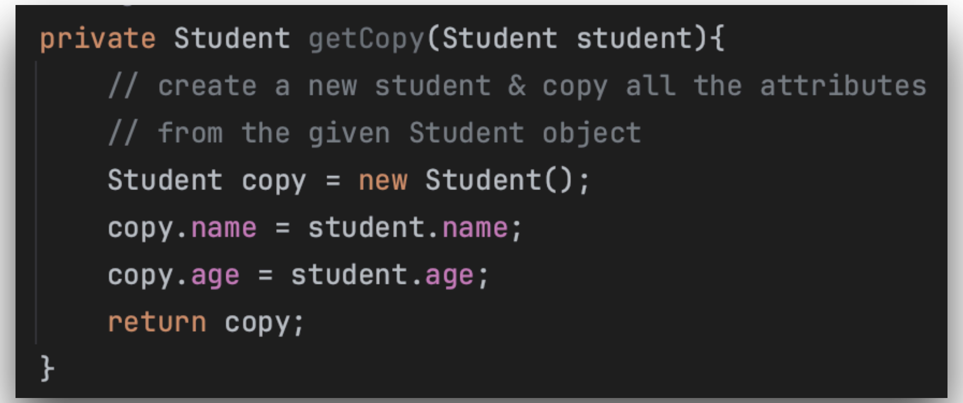
Prototype & Registry

Prototype Design Pattern

- Prototype design pattern helps us to solve problems where we need to create a copy of a given object.

Problem Statement

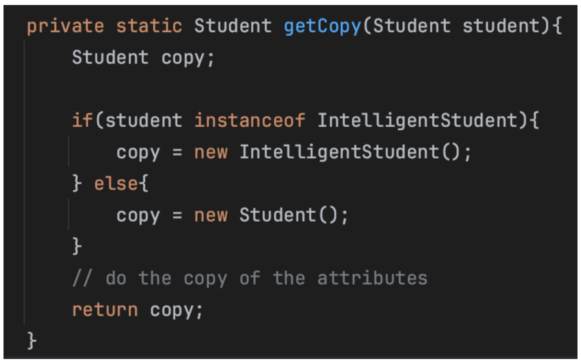
- Given an object of a Student class, we need to create an exact **copy** of that object
    - **Copy** -> Create a new object of the Student class. Then you have to go through all the fields of the original object and copy their values over to the new object in new memory location



```
private Student getCopy(Student student){
    // create a new student & copy all the attributes
    // from the given Student object
    Student copy = new Student();
    copy.name = student.name;
    copy.age = student.age;
    return copy;
}
```

Disadvantages

- **Tight Coupling** - Client will need to know all the attributes details of the given object to create the copy of the object
- If the given object has any private attributes, those can't be accessed by the above approach & hence we can't create a copy of **Student** object with all the attributes
- If the Student class has any subclasses for example, or if the given object is referred by an Interface & not the exact concrete type
    - **Student** (parent) —> **IntelligentStudent** (child class), In this case Student class reference can be used to refer child class objects.
    - So if we receive Student object, we have to do some additional work to find out the what is the actual instance



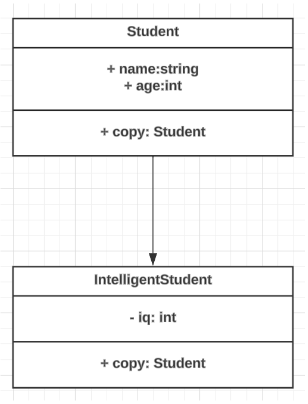
```java
private static Student getCopy(Student student){
    Student copy;

    if(student instanceof IntelligentStudent){
        copy = new IntelligentStudent();
    } else{
        copy = new Student();
    }
    // do the copy of the attributes
    return copy;
}
```

- In the future, if we have another sub class of **Student**, we have to update the **getCopy( )** method & add one more else if to handle the new subclass type. This breaks open closed principle.
- Its hard to create a copy of **Student** object when it has different subtypes

Prototype Design Pattern

- The Prototype pattern delegates the cloning process to the actual objects that are being cloned.
- The pattern declares a common interface for all objects that support cloning. This interface lets you clone an object without coupling your code to the class of that object. Usually, such an interface contains just a single clone method.
- If we want to create a copy of an object, have the logic to create the copy within the class itself.
- By this way, client class will not have tight coupling with the Student class & OCP will not be broken
- Because of the runtime polymorphism, even if the subclass objects are given to us with the Parent class or interface reference, the copy method will be called based on the object that it has.
- We can even copy private fields because most programming languages let objects access private fields of other objects that belong to the same class (If the method that accessing the private fields present inside the class)
- An object that supports cloning is called a prototype.



Pre-built prototypes can be an alternative to subclassing.

Here's how it works

- You create a set of objects, configured in various ways. When you need an object like the one you've configured, you just clone a prototype instead of constructing a new object from scratch.

Real Use Case

- We often face scenarios where we don't want to create the object from scratch, rather we prefer to create a copy from the template prototype & change the values.
- Search Queries is a real use case, where we don't want to create the **SearchAPI** object again & again, as most of the attribute values will be same in all the different **SearchAPI** objects. So create a **SearchAPI** object & for every subsequent request
- Get the copy of that object & update the necessary attributes.
- So in the case **SearchAPI**, store the **SearchAPI** prototype object in a registry associated with a key. (Key value Hashmap) and create the copy of the prototype.

Registry

- Registry is the place where we will store all the different types of prototypes.
- So Registry should be a Singleton. If not multiple registry objects will have the same prototype object created multiple times in each registry
- The main use case of registry is, if something (some prototypes) is needed again & again, store such things in the registry