Brandon Corbett, Zackery Weeks

Dr. David Gibson

CS 4322

20 March 2018

Dependency Injection

Dependency injection is a technique in software engineering that is used to decouple objects from what they depend on. Many times, in object-oriented programming, a certain class will depend on another object in order to function correctly. An example of this could be as follows:

public class Car {

//dependency

ToyotaDoor td = new ToyotaDoor(“red”);

}

In this example, a Car object exists with a dependency inside of it. The dependency in this case is ToyotaDoor which is passed a value of String “red”. The Car object is said to be dependent on the ToyotaDoor object because the ToyotaDoor object is instantiated inside of the Car object and relies on some methods or variables in the ToyotaDoor object in order to function correctly.

This is not a good object-oriented design practice. The goal for good object-oriented designs is high cohesion and low coupling. This means that classes should have methods focused on doing one particular job well rather than many different methods for many different jobs in an unfocused way, and classes should also be as independent from each other as possible.

The high coupling in the previous example makes it difficult to change and maintain code. The Car and ToyotaDoor classes are tightly knit together which means that any change that might need to take place to either class could require a large amount of refactoring, code redesign, and revamping in order to make it work correctly.

Dependency injection is a design technique geared towards injecting dependencies, or, pushing a dependency into a class from the outside. Doing this separates the classes from each other, decreasing coupling, and moves away from using the “new” operator inside of a class. This is accomplished by using the Dependency Inversion Principal. The Dependency Inversion Principal states as follows:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

In other words, objects should depend on abstractions rather than instantiations which decouples the class’s implementations from each other.

This concept will become clearer while taking a look at an example similar to the Car example above.

**public** **class** Car {

//dependency

Ford fordCar = **new** Ford("This is a Ford Mustang car.");

**public** String viewCars() {

System.***out***.println(fordCar.getTypeCar());

**return** fordCar.getTypeCar();

}

}

**public** **class** Ford {

**private** String typeCar;

**public** Ford() {

}

**public** Ford(String typeCar) {

**this**.typeCar = typeCar;

}

**public** String getTypeCar() {

**return** **this**.typeCar;

}

**public** **void** setTypeCar(String car) {

**this**.typeCar = car;

}

}

The code is similar to the first example above. A fordCar object is instantiated in the Car class and passes a string to the Ford class constructor while the viewCars() method calls the getTypeCar() method in the Ford object, prints it, and returns it. The Car class, however, still contains the same dependency coupling. fordCar is instantiated within the Car class and no dependency injection has occurred. It is important to see this coupling so that an engineer can see the problem and determine a course of action to decouple the dependency.

If an engineer goes to the main method in the program and inputs:

**public** **class** main {

**public** **static** **void** main(String[] args) {

Car car1 = **new** Car();

car1.viewCars();

}

}

The end result will be the printing out of “This is a Ford Mustang car.” in the console. There is seemingly no problem here. But, what if an engineer wants to create three cars in the main method? It will not be possible to create three unique cars because of the dependency that exists in the Car class, and each instance of a new car will all print out “This is a Ford Mustang car.”

One solution to this predicament is to create a constructor-based dependency injection. This is done by passing a parameter to the Car constructor rather than instantiating the Ford object within the class.

**public** **class** Car {

//dependency

//Ford fordCar = new Ford("This is a Ford Mustang car.");

Ford fordCar; // removed the "new" operator

**public** Car(Ford fordCar) {

**this**.fordCar = fordCar;

}

**public** String viewCars() {

System.***out***.println(fordCar.getTypeCar());

**return** fordCar.getTypeCar();

}

}

**public** **class** main {

**public** **static** **void** main(String[] args) {

Ford f1 = **new** Ford("This is a Ford Mustang car.");

Ford f2 = **new** Ford("This is a Ford Focus car.");

Ford f3 = **new** Ford("This is a Ford Fiesta car.");

Car car1 = **new** Car(f1);

Car car2 = **new** Car(f2);

Car car3 = **new** Car(f3);

car1.viewCars();

car2.viewCars();

car3.viewCars();

}

}

In this example, the “new” operator has been removed from the fordCar object allowing the Car constructor to set the fordCar instance from outside of the Car class rather than hardcoding the fordCar object inside of the Car class. This decouples the dependency from the Car class and allows for multiple different instances of a Car to be instantiated in the main method.

A similar way to accomplish this same decoupling is using method-based dependency injection. A method can be used to set the instance in a class rather than the constructor.

**public** **class** Suv {

Ford fordCar;

**public** Suv() {

}

**public** **void** setCar(Ford fordCar) {

**this**.fordCar = fordCar;

}

**public** String viewCars() {

System.***out***.println(fordCar.getTypeCar());

**return** fordCar.getTypeCar();

}

}

**public** **class** main {

**public** **static** **void** main(String[] args) {

...

Ford f4 = **new** Ford("This is a Ford Explorer.");

Suv suv1 = **new** Suv();

suv1.setCar(f4);

suv1.viewCars();

}

}

In this example, a method can be used to set the instance within in the Car class rather than the constructor. This is also an effective way to use dependency injection in order to decouple dependencies between classes.

A third way to accomplish decoupling of classes is through interface-based dependency injection.

**public** **interface** TruckSetter {

**public** **void** setTruck(Ford fordTruck);

}

**public** **class** Truck **implements** TruckSetter {

**private** Ford fordTruck;

**public** Truck() {

}

@Override

**public** **void** setTruck(Ford fordTruck) {

**this**.fordTruck = fordTruck;

}

**public** String viewCars() {

System.***out***.println(fordTruck.getTypeCar());

**return** fordTruck.getTypeCar();

}

}

**public** **class** main {

**public** **static** **void** main(String[] args) {

...

Ford f5 = **new** Ford("This is a Ford F-150 truck.");

Ford f6 = **new** Ford("This is a Ford F-250 diesel truck.");

Truck truck1 = **new** Truck();

Truck truck2 = **new** Truck();

truck1.setTruck(f5);

truck2.setTruck(f6);

truck1.viewCars();

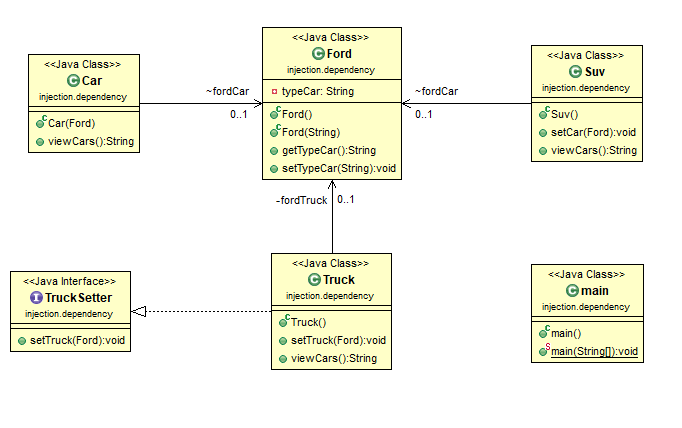
truck2.viewCars();

}

}

In this example, an interface is used in order to inject the dependency in to the Truck class and keep the Truck object and Ford object decoupled. The Truck class contains a setter method that overrides the interface method and is a good example of a well-known object-oriented design principle of programming to an interface and not an implementation.

An example class diagram can be seen below in order to get a better understanding of how these classes work together.



“Dependency Injection.” *Wikipedia,* Wikimedia Foundation, 14 Mar. 2018,

en.wikipedia.org/wiki/Dependency\_injection.

Vogel, Lars. “Quick Links.” *Vogella.com,*

[www.vogella.com/tutorials/DependencyInjection/article.html](http://www.vogella.com/tutorials/DependencyInjection/article.html).

“Dependency Inversion Principle.” *Wikipedia,* Wikimedia Foundation, 25 Feb. 2018,

en.wikipedia.org/wiki/Dependency\_inversion\_principle.

ircmaxell. “Dependency Injection.” *YouTube,* Youtube, 9 Jan. 2013,

[www.youtube.com/watch?time\_continue=6&v=IKD2-MAkXyQ](http://www.youtube.com/watch?time_continue=6&v=IKD2-MAkXyQ).

Singh, Rahul Rajat. “Program to Interface, Not Implementation – Beginner’s Tutorial for

Understanding Interface, Abstract Class and Concrete Class.” *Program to Interface,*

*Not Implementation – Beginner’s Tutorial for Understanding Interface, Abstract Class*

*And Concrete Class – CodeProject,*

[www.codeproject.com/Articles/702246/Program-to-Interface-not-Implementation-Beginners](http://www.codeproject.com/Articles/702246/Program-to-Interface-not-Implementation-Beginners).