**Knowledge and Theory : Java Language and Syntax**

**14.1 *In terms of inheritance, what is the effect of keeping a constructor private?***

It will make sure that no one outside the class can directly instantiate the class. In this case, the only way to create an instance of the class is to provide a static public method, as is done when using the Factory Method Pattern. In addition, because the constructor is private, the class also cannot be inherited.

***14.2 In Java, does the finally block get executed if we insert a return statement inside the***

***try block of a try-catch-finally?***

Yes, it will get executed. The finally block gets executed when the try block

exits. Even when we attempt to exit within the try block (via a return statement, a

continue statement, a break statement or any exception), the finally block will still

be executed. The only way to avoid this if the virtual machine exits during try/catch block execution or the thread gets killed.

***14.3 What is the difference between final, finally, and finalize?***

* **Final applied to a variable (primitive):** The value of the variable cannot change.
* **Final applied to a variable (reference):** The reference variable cannot point to any other object.
* **Final applied to a method:** The method cannot be overridden.
* **Final applied to a class:** The class cannot be inherited.
* **Finally:** There is an optional finally block after the try block or after the catch block.
* **Finalize:** The finalize() method is called by the garbage collector when it determines that no more references exist. It is typically used to clean up resources, such as closing a file.

**14.4 *Explain the difference between templates in* C++ *and generics in Java.***

|  |
| --- |
| **Generics In Java:**  The following code shows the issue with not using generics.  List v = new ArrayList();  v.add("test");  Integer i = (Integer)v.get(0); // Run time error  Although the code is compiled without error, it throws a runtime exception (java.lang.ClassCastException) when executing the third line of code. This type of problem can be avoided by using generics and is the primary motivation for using generics. You can have compile time issues instead of run time issues by telling the compiler that this list has strings.  Using generics, the above code fragment can be rewritten as follows:  List<String> v = new ArrayList<>();  v.add("test");  Integer i = v.get(0); // (type error) compilation-time error    At runtime, the generics are removed again. Java generics didn’t actually allow us to do anything we couldn’t already do. This concept is called type erasure. It is when you add generics to reduce run time errors, then you erase those generics (or type information) during compilation.  A method is never really generic. What I mean by generic is that it can take to different objects of different types. What it really does is take a common object between these types. For example, suppose you had a method called “sum to string”. Suppose this method takes in two generic objects and it returns their concatenated “to string” calls. Java tries to pretend that this method is generic and can take in any object but it really is not. It can take any object of type “object”. So it is really just syntactic sugar when in reality it requires two object types. If you throw an “int” in the parameter, it would be like “woah I only handle classes of type Object”. |

|  |
| --- |
| **Templates in C++:**  template <typename T> T sum(T a, T b) { return a + b; }  The above is actually generic. It can take a string or an int (to types that are completely different and don’t extend a common class) and actually do two different things, as long as they both have the ‘+’ operator. |

**Summary:**

🡪 Java generics are just really syntactic sugar. They get removed at runtime which is called type erasure. When you see a class that takes in a method <T> it really just is expecting a class of type Object.

-> C is really generic. You pass in two completely different types and it will still perform the task.

**14.5 *Explain what object reflection is in Java and why it is useful.***

* Getting an unknown objects methods, fields and class at runtime.
* Creating an instance of a class
* Getting and setting the objects fields directly without having to care about what the access modifiers are (public, private and protected are access modifiers).
* Calling methods on objects without caring about their acces modifiers.

|  |
| --- |
| Class myObjectClass = myObject.**class**; //Given an unknown object, get its class  Method[] method = myObjectClass.getMethods();//get the all the methods in the class  //Here the method takes a string parameter if there is no param, put null  Method method = aClass.getMethod("method\_name", String.**class**);  Object returnValue = method.invoke(**null**, "parameter-value1"); |

So you can have an object, and get its class and methods at runtime. You can also try to see what actions you can perform at runtime without knowing which type of object this is. This is useful for example if the object you are using is an error packet and you have no idea what type of error it is. Instead of using instance of checks (which could be bad if you have 100’s of different types of error packets), you could simply get an array of the methods that this object has. You could then check to see if a particular method is in that array and if so, call it.