Object-Oriented Design Principles

* **Interface**
  + Interfaces cannot contain instance variables. If you declare a data member in an interface, it should be initialized, and all such data members **are implicitly treated as “public static final” members**.
  + An interface **cannot declare static methods**. It can only declare instance methods.
  + You cannot declare members as protected *or* private. Only public access is allowed for members of an interface.
  + All methods declared in an interface are implicitly considered to be abstract. If you want, you can explicitly use the abstract qualifier for the method.
  + An interface can be declared with empty body (i.e., an interface without any members. Such interfaces are known as *tagging interfaces* (or *marker interfaces*).
  + Unlike top-level interfaces that can have only public or default access**, a nested interface** can be declared as public, protected, or private.
* **ABS vs Interface**
  + An abstract class can have both static and non-static constants.
  + Interfaces can have only static constants. If you declare a field, it must be initialized. All fields are implicitly considered to be declared as public static and final.
* **ABS or Interface**
  + If you are identifying a base class that abstracts common functionality from a set of related classes, you should use an abstract class. If you are providing common method(s) or protocol(s) that can be implemented even by unrelated classes, this is best done with an interface.
  + If you want to capture the similarities among the classes (even unrelated) without forcing a class relationship, you should use interfaces. On the other hand, if there exists an is-a relationship between the classes and the new entity, you should declare the new entity as an abstract class.
* **Object Composition** 
  + „has a“ releationship
  + Smaller objects that form together larger objects
  + Computer is composed of keyboard, CPU, memory etc. Computer shares a „has a“ relationship with the rest
* **Inheritance**
  + Implements „is a“ relationship
* **Composition vs. Inheritance**
  + Computer has a CPU
  + Circle is a shape
  + Cat is an animal
  + Circle has a point
  + A laptop is a computer
  + A vector is a list
  + Use inheritance when a subclass specifies a base class, so that you can exploit dynamic polymorphism. In other cases, use composition to get code that is easy to change and loosely coupled. In summary, **favor composition over inheritance**.

**Design patterns**

**Simple factory**

****

* This pattern creates and returns objects of classes that extend a common parent class

or implement a common interface.

* The objects are created without exposing the instantiation logic to the client
* The calling class is decoupled from knowing the exact name of the instantiated class.
* Because method getAppInstance() in class AppFactory is a static method, this pattern is also referred to as the **Static Factory pattern or Factory Class pattern.**

**Factory Method pattern**



* The intent of the Factory Method pattern is to define an interface for creating an object but let subclasses decide which class to instantiate. The Factory Method pattern lets a class defer instantiation to its subclasses.
* This arrangement promotes flexibility to change the App object returned from concrete factory classes (WordAppFactory and TextEditAppFactory)

**Abstract factory**



* The Abstract Factory pattern is used to create a family of related products (in contrast, the Factory Method pattern creates one type of object).
* This pattern also defines an interface for creating objects, but it lets subclasses decide which class to instantiate.
* This arrangement also promotes flexibility to change the App or Font object returned from Concrete Factory pattern classes (WordAppFactory and TextEdit- AppFactory).

**Benefits of factory design**

* Prefer method invocation over direct constructor calls
* Prevent tight coupling between a class implementation and your application
* Promote creation of cohesive classes
* Promote programming to an interface
* Promote flexibility. Object instantiation logic can be changed without affecting the clients that use objects. They also allow addition of new concrete classes.
* **Java Class Design**

If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the super class does not have a no-argument constructor, you will get a compile-time error. Object *does* have such a constructor, so if Object is the only superclass, there is no problem.

If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that there will be a whole chain of constructors called, all the way back to the constructor of Object. In fact, this is the case. It is called ***constructor chaining*,** and you need to be aware of it when there is a long line of class descent.