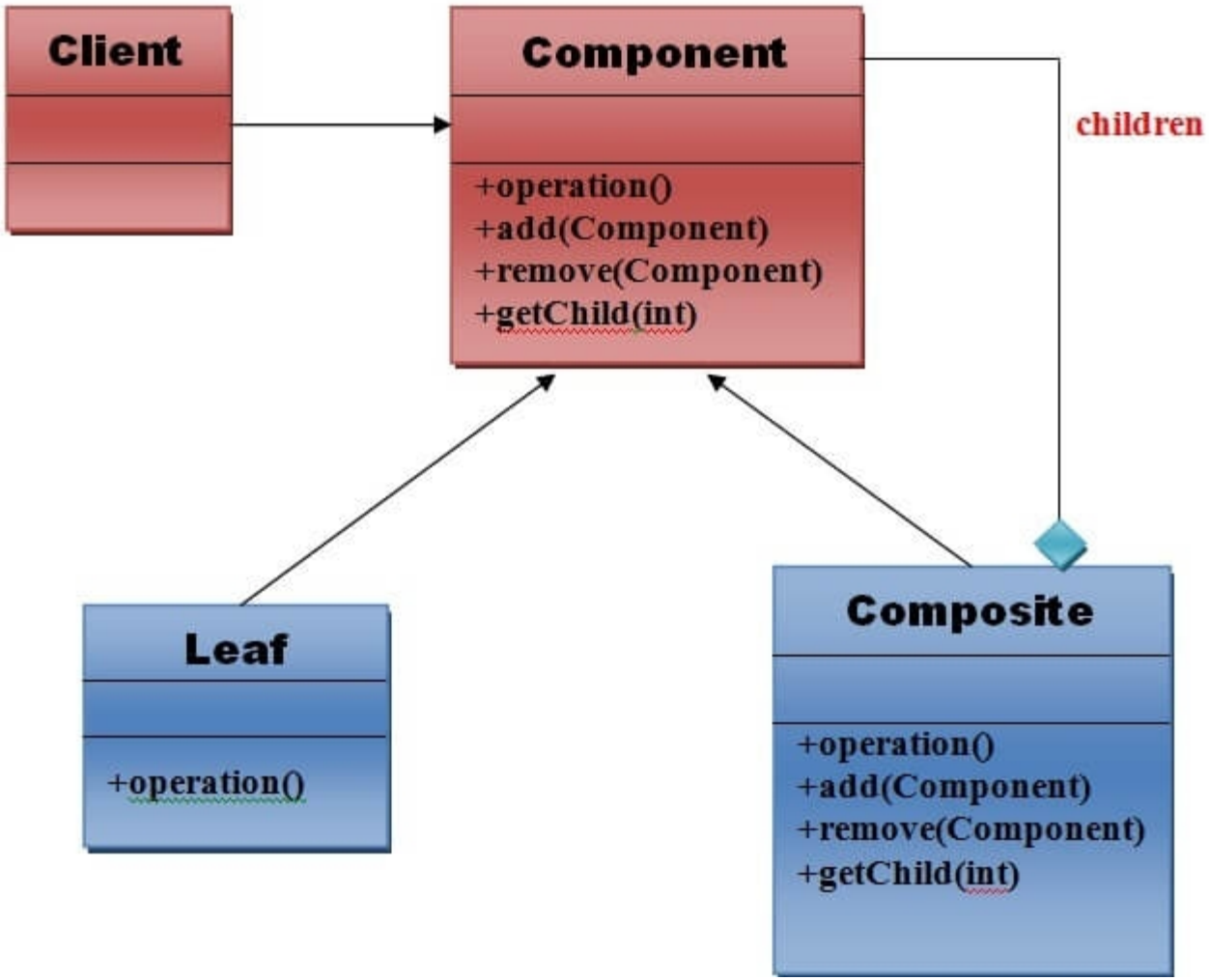


Composite

- Introduction
  - allow clients to operate in generic manner on objects that may or may not represent a hierarchy of objects
- Advantage
  - defines class hierarchies that contain primitive and complex objects.
  - makes easier to you to add new kinds of components.
  - provides flexibility of structure with manageable class or interface.
- Usage when
  - want to represent a full or partial hierarchy of objects.
  - the responsibilities are needed to be added dynamically to the individual objects without affecting other objects.

Composite(contd)

- Example:

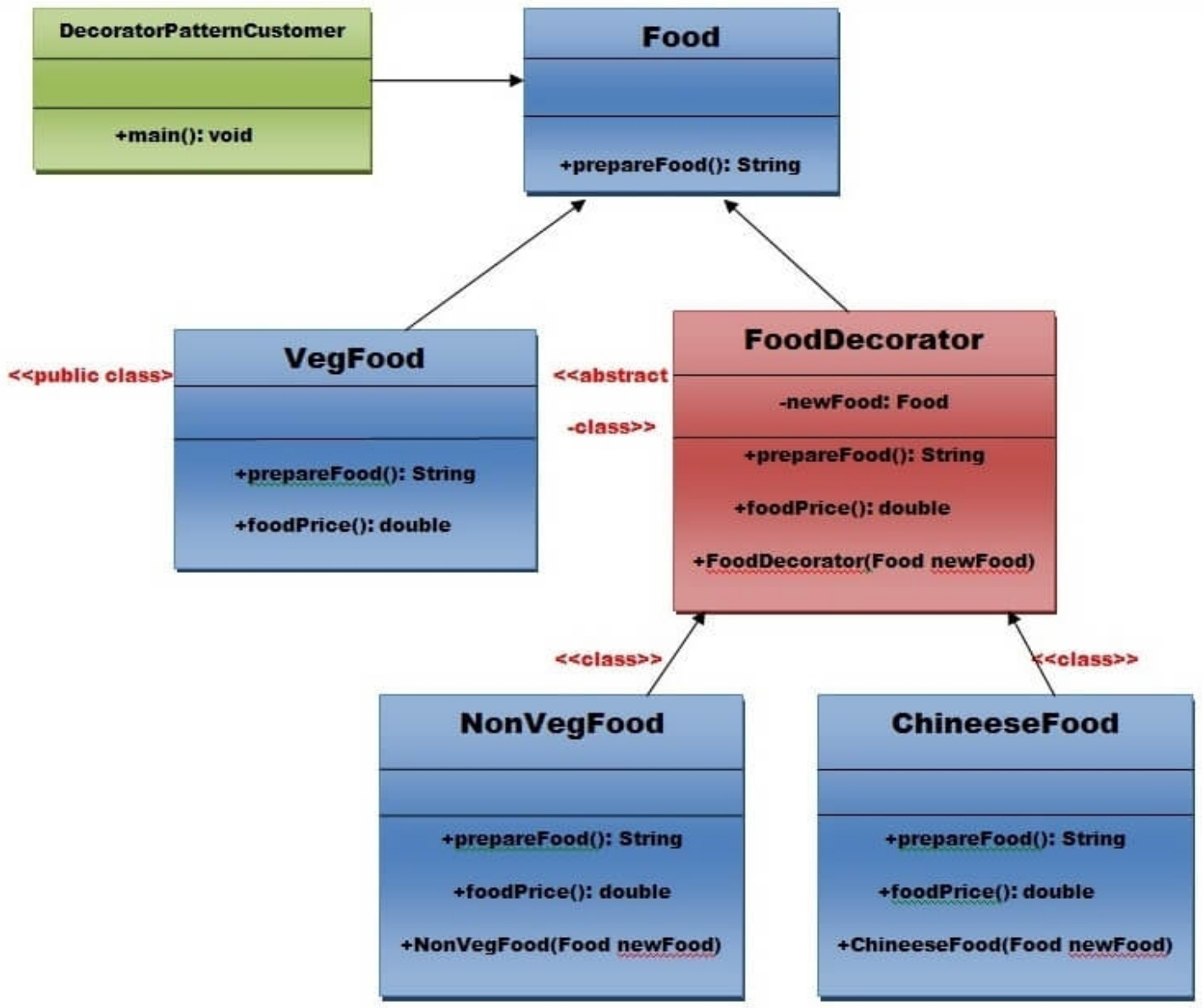


Decorator

- Introduction
  - attach a flexible additional responsibilities to an object dynamically
- Advantage
  - provides greater flexibility than static inheritance.
  - enhances the extensibility of the object, because changes are made by coding new classes.
  - simplifies the coding by allowing you to develop a series of functionality from targeted classes instead of coding all of the behavior into the object.
- Usage when
  - want to transparently and dynamically add responsibilities to objects without affecting other objects.
  - want to add responsibilities to an object that you may want to change in future.
  - Extending functionality by sub-classing is no longer practical

Decorator(contd)

- Example:

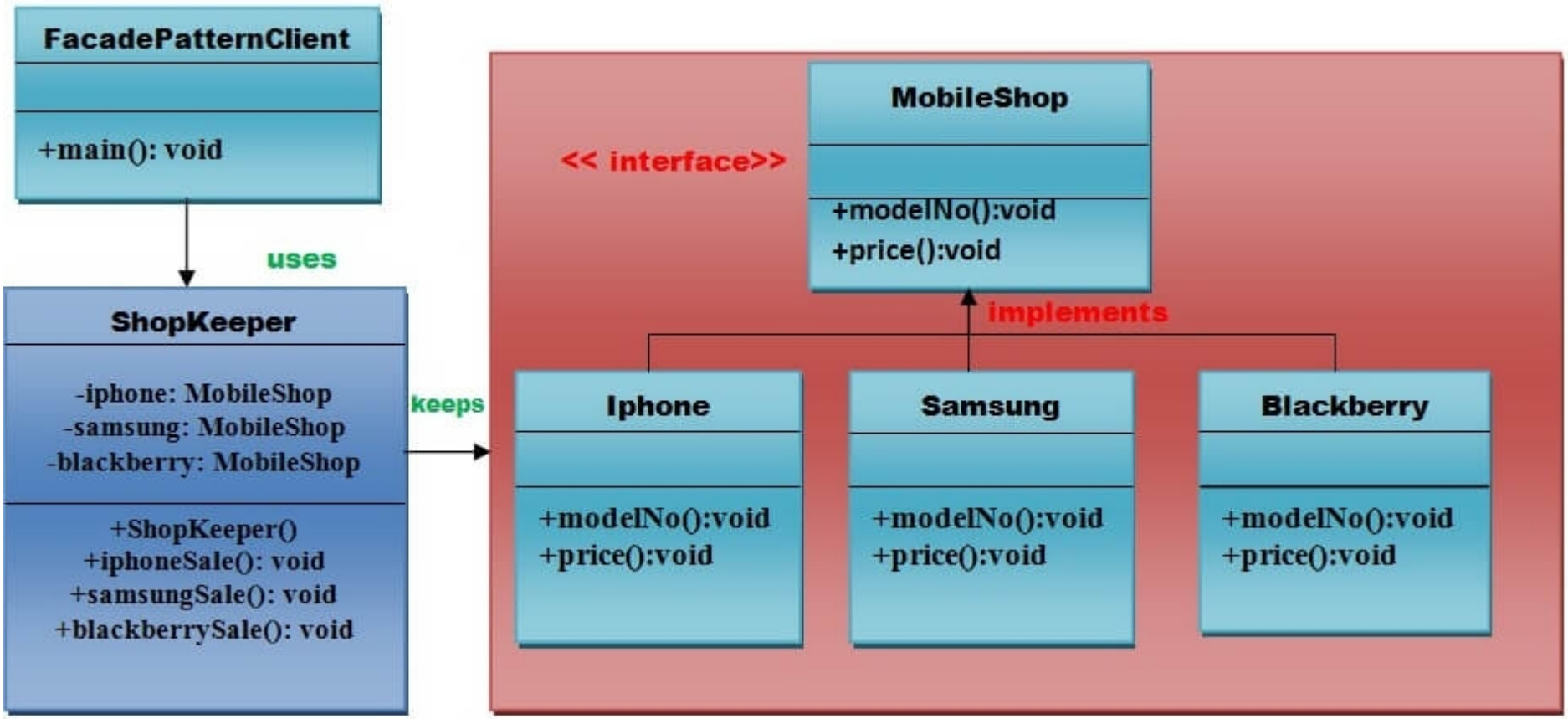


Facade

- Introduction
  - provide a unified and simplified interface to a set of interfaces in a subsystem, therefore it hides the complexities of the subsystem from the client
- Advantage
  - shields the clients from the complexities of the sub-system components.
  - promotes loose coupling between subsystems and its clients.
- Usage when
  - want to provide simple interface to a complex sub-system.
  - several dependencies exist between clients and the implementation classes of an abstraction.

