Answer 1:

Class Adapter	Object Adapter
A Class Adapter uses the concept of Inheritance in order to adapt an adaptee.It is created by implementing or inheriting both the expected interface and the existing interface.	An Object Adapter uses Composition to adapt the adaptee. It basically contains reference to the instance of the class it is wrapping and it works by making calls to this wrapped instance.
It can only wrap a class	It can wrap either an interface or a class
An interface can not be wrapped by a Class Adapter as it derives from a base class	An interface or a class can be wrapped up by an object adapter because it contains the class or interface object instance it wraps as a member
Class Adapters are not very flexible. There is only one of the class adapters and not an adapter or adaptee	Object Adapters are flexible. It can write a little code and delegate to the adaptee.
Class Adapters do not have to reimplement the entire adaptee. They can override the behavior of only their adaptee by subclassing	Object Adapter has to modify the adapter code to get any intended behavior. When this is done, the behavior gets added to the adapter code and works with the adaptee and all its subclasses

Answer 2:

A hook is sort of like a place holder function implementation in the base class. This ensures that the sub classes are not forced to implement the function unless they have a need for it. In other words we use hooks when a part of the algorithm is optional to be implemented by the subclass. So basically if the subclass doesn't need that functionality, it can be skipped.

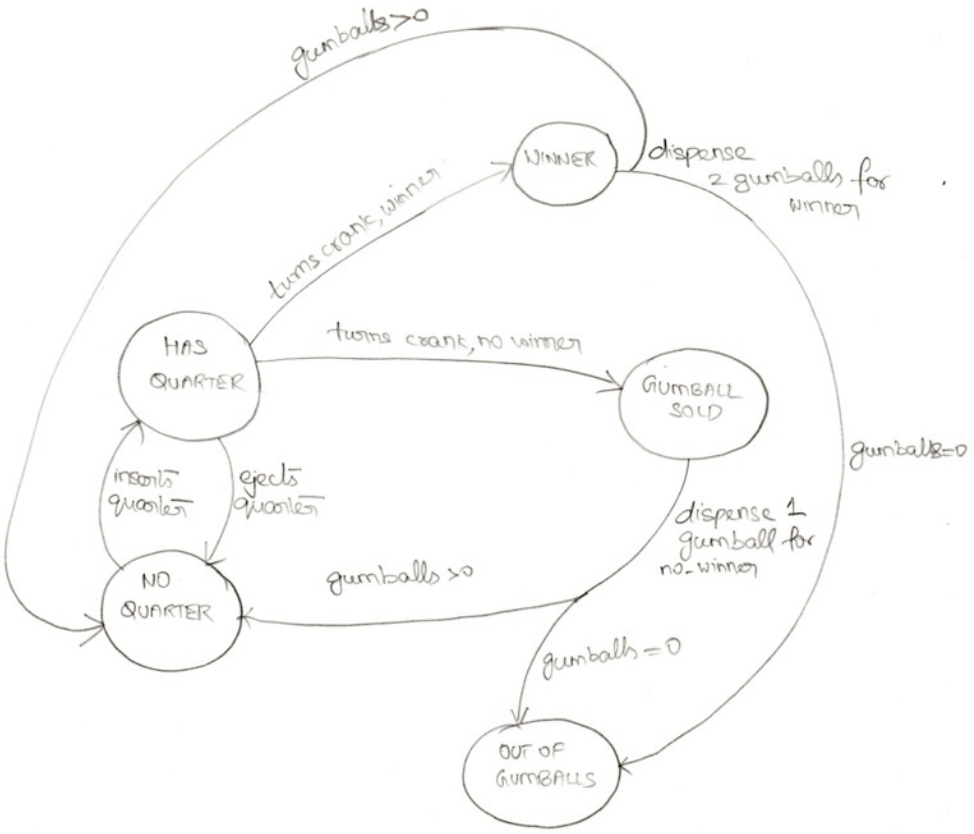
It can also be used to give a subclass the chance to react to some step in the template method that has happened or is about to happen. A hook can also provide a subclass with the ability to make a decision for an abstract class.

Answer 3:

A remove() method is considered an optional method, but since it is part of the interface, it has to be implemented. But the behavior of a remove() method is unspecified if the collection changes while iterating over the same collection of objects. For example, in a multithreaded environment, one thread could end up removing an item from a list while another thread is about to read it. As a result this will cause errors for the second thread, if not handled correctly.

It is the programmers prerogative to ensure that proper checks are put in place if there are possibilities of multiple access to a collection object in parallel.

Answer 4:



Answer 5:

a) javax.xml.parsers.DocumentBuilderFactory#newInstance()

Pattern Used: Abstract Factory

As per the definition for an Abstract Factory Pattern, it allows a client to use an interface for creating families of related or dependent objects without specifying their concrete class. So basically an Abstract Factory can be considered as a Factory that produces factories. The DocumentBuilderFactory follows the same principle. It is a factory that produces factories. The newInstance() method definition in the JavaDoc states that is used to create a new factory instance that is used to look up the actual DocumentBuilderFactory implementation class to load. This is a classic example of what an Abstract Factory pattern does.

b) java.lang.StringBuilder#append()

Pattern Used: Builder Pattern

A Builder Pattern is used when the algorithm for creating a complex object should be independent of the parts that make up the object. The StringBuilder class append() method basically allows for the implementation of various independent methods to handle different types of fields. The internal implementation of how it actually does this is hidden from the client. It allows for a more composite implementation as is provided by the Builder Pattern

c) java.util.Collections#unmodifiableXXX()

Pattern Used: Decorator Pattern

The goal of a Decorator pattern is to allow classes to be easily extended to incorporate new behavior without modifying the existing code. The Unmodifiable collection returns an unmodifiable view of the specified collection. So it is basically decorating a normal collection with a new behavior without actually modifying the existing code. Hence it is following a Decorator patten.

d) java.lang.Runtime#getRuntime()

Pattern Used: Singleton pattern

A singleton pattern is used when there should be one and only one instance of an object. The getRuntime() method definition states that it returns a runtime object associated with the current java application. Since there must be only one runtime object at any time and most of the methods of the Runtime class are instance methods, it is an example of a Singleton pattern.

e) java.util.Calendar#getInstance()

Pattern Used: Factory Method pattern

A Factory Method pattern defines an interface for creating an object but lets individual subclasses decide which classes to instantiate. The getInstance() method returns a calendar using the default time zone and locate. There are other methods which are used to get corresponding instances of the Calendar object based on their requirements, thereby allowing a subclass to decide which class to instantiate, as is provided b the Factory Method Pattern

f) javax.servlet.http.HttpSessionBindingListener

Pattern Used: Observer Pattern

An Observer pattern is used to define a dependency between objects and to notify dependent objects when the state of one of the objects changes. The dependent object need to be notified and updated automatically. A HttpSessionBindingListener causes an object to notified whenever it is bound or unbound from a session. This feature is reminiscent of the Observe pattern

g) java.util.Comparator#compare()

Pattern Used: Strategy pattern

The definition of the Strategy pattern implies that it is used to define a family of algorithms encapsulates each one and makes them interchangeable. The compare() methods compares its two arguments for order, while allowing the implementor to ensure various different functionalities. Since it allows for the logic to be handled by defining classes that encapsulate different algorithms, it is an example of the Strategy pattern.