# Assignment 03 Application Design: Patterns and Frameworks 44642

Answer **all** the questions below. In your answer for each question explain with sample code or image whichever is preferable.

1. What are generics?

Generics are a feature of programming languages that let you create code that can operate on several data types without having to declare the precise type until runtime.

Generics can be utilized in the context of patterns and designs to produce more adaptable and reusable code. For instance, you could use generics to build a class that can house any kind of object or a sorting method that can handle any kind of data.

Since you can reuse the same code for several sorts of data, generics help you develop more modular, maintainable programs. Additionally, by doing so, code efficiency and code duplication can be reduced.

Here's an example of how to use generics:

Graphical user interface, application, Word

Description automatically generated

Driver class:

Graphical user interface, text, application, email

Description automatically generated

1. Can we change the scope of the overridden method in the subclass for private, public, default and protected? Explain how can it be changed for each scope?

A method can be overridden in a subclass in Java, but the overridden method's scope cannot be altered. The superclass method's scope must match or be wider than the scope of the overridden method in the subclass.

Here is a list of Java's accessibility levels:

private: Only accessible within the class in which it is defined

default or package-private: Accessible within the same package

protected: Accessible both by subclasses inside and outside of the same package

public: Accessible from anywhere in the program

A method's behavior can be altered but its level of accessibility cannot be altered when it is overridden in a subclass. As an example, consider the following:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

In this example, we have a SuperClass with methods of different accessibility levels, and a SubClass that extends SuperClass and overrides some of its methods.

publicMethod(): We override this method in SubClass and change its behavior, but we cannot change its accessibility level, which remains public.

protectedMethod(): We override this method in SubClass and change its behavior, but we cannot make it less accessible. The scope remains protected.

defaultMethod(): We override this method in SubClass and change its behavior, but we cannot make it more or less accessible. The scope remains default.

privateMethod(): We cannot override this method in SubClass because it is private in SuperClass and not accessible from SubClass.

In conclusion, you can alter the behavior of methods in a subclass by overriding them, but you cannot alter their level of accessibility. The superclass method's scope must be equal to or wider than the scope of the overridden method in the subclass.

1. What is the covariant return type?

A feature of Java 5 called covariant return type enables a subclass method to return a more specific type than the method it overrides in the superclass. It is a means to increase the level of specificity in the method return type without affecting the subclass-superclass hierarchy or the restrictions on method overriding.

Here is an example of how a covariant return type works:

Graphical user interface, application, Teams

Description automatically generated

Graphical user interface, text, application, Teams

Description automatically generated

Graphical user interface, text, application, Teams

Description automatically generated

The reproduce() method of the Animal class in the current example returns an Animal object. With a more specialized return type of Mammal, the Mammal class extends Animal and replaces the reproduce() method. Because Mammal is a subtype of Animal, this is possible.

An Animal object will not be returned when you call the reproduce() function on a Mammal instance; instead, a Mammal object will be returned. The covariant return type feature keeps this possible.

When you want to give a more particular return type for a method in a subclass without violating the conditions for method overriding, a covariant return type can be helpful. It's frequently employed in factory methods and other methods that return objects of a particular kind.

1. Can we override the static and private methods? Why?

We cannot override static and private methods because they are resolved and accessed in different ways than non-static and non-private methods, and because private methods are not visible outside of the class in which they are defined.

Graphical user interface, application, Teams

Description automatically generated

Graphical user interface, application, Teams

Description automatically generated

Graphical user interface, application, Teams

Description automatically generated

(See the statement "Animal obj = new Dog ()") The Animal class reference 'obj' refers to an object of the Dog class. When the makeSound() method is invoked on an object, it is done so in accordance with the type of the referred object, not the reference.

1. Difference between String Buffer and StringBuilder?

|  |  |
| --- | --- |
| **StringBuffer** | **StringBuilder** |
| StringBuffer is synchronized i.e., thread safe. It means two threads can’t call the methods of StringBuffer simultaneously. | StringBuilder is non-synchronized i.e., not thread safe. It means two threads can call the methods of StringBuilder simultaneously. |
| StringBuffer is less efficient than StringBuilder. | StringBuilder is more efficient than StringBuffer. |

StringBuffer vs StringBuilder Performance Testing

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

1. Difference between String class and String Buffer?

|  |  |
| --- | --- |
| String Class | StringBuffer |
| The String class is immutable. | The StringBuffer class is mutable. |
| String is slow and consumes more memory when we concatenate too many strings because every time it creates new instance. | StringBuffer is fast and consumes less memory when we concatenate t strings. |
| String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method. | StringBuffer class doesn't override the equals() method of Object class. |
| String class is slower while performing concatenation operation. | StringBuffer class is faster while performing concatenation operation. |
| String class uses String constant pool. | StringBuffer uses Heap memory |

Graphical user interface, application, Word, Teams

Description automatically generated

1. Can we declare constructor as final?

No, we cannot declare a constructor as final.

A class, method, or variable cannot be subclassed, overridden, or reassigned, respectively, is denoted with the final keyword. Constructors are specialized methods used to generate new instances and initialize objects of a class. Constructors aren't inherited by subclasses by design, and their behavior can't be changed.

Declaring a constructor as final would be redundant and pointless as a result. The behavior of the constructor or the class would not be affected.

In result, we cannot define a constructor to be final. The syntax of the Java language restricts that, and it would be useless.

To illustrate this, let's consider an example code snippet:

Graphical user interface, application

Description automatically generated

It will fail to compile this code with the message "Modifier 'final' not allowed here" when we attempt to do so. We can't declare constructors to be final under the Java language specification, which is why.

Instead, we can employ additional methods, such as naming a class final or making its constructors private, if we want to restrict access to its constructors or prevent it from being subclassed. Consider this:

Graphical user interface, application, Word

Description automatically generated

This class cannot be subclassed because it has been declared final in the example. Additionally, we have restricted the ability to create new instances to methods inside of the class by making the constructor private, which means it cannot be accessed from outside the class.

In conclusion, although constructors cannot be declared final in Java, there are still ways to write code that has the same effect.

1. Can we have try without catch block in java?

In Java, a try block can exist without a catch block, so the answer is yes. When using a try block without a catch block, a finally block must be used to guarantee that resources are released or cleaned up properly, regardless of whether an exception is thrown or not. No of whether an exception is raised or not, the finally block is always carried out following the try block.

Here's an illustration code snippet:

Graphical user interface, text, application

Description automatically generated

In this illustration, the try block contains some code that could raise an exception. Although a catch block is not there, we have a finally block that will be run when the try block is finished, whether or not an exception is thrown. The finally block can be used to close a file or release a lock, among other resources that were used in the try block.

Exceptions thrown in the try block will propagate up the call stack until they are captured by an enclosing catch block or until they cause the program to stop, it is vital to note. Because of this, if we use a try block without a catch block, we need to make sure that the exception is caught elsewhere in the program or handled in a way that prevents program termination.

In conclusion, Java allows us to have a try block without a catch block, but we still need a finally block to make sure that resources are correctly released or cleaned up and that any thrown exceptions are caught or handled in a way that prevents the program from terminating.

1. What is try with the resource?

With the introduction of "Try with resources" in Java 7, it is now possible to automatically handle resources that need to be closed, such as file streams, network sockets, database connections, etc. It makes the code simpler and aids in preventing frequent mistakes that could happen when handling resources manually.

Graphical user interface, text, application, email

Description automatically generated

1. Can we modify the throws clause of the superclass method while overriding it in the subclass?

Yes, we can modify the throws clause when overriding the superclass method in the subclass. But when handling exceptions, there are a few rules that must be followed when overriding.

If the superclass method does not declare an exception, the subclass overridden method may declare the unchecked exception but not the checked exception.

If the overridden subclass method of the superclass declares an exception, it may do the same, declare none, or declare a subclass exception, but not a parent exception.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

1. What is an association, aggregation, and composition in UML?

Associations, aggregations, and compositions are three different kinds of relationships that can occur between classes or objects in the UML (Unified Modeling Language).

Association: An association is a relationship between two or more objects in which one object is conscious of the other object or objects and has the ability to communicate with them. Associations can be navigable in one direction or both, and they might be one-way or two-way. For instance, if every student is enrolled in one or more courses, a Student class might be connected to a Course class.

Aggregation: Aggregation is a specific kind of relationship in which one thing is considered to be "part" of another object. The part object may exist separately from the entire object and may also be a component of other whole objects. For instance, if each automobile has four wheels, an automobile class and a Wheel class may aggregate.

Composition: Another unique sort of linkage is composition, when one thing is a "part" of another object but cannot exist separately from the full object. All of an object's components are destroyed together with the whole when it is destroyed. For instance, the composition of a House class and a Room class might be such that each house is made up of one or more rooms, and that a room cannot exist without a house.

In summary, there are three main kinds of links that can exist between classes or objects in the UML: associations, aggregations, and compositions. Aggregations are "part-of" relationships where the part object can exist independently of the whole item, compositions are "part-of" relationships where the part object cannot exist independently of the whole object, and associations are general relationships between things.

1. Difference between final, finally and finalize()?

Final, Finally, and finalize() are three distinct concepts in Java that are sometimes mixed together. Let's examine how these ideas differ from one another:

Final: The keyword "final" denotes that a variable, method, or class cannot be altered or overridden. Once a variable's value has been assigned, it cannot be changed, and when a method or class is designated final, it cannot be overridden by any subclasses.

Finally: A block of code called finally is used to guarantee that a part of code is always performed, regardless of whether an exception is thrown or not. Even if an exception is thrown and not caught, the try block and any associated catch blocks come before the finally block in the execution order.

finalize(): The garbage collector will invoke the method finalize() just before destroying an object. Before the object is destroyed, this method can be customized to carry out any required resource releases or cleanup tasks.

Finally, the words "final" and "finally" are used to denote things that cannot be altered or overridden, "always" ensures that a line of code is always performed, and "finalize()" is a method that the garbage collector calls just before an object is destroyed.

1. Difference between Vector and ArrayList?

|  |  |
| --- | --- |
| ArrayList | Vector |
| ArrayList is**not synchronized.** | Vector is **synchronized** |
| If the array's maximum number of elements is reached, the ArrayList increases the size of the array by 50%. | If the total number of elements exceeds the array's limit, a vector increment of 100% signifies that the array size is doubled. |
| ArrayList is **not a legacy** class. It is introduced in JDK 1.2. | Vector is a **legacy** class. |
| ArrayList is fast because it is non-synchronized. | Vector is slow because it is synchronized, i.e., in a multithreading environment, it holds the other threads in runnable or non-runnable state until current thread releases the lock of the object. |
| ArrayList uses the **Iterator** interface to traverse the elements. | A Vector can use the **Iterator** interface or **Enumeration** interface to traverse the elements. |

1. What are the different ways to make ArrayList methods synchronized?

Because ArrayList is not thread-safe by default, multiple threads may access its functions concurrently without any guarantees of thread safety. However, there are a number of techniques to synchronize an ArrayList.

* Making use of the Collections.The synchronizedList() method, which is supported by the supplied list, returns a synchronized (thread-safe) list.
* Using the synchronized keyword: The list object itself can be synced using the synchronized keyword. When calling methods from the list, this is possible.
* Using a thread-safe alternative: You can use a thread-safe substitute for ArrayList, such as Vector or CopyOnWriteArrayList.

1. Difference between Hash table and Hash Map?

|  |  |
| --- | --- |
| HashMap | Hashtable |
| 1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code. | Hashtable is **synchronized**. It is thread-safe and can be shared with many threads. |
| 2) HashMap allows one null key and multiple null values. | Hashtable doesn't allow any null key or value. |
| 3) HashMap is a new class introduced in JDK 1.2. | Hashtable is a legacy class. |
| 4) HashMap is fast. | Hashtable is slow. |
| 5) We can make the HashMap as synchronized by calling this code  Map m = Collections.synchronizedMap(hashMap); | Hashtable is internally synchronized and can't be unsynchronized. |
| 6) HashMap is traversed by Iterator. | Hashtable is traversed by Enumerator and Iterator. |
| 7) Iterator in HashMap is fail-fast. | Enumerator in Hashtable is not fail-fast. |
| 8) HashMap inherits AbstractMap class. | Hashtable inherits Dictionary class. |

1. In Java 8, explain how Hasp Map internally works?

In Java 8, HashMap uses an array of linked nodes/buckets to store the key-value pairs. Each node in the array represents a bucket, which contains a linked list of nodes with the same hash code. Here's a brief overview of how HashMap works internally in Java 8:

When a new HashMap is created, it initializes an array of a default size (16) and a default load factor (0.75).

To insert a key-value pair, the key is first hashed using the hashCode() method. The hash code is then used to calculate the index of the bucket in the array where the key-value pair will be stored. This is done using the formula (n - 1) & hash, where n is the length of the array.