Facade(contd)

- Example:

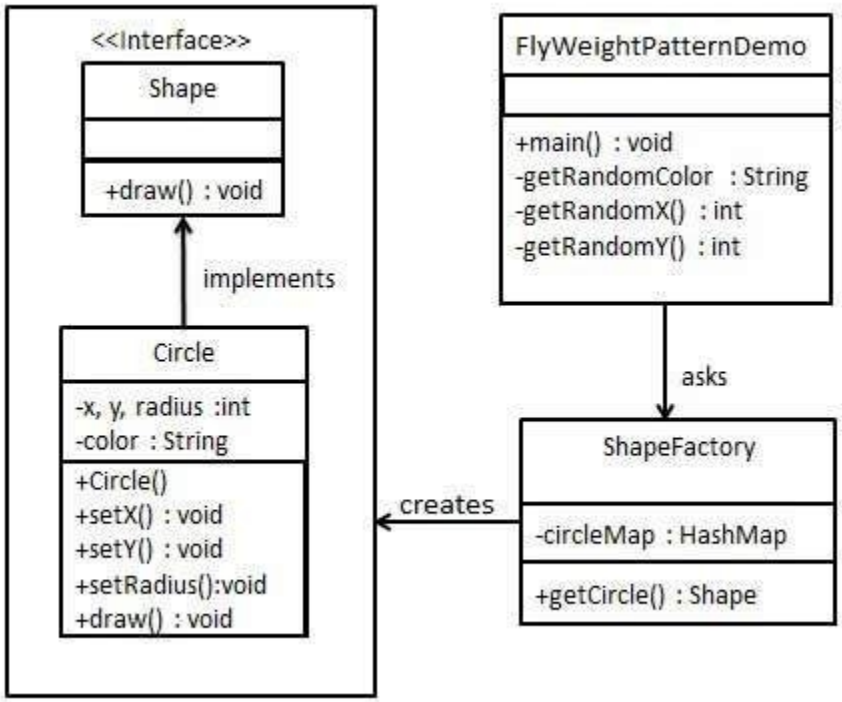


Flyweight

- Introduction
  - to reuse already existing similar kind of objects by storing them and create new object when no matching object is found
- Advantage
  - reduces the number of objects.
  - reduces the amount of memory and storage devices required if the objects are persisted
- Usage when
  - an application uses number of objects
  - the storage cost is high because of the quantity of objects.
  - the application does not depend on object identity.

Flyweight(contd)

- Example:

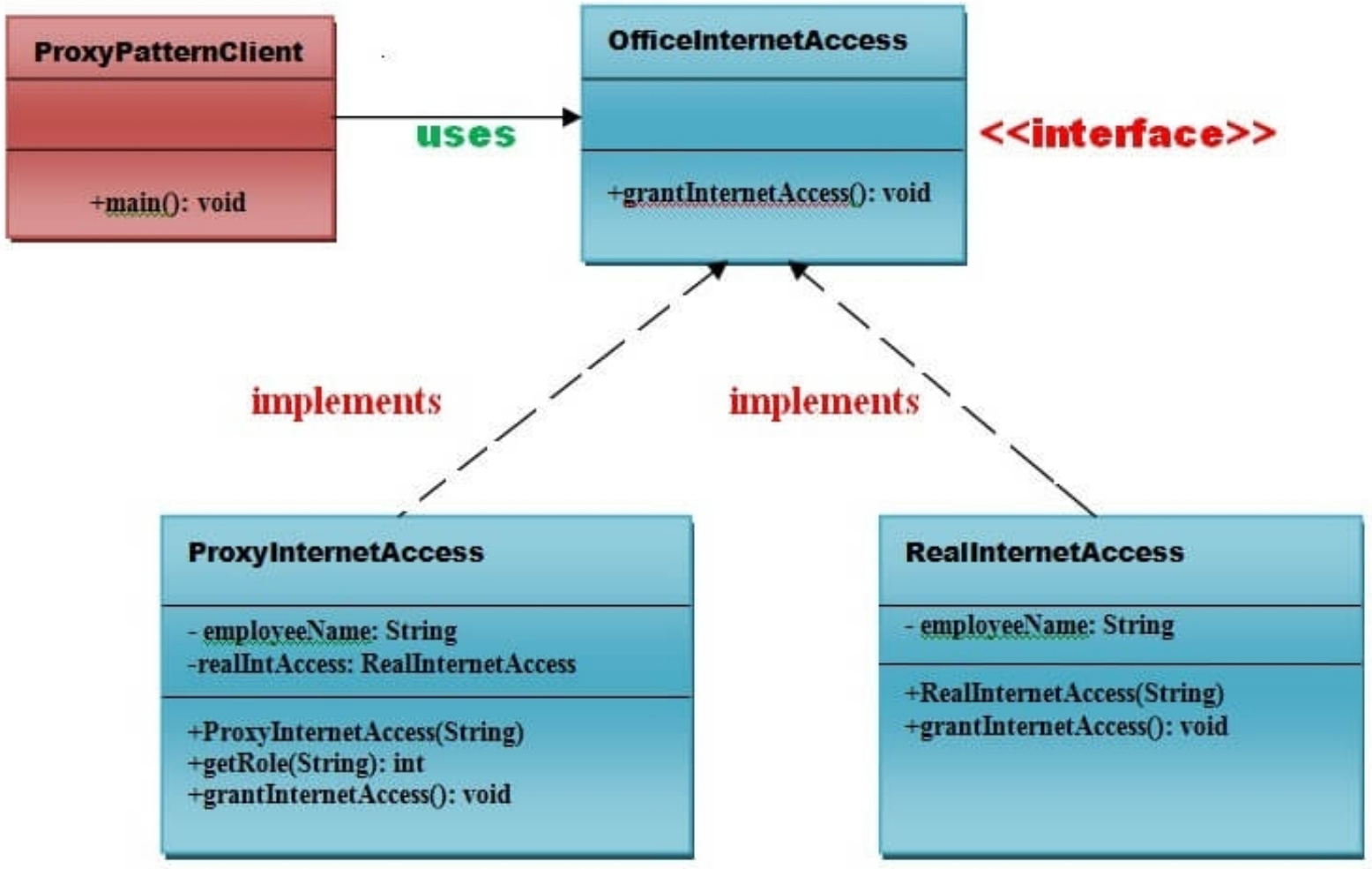


Proxy

- Introduction
  - provides the control for accessing the original object
- Advantage
  - provides the protection to the original object from the outside world
- Usage when
  - in Virtual Proxy scenario
  - in Protective Proxy scenario
  - in Remote Proxy scenario
  - in Smart Proxy scenario

Proxy(contd)

- Example:

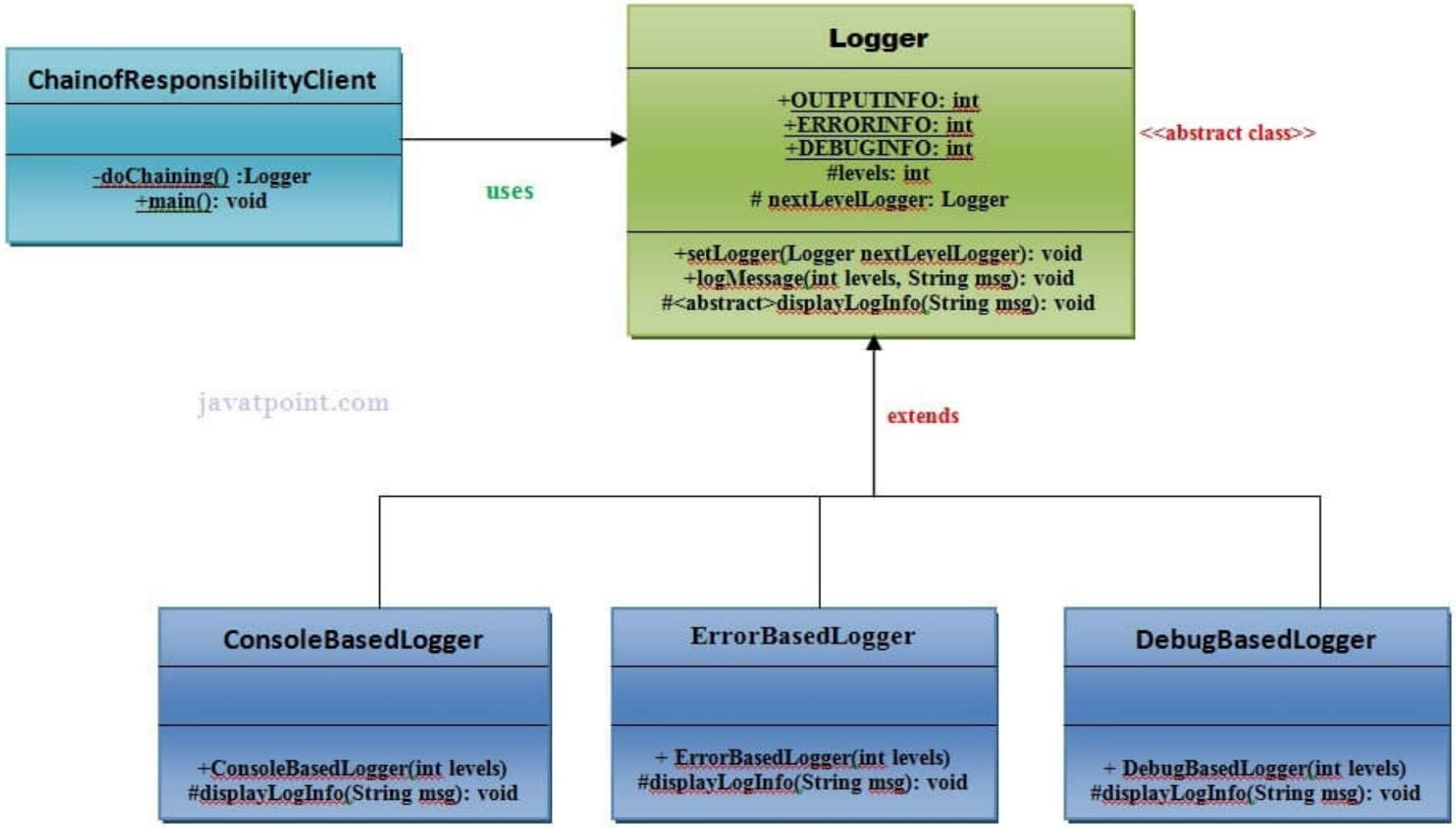


Chain of Responsibility

- Introduction
  - avoid coupling the sender of a request to its receiver by giving multiple objects a chance to handle the request
- Advantage
  - reduces the coupling.
  - adds flexibility while assigning the responsibilities to objects.
  - allows a set of classes to act as one; events produced in one class can be sent to other handler classes with the help of composition.
- Usage when
  - more than one object can handle a request and the handler is unknown.
  - the group of objects that can handle the request must be specified in dynamic way.

Chain of Responsibility(contd)

- Example:



Command

- Introduction
  - encapsulate a request under an object as a command and pass it to invoker object. Invoker object looks for the appropriate object which can handle this command and pass the command to the corresponding object and that object executes the command
- Advantage
  - separates the object that invokes the operation from the object that actually performs the operation.
  - makes easy to add new commands, because existing classes remain unchanged.
- Usage when
  - need parameterize objects according to an action perform.
  - need to create and execute requests at different times.
  - need to support rollback, logging or transaction functionality.

Command(contd)

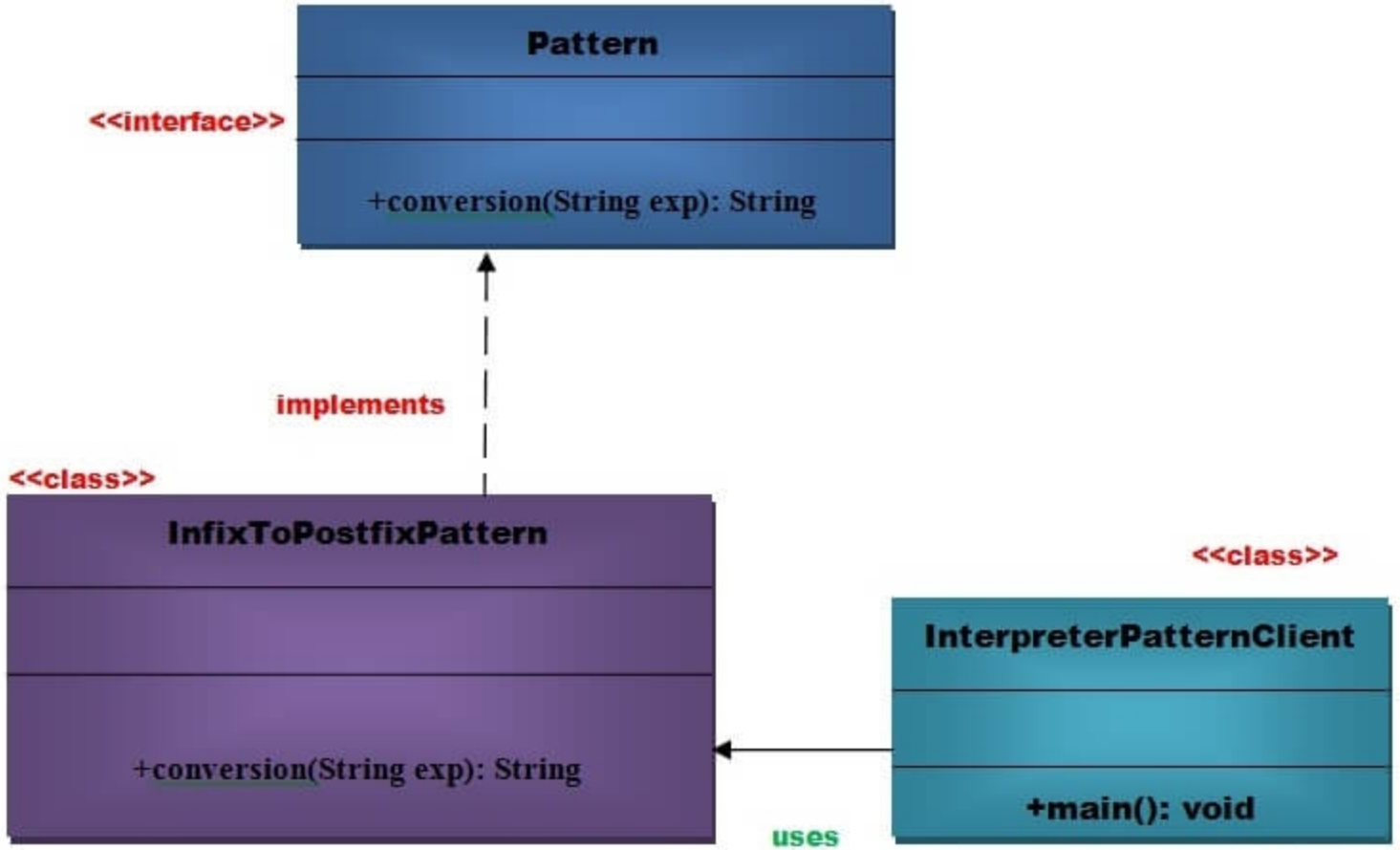
- Example:

Interpreter

- Introduction
  - to define a representation of grammar of a given language, along with an interpreter that uses this representation to interpret sentences in the language
- Advantage
  - easier to change and extend the grammar.
  - Implementing the grammar is straightforward.
- Usage when
  - the grammar of the language is not complicated.
  - the efficiency is not a priority.

Interpreter(contd)

- Example:

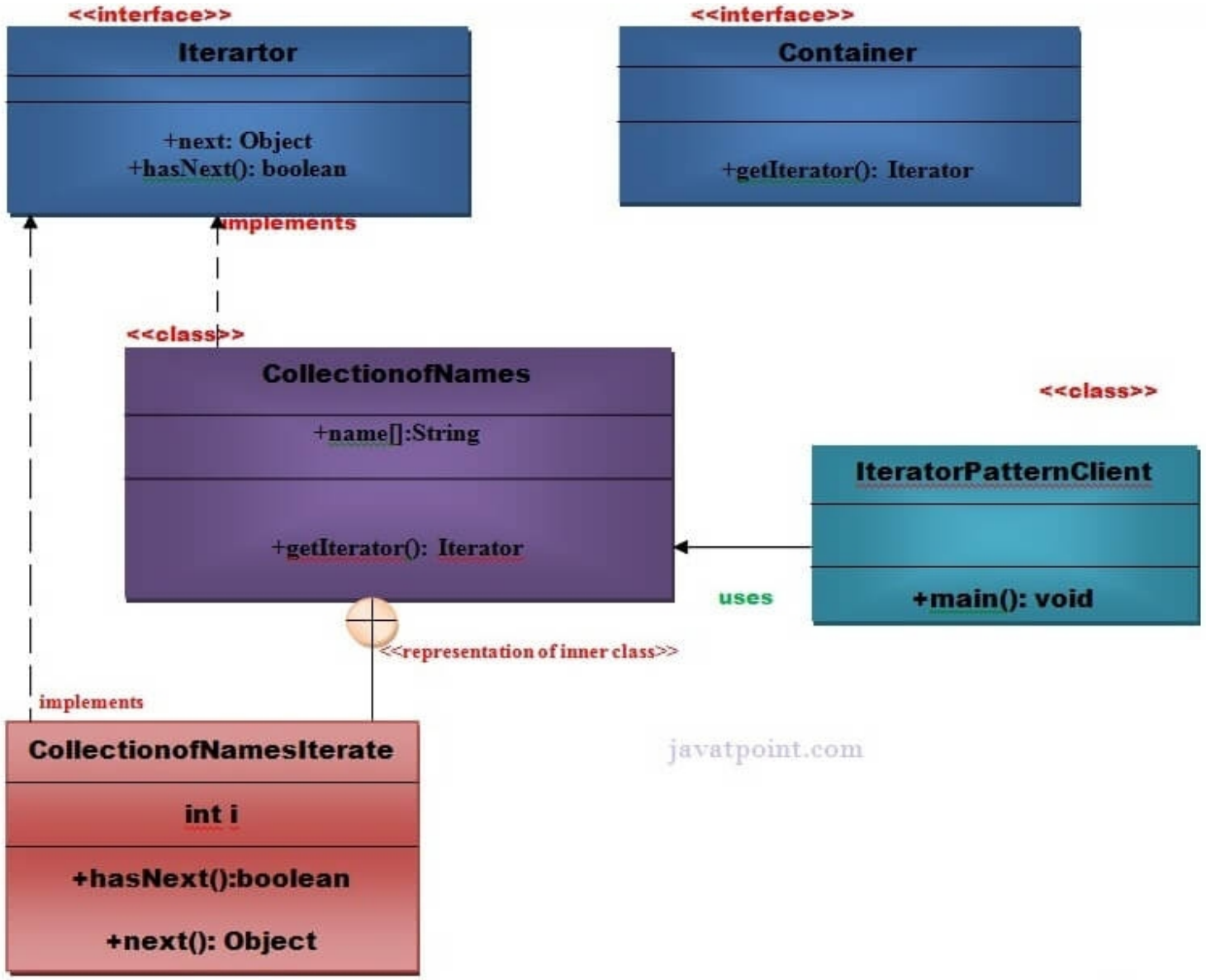


Iterator

- Introduction
  - to access the elements of an aggregate object sequentially without exposing its underlying implementation
- Advantage
  - supports variations in the traversal of a collection.
  - simplifies the interface to the collection.
- Usage when
  - want to access a collection of objects without exposing its internal representation.
  - there are multiple traversals of objects need to be supported in the collection.

Iterator(contd)

- Example:

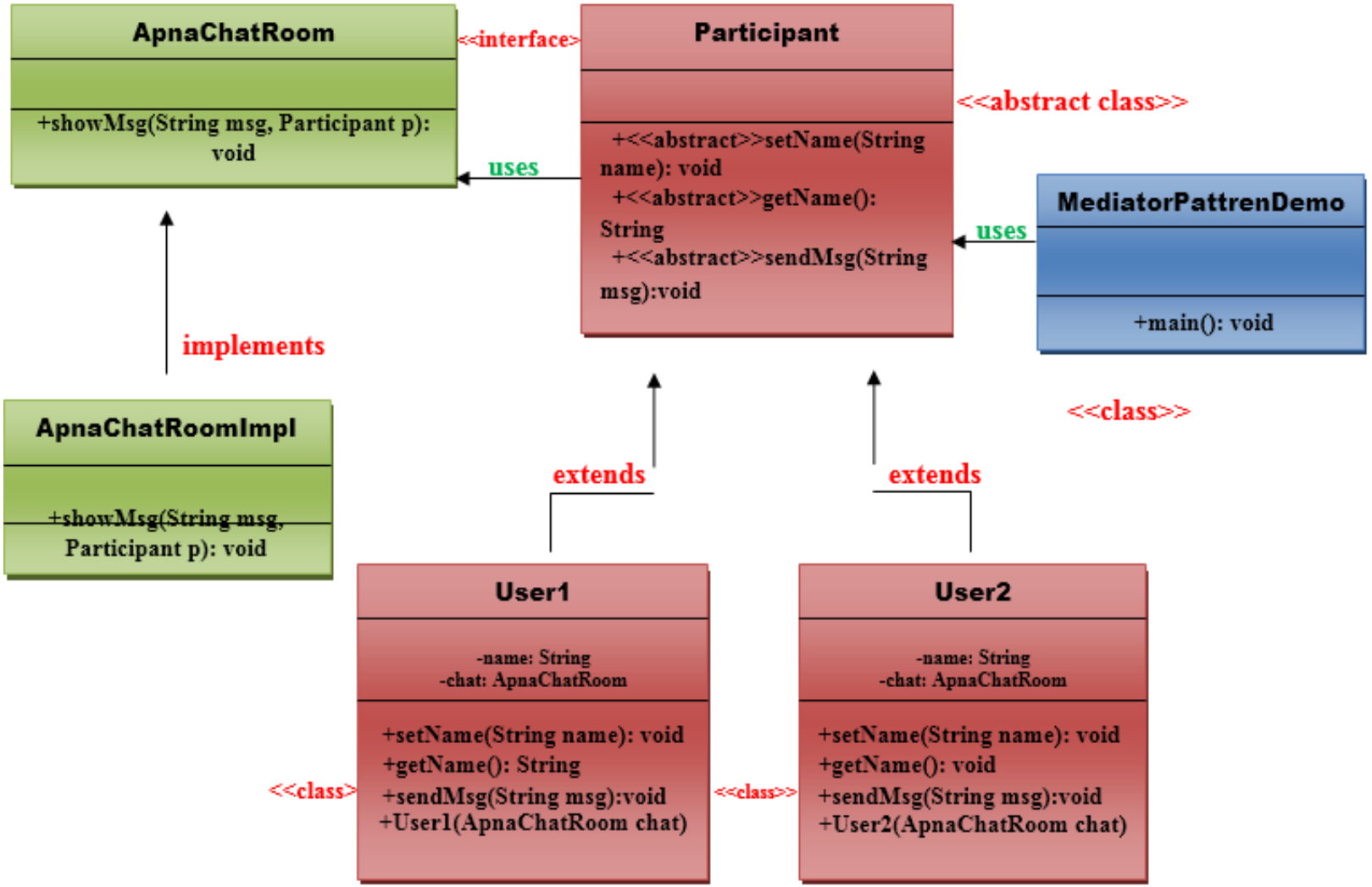


Mediator

- Introduction
  - to define an object that encapsulates how a set of objects interact
- Advantage
  - decouples the number of classes.
  - simplifies object protocols.
  - centralizes the control.
  - The individual components become simpler and much easier to deal with because they don't need to pass messages to one another.
- Usage when
  - commonly used in message-based systems likewise chat applications.
  - the set of objects communicate in complex but in well-defined ways.

Mediator(contd)

- Example:

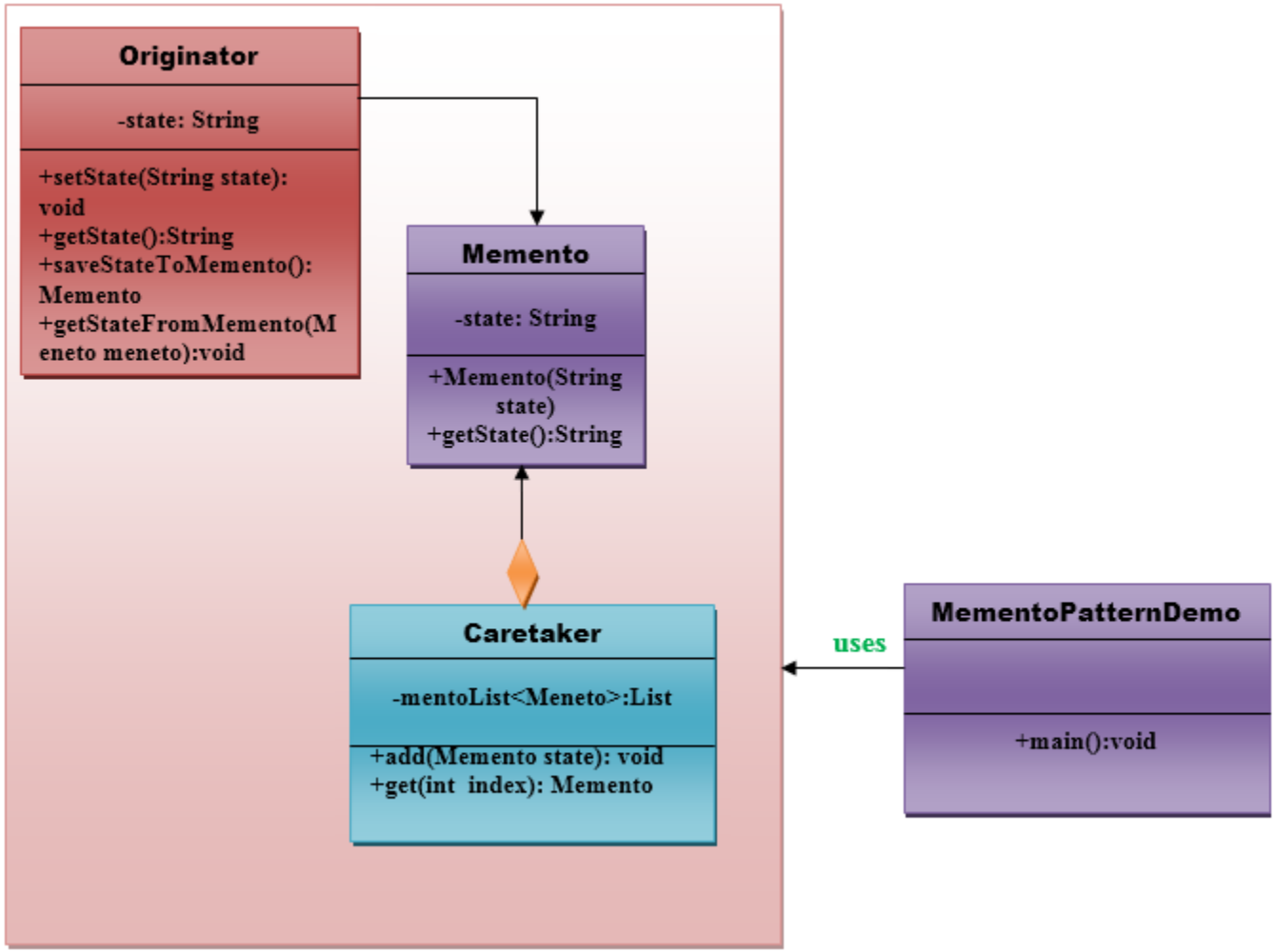


Memento

- Introduction
  - to restore the state of an object to its previous state
- Advantage
  - preserves encapsulation boundaries.
  - simplifies the originator.
- Usage when
  - in Undo and Redo operations in most software.
  - also used in database transactions.

Memento(contd)

- Example:

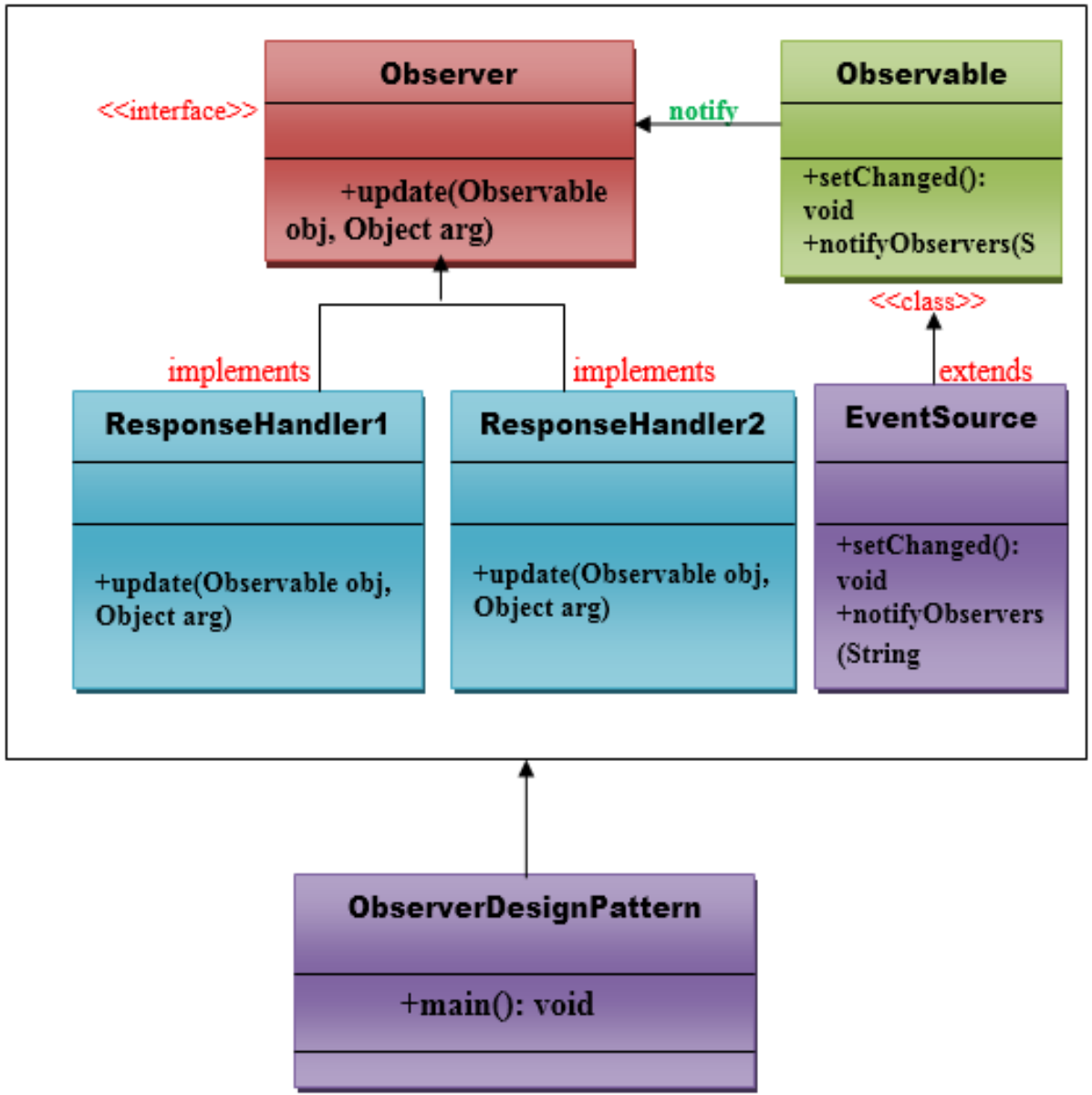


Observer

- Introduction
  - just define a one-to-one dependency so that when one object changes state, all its dependents are notified and updated automatically
- Advantage
  - describes the coupling between the objects and the observer.
  - provides the support for broadcast-type communication.
- Usage when
  - the change of a state in one object must be reflected in another object without keeping the objects tight coupled.
  - the framework we writes and needs to be enhanced in future with new observers with minimal chamges.

Observer(contd)

- Example:

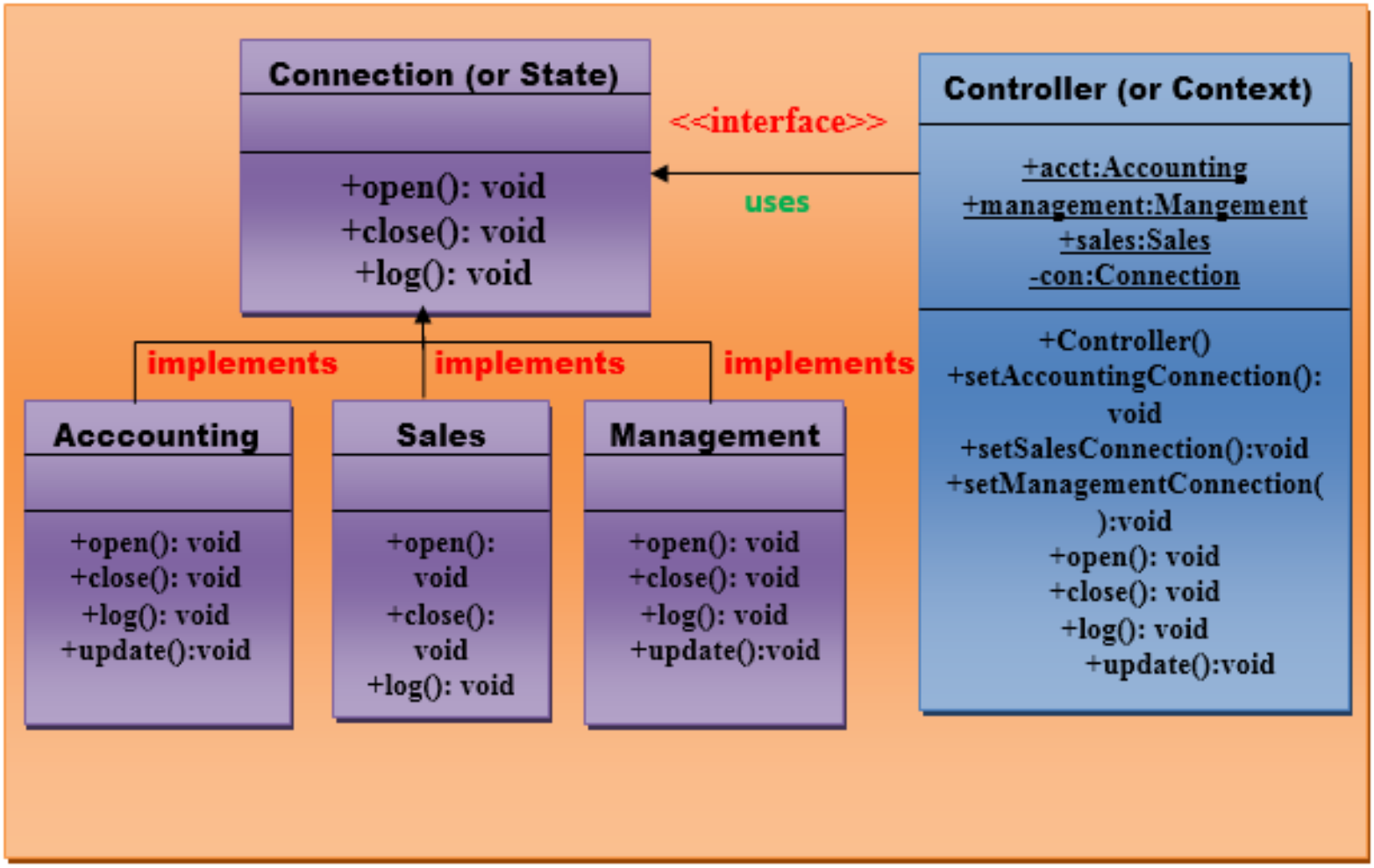


State

- Introduction
  - the class behavior changes based on its state"
- Advantage
  - keeps the state-specific behavior.
  - makes any state transitions explicit.
- Usage when
  - the behavior of object depends on its state and it must be able to change its behavior at runtime according to the new state.
  - the operations have large, multipart conditional statements that depend on the state of an object.

State(contd)

- Example:

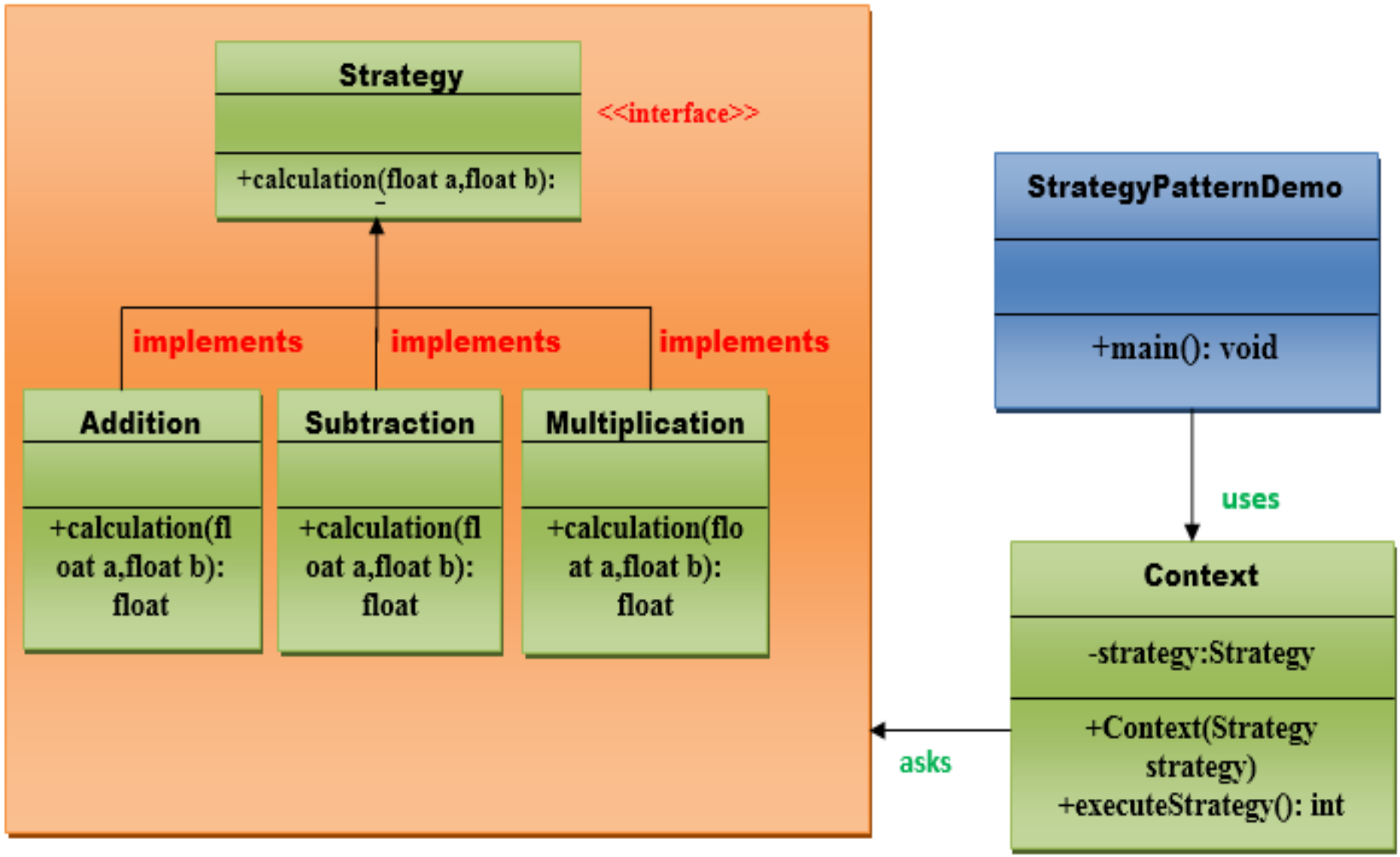


Strategy

- Introduction
  - defines a family of functionality, encapsulate each one, and make them interchangeable
- Advantage
  - provides a substitute to subclassing.
  - defines each behavior within its own class, eliminating the need for conditional statements.
  - easier to extend and incorporate new behavior without changing the application.
- Usage when
  - the multiple classes differ only in their behaviors.e.g. Servlet API.
  - different variations of an algorithm.

Strategy(contd)

- Example:



Template

- Introduction
  - define the skeleton of a function in an operation, deferring some steps to its subclasses
- Advantage
  - for reusing the code
- Usage when
  - the common behavior among sub-classes should be moved to a single common class by avoiding the duplication.

Template(contd)

- Example: