Creational design patterns: Dependency Injection, Factory Method, Builder, Singleton
Construction versus Use

- When designing a system, we distinguish between construction and use.
  - Construction = Creating an instance of some concrete class.
  - Use = Calling methods on an object
    - Do not need to know the object’s concrete class,
    - Only need to know that it implements certain interface(s).

- The distinction between construction and use (aka maintenance) happens in construction engineering as well:
  - Construction site is very different than a building full of tenants.
  - Different needs & priorities, different rules, different services needed, etc.
Construction vs. Use

- Generally good design (especially for large systems):
  - Most of the application is written in terms of interfaces.
  - Introduce objects whose sole responsibility is constructing other objects.
    - These are the only objects that depend on concrete implementations.
  - Make the *boundary* between construction and use clear.
    - Ensure it is easy to swap in different implementations
Dependency Injection

● *Dependency Injection* is a simple pattern/technique:
  ○ A class should avoid instantiating its own *dependencies*
    ■ By using the `new` keyword, or by calling some static function.
  ○ Instead let someone else *inject* these dependencies
    ■ By passing them as constructor arguments, or using setter methods.
  ○ Important: Dependencies are declared using *interfaces*
    ■ NOT using concrete implementations

● Who is that “someone else” that injects dependencies?
  ○ Could be a main method
  ○ Or something more elaborate like a dependency injection container (e.g. [Spring](http://spring.io), [Guice](http://guicegooglesource.google))
Dependency Injection

• In high-level, a dependency Injection container is responsible for:
  ○ Mapping between interfaces and concrete implementations
  ○ Instantiate the concrete implementations
• In practice, DI containers can become fairly complex (and hide a lot of the complexity involved of large-scale object-oriented applications).
• Here is a short video explaining dependency injection

Warning: If you’re using dependency injection for a small project, there is a good chance you are over-engineering your application.
Designing Interfaces

- Why is it good to design your application in terms of interfaces?
  - Allows you to focus on the domain problem, and ignore implementation details
    - Which also results in code that is cleaner and easier to read
  - Can use different implementations
    - Gradual improvements
    - Support different integrations
      - Ex: Different sign-in providers such as Google, Facebook, GitHub, etc.
  - Easier to test (we can inject test doubles for testing purposes)
Stating The Problem

● **Problem:** In some cases, your application needs to instantiate dependencies during runtime, but you still want to avoid depending on concrete classes.

● In other words ... 
  An application needs to create instances of type T.
  ○ T is an interface (or an abstract class)
  ○ The application *must not* depend on any specific implementation of T. (i.e. it must NOT construct objects of type T by itself)
  ○ The application depends only on T, and work with any implementation of T.
Why Is It a Problem?

- What’s wrong with depending on a specific implementation?
  - The specific implementation might not exist,
  - you might want to replace it in the future,
  - It might not be the one you want to use for testing,
  - When the specific implementation changes, need to recompile the whole application.

- In other words ... Lack of flexibility.
Solution

- Intuition: Application delegates the responsibility (of creating objects of type T) to someone else.
  - Let “someone else” invoke constructors (of various implementations of T).
  - Depend on that “someone else”, instead of any specific implementation of T.
  - For flexibility, the “someone else” should be an interface (and not a concrete class).
The *Factory Method* design pattern is one way to implement our intuition.

Let's see some diagrams describing the *Factory Method* pattern ...
Step 1: The application invokes the constructors of different implementations of T.
Step 2: Application depends on factory objects to construct instances of type T. Factory objects invoke the constructors of different implementations of T.
Step 3: Application depends only on the factory interface. Any factory implementation can be used to constructs instances of type T.
We can simplify the codebase using functional interfaces.

Instead of defining the `Factory` interface (which has a single method that takes no arguments and returns an object of type T), use `Supplier<T>`.
Or, if you prefer UML ...

```
Creator

# factoryMethod() : Product
+ anOperation()

Product

ConcreteCreator

+ factoryMethod() : Product

ConcreteProduct
```

«use»

«create»
The *Factory Method* Design Pattern

- Delegate responsibility
  - Don’t construct objects of type T yourself!
  - Ask a factory object to do it for you.

- Make the factory an interface
  - With a single method that returns T.
  - Different implementations of the factory interface return different implementations of T.
Factory Method

Let’s see a code example ...
Abstract Factory

● Related design pattern: Abstract Factory
  ○ Very Similar to Factory Method, except that the factory interface defines multiple methods.

● Instead of creating just one (type of) object, we define an interface for creating “families of related or dependent objects”.
  ○ Can be implemented using Factory Methods (but there are other options as well).
Factory Method vs. Abstract Factory

- Difference in implementation - Inheritance vs. Composition
  - When implementing a Factory Method interface, the implementing class is a factory. (inheritance)
  - When implementing an Abstract Factory interface, the implementing class has a factory. (composition)
Let’s see another creational design pattern that solves a slightly different problem ...
Telescoping Constructor

- Telescoping Constructor
  - A constructor with many arguments
  - Possibly of the same type

- Ex:
  
  ```java
  new CanadianAddress("Apt 1A", "301", "College", "St.", "Toronto", "Ontario", "B4D C4T");
  ```
Telescoping Constructor

● Error prone
  ○ Correctness might depend on argument order
    ■ Developers might get it wrong
    (Or, waste time double checking)
    ■ Compiler cannot save you

● Harder to test
  ○ More arguments ⇒ Even more combinations to test

● Results in ugly code
The Builder Pattern

- Break construction into two separate tasks:
  - Collecting arguments
  - Creating the instance

- Solve the telescopic constructor problem by collecting the arguments one at a time

- Improve code quality
Let’s see a code example ...
Builder, Argument Validation

- Perform simple validation when collecting arguments
  - Null, empty string, negative number, etc.

- Perform (semantic) validation in `build()`
  - Conditions that involve multiple arguments
  - Ex: Conflicting postal code and province
Singleton

- Another creational design pattern
  - Ensures that only one instance of a certain class can be created

- Ex:

```java
public class Foo {
    private static Foo instance = null;

    public static Foo getInstance(){
        if(instance == null){
            instance = new Foo();
        }
        return instance;
    }
}
```

*Note: Implementation is slightly more complex for multi-threaded programs.*
Singleton

- Controversial design-pattern
  - Feels like a global
  - Considered an anti-pattern by some people

- Think twice before using it
  - There are cases where singleton is the right solution, but they are fairly rare