If the bucket is empty, a new node is created with the key-value pair and added to the bucket.

If the bucket is not empty, a linked list of nodes is traversed to find the node with the same key as the one being inserted or updated. If a matching node is found, its value is updated. If no matching node is found, a new node is created with the key-value pair and added to the linked list.

When the number of key-value pairs in the map exceeds the load factor, the array is resized by creating a new array with double the size of the old array and rehashing all the key-value pairs.

To retrieve a value from the map, the key is hashed to find the index of the bucket where the key-value pair is stored. The linked list in the bucket is then traversed to find the node with the same key as the one being searched for.

If a matching node is found, its value is returned. If no matching node is found, null is returned.

In Java 8, HashMap also uses a technique called treeification to improve the performance of the get() and put() operations when the number of nodes in a bucket exceeds a certain threshold. When a bucket contains more than eight nodes, the linked list is transformed into a tree to reduce the time complexity of searching and inserting nodes.

Graphical user interface, text, application, email

Description automatically generated

1. Difference between fail fast and fail-safe iterator?

In Java, an iterator is an object that enables iteration over a collection, and it allows us to access the elements of a collection one by one. There are two types of iterators in Java: fail-fast iterators and fail-safe iterators.

Fail-Fast Iterator:

A fail-fast iterator is an iterator that throws a ConcurrentModificationException when it detects that a collection has been modified while it is iterating over it. Fail-fast iterators operate on the original collection and are sensitive to changes made by other threads. They are called fail-fast because they immediately fail as soon as a concurrent modification is detected.

Fail-Safe Iterator:

A fail-safe iterator is an iterator that creates a copy of the collection when it is created and iterates over the copy instead of the original collection. Fail-safe iterators are not affected by modifications made to the original collection and will not throw a ConcurrentModificationException. They are called fail-safe because they can safely continue iterating even if changes are made to the original collection.

Graphical user interface, text, application

Description automatically generated

1. Can we start the thread twice?

No, we cannot start a thread twice in Java. If we try to start a thread that has already been started, it will throw an IllegalThreadStateException.

Once a thread has been started, it begins executing its run() method. If we try to start the same thread again, we would essentially be attempting to execute the same run() method multiple times simultaneously, which can lead to unpredictable behavior and can cause the program to crash.

If we need to run the same logic multiple times, we can create a new thread each time and start it. Alternatively, we can create a loop inside the run() method that will allow the same logic to be executed multiple times within the same thread.

Graphical user interface, text, application

Description automatically generated

1. What are the different ways to create a thread in java? Which one is preferred?

There are two ways to create a thread in Java:

Extending the Thread class:

We can create a thread by creating a class that extends the Thread class and overrides its run() method. We can then create an instance of our class and call the start() method to start the thread.

Implementing the Runnable interface:

We can also create a thread by implementing the Runnable interface and passing an instance of our class to a Thread constructor. The run() method should be defined inside the Runnable class.

The preferred way to create a thread is by implementing the Runnable interface, as it allows for more flexibility and does not limit the inheritance hierarchy of the class. When we extend the Thread class, we cannot extend any other classes, whereas implementing the Runnable interface does not restrict us from extending other classes.

In addition, implementing the Runnable interface promotes better separation of concerns, as the Runnable object can be passed to different threads, which can execute the same run() method in parallel. This allows for more efficient use of system resources and better scalability.

In summary, while both ways are valid for creating threads in Java, implementing the Runnable interface is generally preferred due to its greater flexibility and separation of concerns.

1. What are the different states a thread will go through?

In Java, a thread can go through different states as it is executed. The different states are:

New: When a thread is created but not yet started, it is in the new state.

Runnable: When a thread has been started and is executing, it is in the runnable state. However, it may not be currently executing if the CPU is busy with other threads.

Blocked: When a thread is waiting for a monitor lock to be released, it is in the blocked state. This can happen if the thread is waiting to enter a synchronized block that is currently held by another thread.

Waiting: When a thread is waiting for another thread to perform a particular action, it is in the waiting state. This can happen if the thread is waiting for a specific condition to be met or if it is waiting for a specific amount of time.

Timed waiting: When a thread is waiting for a specific amount of time to pass, it is in the timed waiting state. This can happen if the thread is waiting for a specific amount of time using a method like Thread.sleep() or Object.wait().

Terminated: When a thread has completed its execution or has been terminated, it is in the terminated state.

It is important to note that the transition between these states is not always linear, and a thread can move back and forth between states as it is executed.

1. What is Serialization? How do we achieve it?

Serialization is the process of turning an object into a stream of bytes so that it may be quickly transferred across a network or saved in a file. The java.io package in Java is used to implement serialization.Interface that is serializable.

Simply having the class implement the Serializable interface will make an object serializable. This interface is a marker interface that informs Java that the object can be serialized; it has no methods.

Diagram

Description automatically generated

Serialization is useful when we need to transmit objects over a network or store them in a file. By converting an object to a stream of bytes, we can easily send it over a network or write it to a file. Deserialization is the reverse process of converting a stream of bytes back into an object.

1. What is immutable class? Is String class immutable?

An immutable class is a class whose instances cannot be modified once they are created. In other words, an object of an immutable class is read-only and cannot be changed after it is created.

An immutable class has the following properties:

* The state of an object cannot be modified after it is created.
* All fields of an immutable class must be final.
* All fields of an immutable class must be initialized in the constructor.
* An immutable class cannot have any setters.

The String class in Java is an immutable class. Once a String object is created, its state cannot be changed.

1. Do immutable classes thread safe? If yes then how?

Yes, immutable classes are intended to be thread-safe. As a result, multiple threads can interact with and modify instances of an immutable class without risking race situations or synchronization problems.

Because they are read-only and cannot be updated after being created, immutable classes are thread-safe. As a result, it is impossible for multiple threads to attempt to edit the same immutable class instance at the same time. Additionally, final fields—which are automatically thread-safe in Java—are typically used to create immutable classes.

1. Can we call the garbage collector explicitly? Will it trigger the garbage collector?

Although Garbage Collector can be contacted directly, JVM controls whether or not to handle the call. Never, ever, ever use a garbage collection call in your code. The JVM uses an algorithm internally to decide when to make this call. Calls to system.gc() are merely suggestions to the JVM, which is free to disregard them whenever it wants.

Graphical user interface, text, application

Description automatically generated

1. What are Java 8 features? Explain all of them with examples?

i)forEach () Method in Iterable Interface:

The forEach() method was introduced in Java 8 in the Iterable interface. It is used to iterate over the elements of a collection and perform a given action on each element. The forEach () method takes a lambda expression or a method reference as an argument.

ii) Optional Class:

To handle null values, Java 8 added a new feature called the Optional class. It is used to symbolize a value that might or might not exist. It offers ways to determine whether a value is present, to obtain the value if it is, to supply a default value if the value is not there, and to retrieve the value if it is not.

iii) Lambda expression:

In order to offer an efficient manner of writing code that uses a single method interface, lambda expressions were introduced in Java 8. A lambda expression is an anonymous function that can be used to construct functional interfaces. It can be passed around like a variable.

iv) Java Functional Interfaces:

The term "functional interface" refers to interfaces with a single abstract method. The java.util.function package was included in Java 8 and contains a number of predefined functional interfaces. These interfaces can serve as the foundation for method references and lambda expressions.

v) Java Date Time API:

In order to provide a better method of handling date and time values, the Java Date Time API was created in Java 8. Classes like LocalDate, Local Time, LocalDateTime, ZonedDateTime, Duration, and Period are included.

1. How to make a pure singleton?

A design pattern known as a pure singleton permits the creation of just one instance of a class during the course of an application. A pure singleton pattern can be implemented in Java in a number of different ways.

Lazy initialization of a singleton means that the instance of the singleton class is created only when required. Here is an example of implementing a singleton with lazy.

Eager initialization: Eager initialization of a singleton means that the instance of the singleton class is created at the time of class loading. Here is an example of implementing a singleton with eager initialization

Graphical user interface, text, application, email

Description automatically generated

1. How to make a singleton synchronized?

We must employ a double-checked locking technique in our implementation to synchronize a singleton. With this method, even in a multithreaded environment, a single instance of the singleton is produced.

The getInstance() method is synchronized in this implementation. As a result, the method can only be executed by one thread at once. The volatile keyword is also used to guarantee that changes made to an instance variable by one thread are immediately visible to all other threads. Finally, to guarantee that only one instance of the singleton is created, we employ the double-checked locking mechanism.

Graphical user interface, text, application

Description automatically generated

**Submission:**  Change the document name as ***Lastname*Assignment03** where *Lastname* is your Last name and submit.

GitHub Link: