**What are Design Patterns?**

* Think of design patterns like software templates. **They’re a programming design to base your code off.**
* They’re handy to know because they help us solve recurring problems that arise when developing applications.
  + Also, a lot of dependencies and modules we’ll use in our software careers will make use of design patterns for cleaner and more efficient functionality.
* Some popular design patterns we’ll use in training:
  + **Singleton**
  + **Factory**
  + DAO
  + Java layered architecture
  + Publisher-Subscriber

**Singletons**

* A singleton is a class that can have **only one instance at a time.**
  + Hence the name **single**ton
* What’s the point?
  + The “new” keyword creates a new object of a class
  + BUT sometimes, you don’t want multiple objects of a class
    - Maybe you want all the attributes of the Class to be consistent across all instances.
  + Using a singleton ensures that only one object can ever be instantiated. There will never be two+ different instances of the Class.
* How do you implement a singleton?
  + Arguably, the most common way is to have a **private constructor, and a static method** that creates an object of the class.
* You will probably never have to make your own singleton, but you’ll often find yourself using singletons in built-in Java libraries.

\*Made a singleton in the design pattern + logging demo\*

**Factories**

* A Factory is a class that lets us create objects without exposing the creation logic to the client (the messy code that creates objects is abstracted away). Great way to clean up your code.
* What’s the point?
  + Using the “new” keyword is fine if you’re only making a couple types of one object
  + What if we had some Cookie Interface and 500 different implementations for each type of Cookie? Would we have to write a huge chunk of code (maybe a switch statement?) using the “new” keyword for each possible type of Cookie implementation?
  + This problem can make your code super messy! But this is where factories come in
* How to implement
  + We’ll need an Interface that represents a generic type of Classes
    - E.g. A Cookie Interface implemented by different Cookie-Based Classes.
  + We simply move all the decision-making code into an entirely separate class ending in Factory. E.g. CookieFactory
  + In this way, you can abstract away all the ugly decision-making code, with a method that executes all of it.

\*Made a factory in the design pattern + logging demo\*

**SOLID Design Principles**

The SOLID design principles are a set of five principles aimed at making software designs more understandable, flexible, and maintainable. They were introduced by Robert C. Martin (Uncle Bob) and are widely used in object-oriented programming. They are:

* **Single Responsibility Principle (SRP)**
  + A class should have only one reason to change or be used. This means that a class should only have **one job or responsibility.**
* **Open/Closed Principle (OCP)**
  + Entities (classes, modules, functions, etc.) should be **open for extension but closed for modification.** This means that you should be able to add new functionality without changing the existing code.
    - Overruled by the “final” keyword in a Class.
* **Liskov Substitution Principle (LSP)**
  + **Subtypes must be substitutable for their base types.** This means that if a program is using a base class, it should be able to use any of its subclasses as a substitute. This is why we can say Dog d = new Hound();
    - (assuming Hound extends Dog)
* **Interface Segregation Principle (ISP)**
  + Clients should not be forced to depend on interfaces they do not use. This means that **a class should not have to implement methods it doesn't use.** Instead, you should break up your interfaces into smaller, more specific ones that fit specific needs.
    - We would want our concrete Dog class to implement a Dog Interface, as opposed to an Animal Interface. We want parent entities to be as child specific as possible.
* **Dependency Inversion Principle (DIP)**
  + **High-level modules should not depend on low-level modules.** Both should depend on abstractions. This means that you should depend on abstractions, not on concrete implementations. This leads to a decoupled system where high-level and low-level objects can be modified independently of each other.
    - Basically, if you’re writing a class that inherits from another entity, it should **inherit from something abstract,** like an Interface, instead of another concrete class.
      * Instead of DoubleChocolateChip extending ChocolateChip, we could have a ChocolateChip Interface that all subtypes implement.
        + This one isn’t SUPER closely followed at all times, even by the Java devs but it’s good to follow if possible. It even contradicts the Open/Close principle