Unit 1 – 3
Dependency Injection
&
Inversion of Control

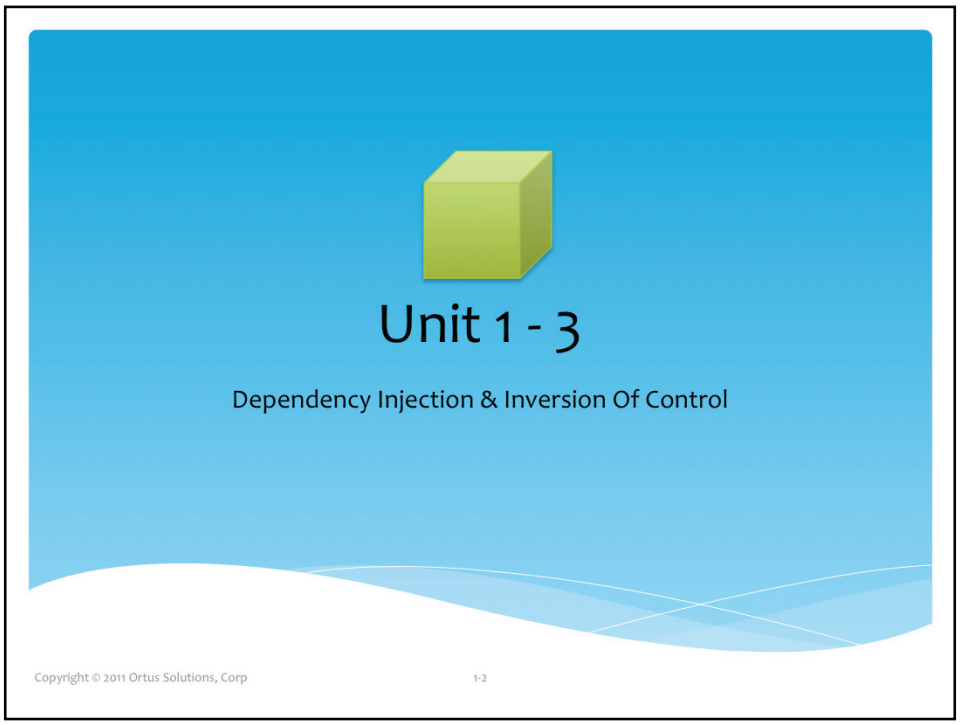
This is a free chapter from our CBOX202: WireBox Dependency Injection course (www.coldbox.org/courses/cbox202) and is freely donated to the ColdFusion community by Ortus Solutions, Corp (www.ortussolutions.com), creator and maintainer of ColdBox, WireBox, LogBox, CacheBox, MockBox, ContentBox or Anything Box ☺

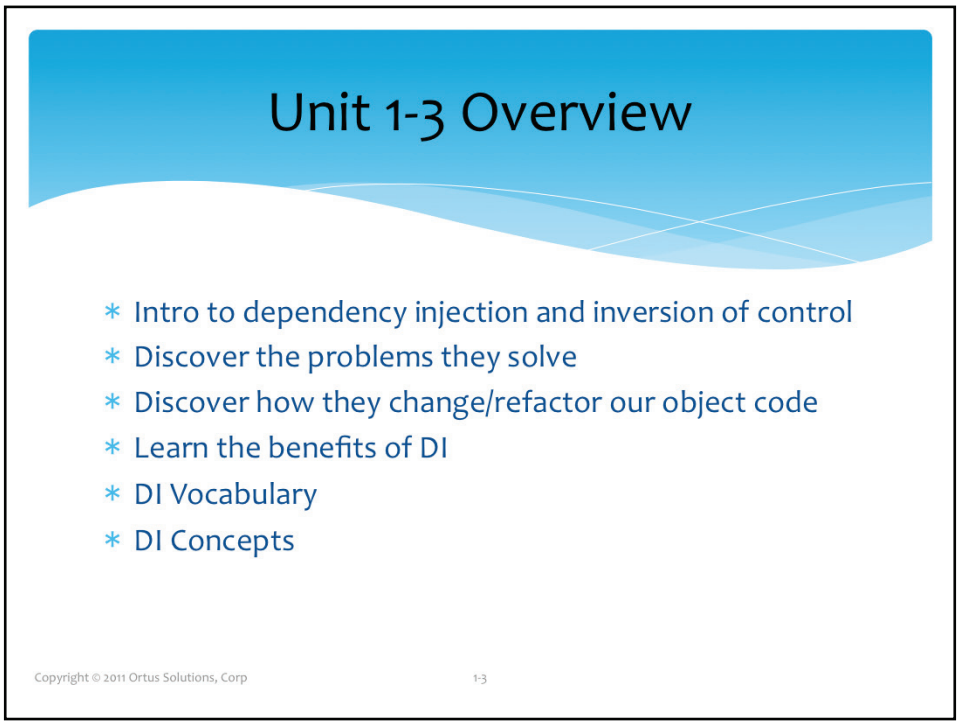
We believe dependency injection is a necessary software approach to building object oriented applications in ColdFusion that will bring tremendous benefits to your applications. We hope that you enjoy this chapter: *An introduction to dependency injection and inversion of control* and also that you try out WireBox for your dependency injection and AOP needs.

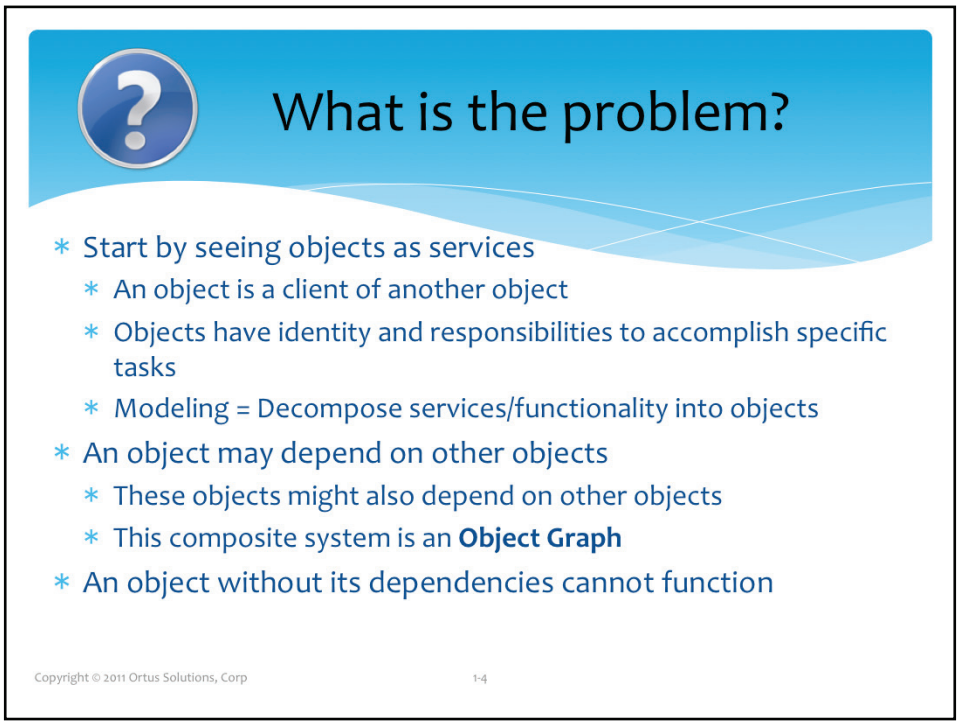
WireBox Links:

- http://coldbox.org/download#wirebox
- http://wiki.coldbox.org/wiki/WireBox.cfm
- http://wiki.coldbox.org/wiki/WireBox-AOP.cfm

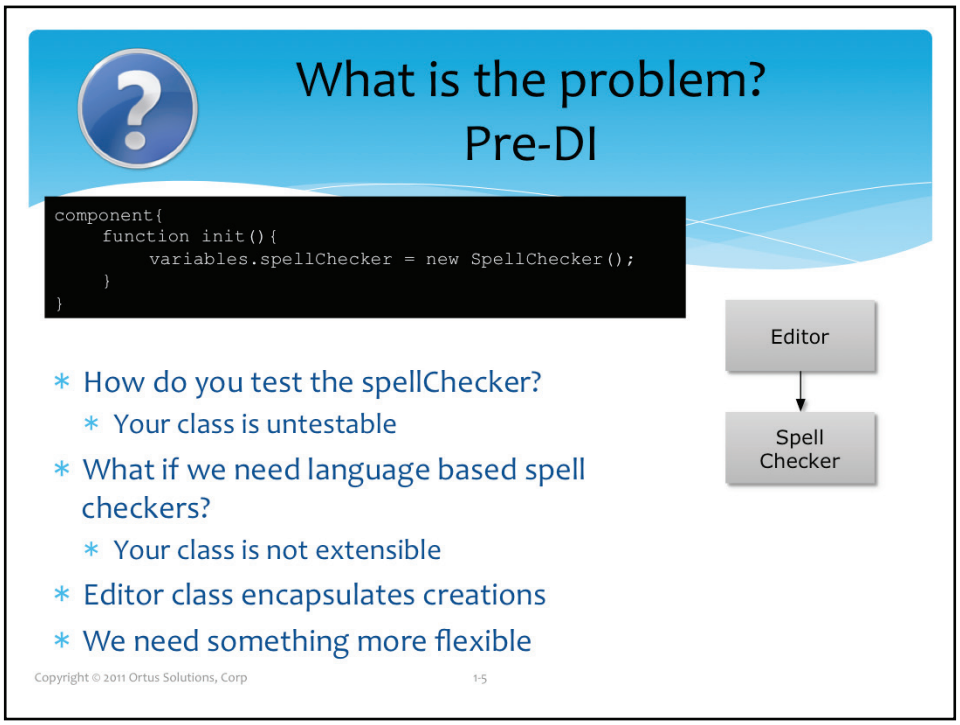
Luis Majano
CEO


Unit 1 - 3
Dependency Injection & Inversion Of Control
Copyright © 2011 Ortus Solutions, Corp
1-2


Unit 1-3 Overview
* Intro to dependency injection and inversion of control
* Discover the problems they solve
* Discover how they change/refactor our object code
* Learn the benefits of DI
* DI Vocabulary
* DI Concepts
Copyright © 2011 Ortus Solutions, Corp
1-3

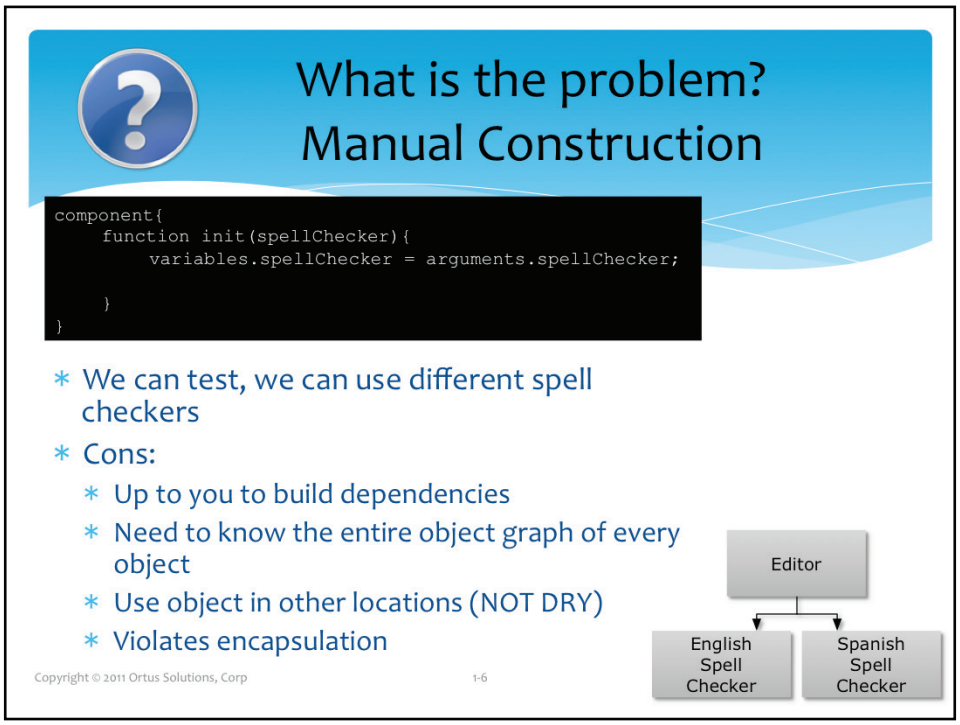


What is the problem?
We should start by seeing objects as services. When we start doing domain modeling we identify services and functionality and start assigning them to our objects. Our objects (thinking about cohesion and ontology) should have identity and clear responsibilities that will accomplish their specific tasks. In this mélange of functionality we will suddenly realize that objects depend on other objects and these objects might also depend on other objects. This chain of dependency is what composes an object graph. This object graph functions as a single unit for the target object and in turn this target object cannot function or provide services unless it can rely on its dependencies.

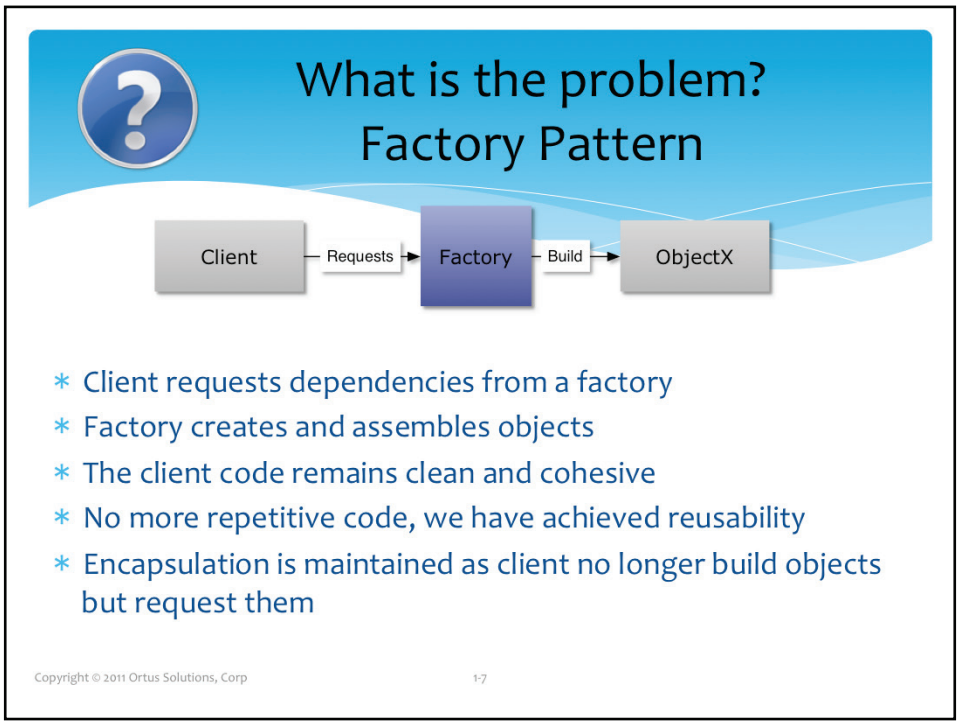


Pre-DI Solution or Problem?

To understand the elegance of dependency injection we must go through the steps of seeing the actual problem it solves. In our first attempt at creating an Editor object we do what we have always done, create its dependency in our constructor (or maybe another method). This object (**SpellChecker**) is a dependency or a collaborator of our Editor object. The Editor object relies on the **SpellChecker** in order to work. By doing this approach we can make the Editor work but how do we test the behavior that deals with spell checking? We can't! Our class is untestable, there is no way for us to replace or mock the spell checker as the Editor class encapsulates its creation. This might be sufficient for some use cases, but in all reality untestable classes are not good and un-flexible classes are not good either. We need something more flexible.



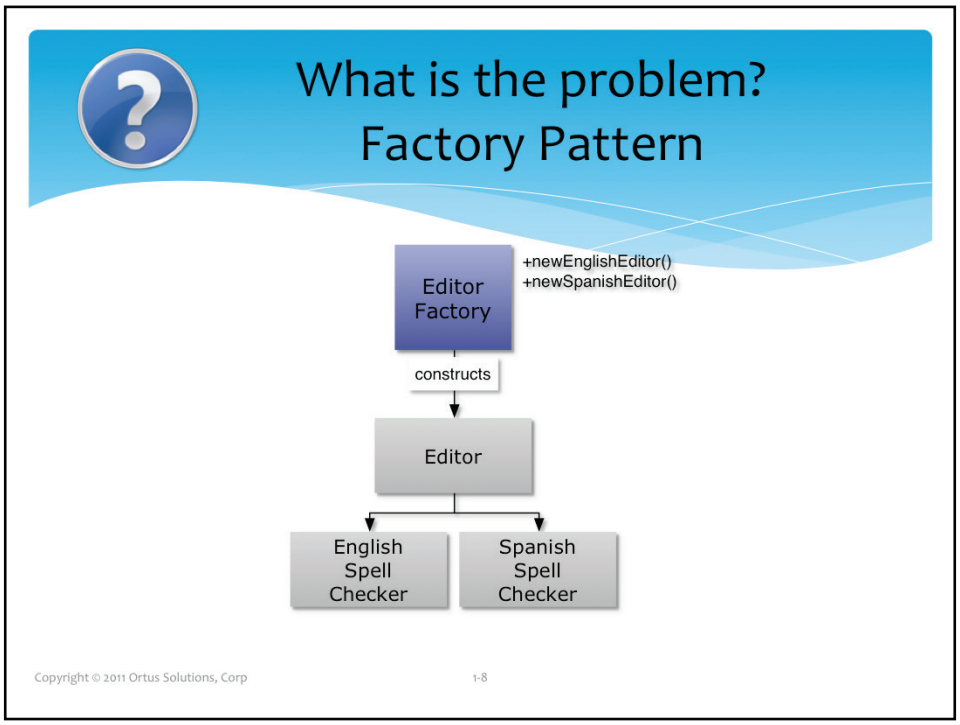
Manual Creation Solution or Problem?
Our first solution might be tweaking our code to accept the **SpellChecker** class in our constructor or via a setter method. BOOYA! We now can test and mock this **SpellChecker** class, but it gets better. We can now even change the behavior of the Editor by initializing it with English or Spanish spell checkers. We can definitely see improvement, flexibility and testability. However, we are completely breaking encapsulation because once we create an Editor object we must also create all of its required dependencies and so forth. This knowledge of the internals of every single object breaks encapsulation. Not only that, what if we need the Editor in another part of our code? Do we re-create all of the boilerplate code to create it? Do we rely on copy-paste techniques? You might, but I am pretty sure you are just introducing more maintenance issues and unreliability to your software. There has to be something cleaner and more reusable.

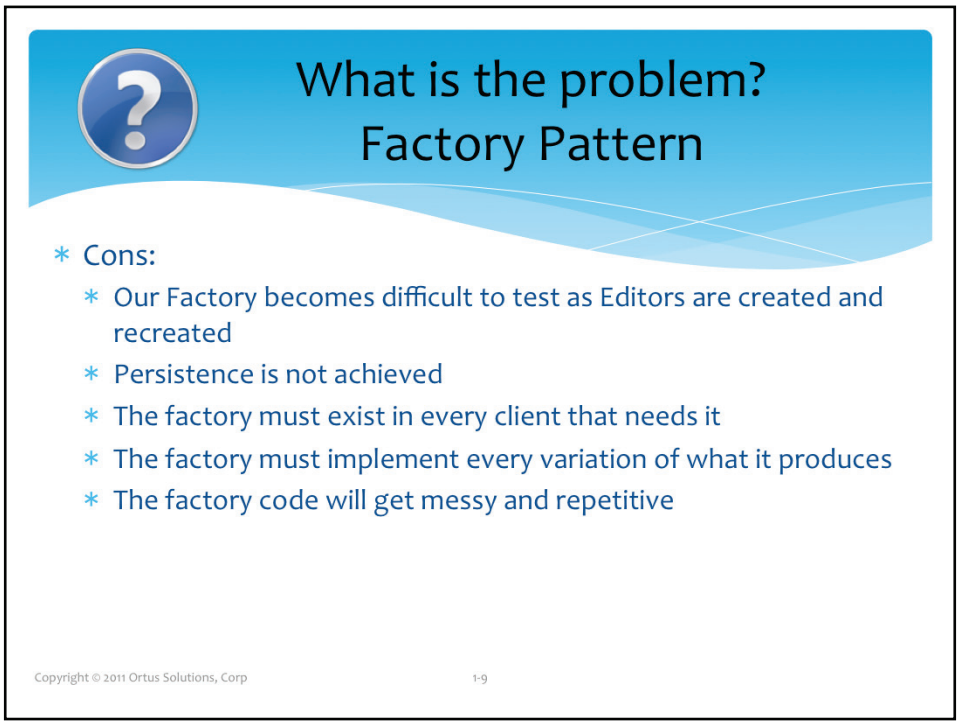


Factory Pattern
Our next approach in our silver bullet journey is the Abstract Factory Pattern.

"Provides a way to encapsulate a group of individual factories that have a common theme. In normal usage, the client software creates a concrete implementation of the abstract factory and then uses the generic interfaces to create the concrete objects that are part of the theme. The client does not know (or care) which concrete objects it gets from each of these internal factories since it uses only the generic interfaces of their products. This pattern separates the details of implementation of a set of objects from their general usage." – Wikipedia

With the factory approach clients request objects from the factory instead of creating them manually. Encapsulation is now preserved. The factory is now in charge of creating and assembling these objects for clients. The client code will now also remain cleaner and more cohesive. This allows for the reuse of the Editor class in multiple locations with minimal effort.

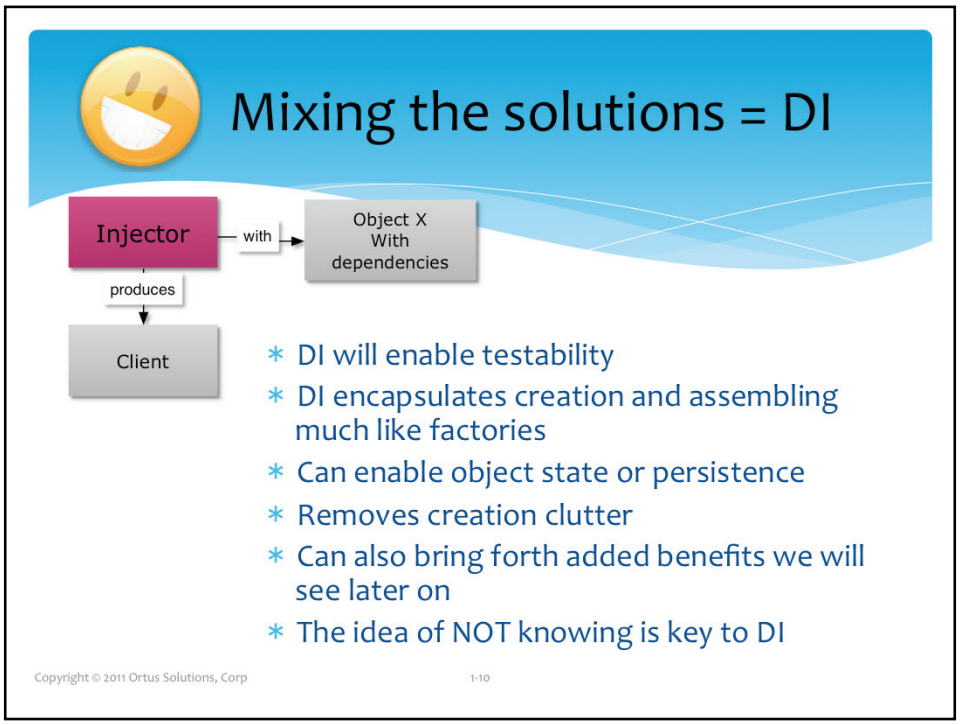

What is the problem?
Factory Pattern
Editor Factory
+newEnglishEditor()
+newSpanishEditor()
constructs
Editor
English Spell Checker
Spanish Spell Checker
Copyright © 2011 Ortus Solutions, Corp
1-8



Even though our factory pattern resolves many of our previous issues, it is still plagued with some serious drawbacks that must be address. If not addressed, the we must start making time from now for all the maintenance code we will have to write and refactor and write and test and refactor and write and test. Why? Well, let's begin.

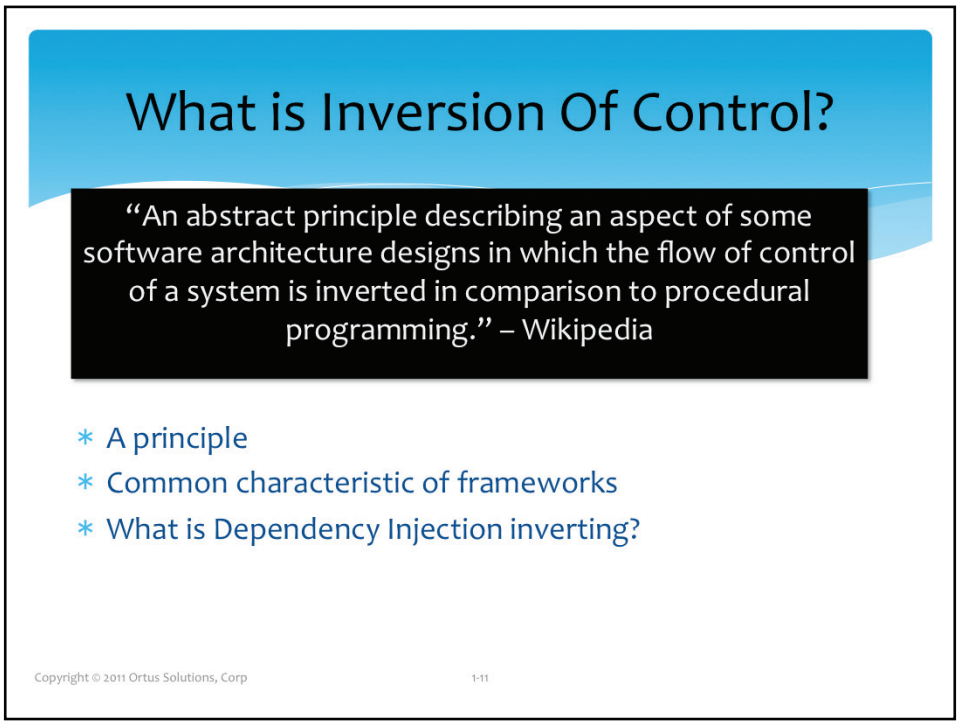
- Every time we request for a different type of Editor our factory will have to go to the same motions of creating and wiring the objects together in every unique method.
- If a change is made to the Editor, we must address it in every single method that creates editors
- We are now having repetitive code, unclean code and serious DRY issues
- How do we manage state for the Editors? If we create them once and store them locally, how can we test each one of them

As you can now see, our Factory pattern takes us only a certain distance ahead but we will always hit that inevitable roadblock. Therefore, our solution relies on what we will call dependency injection. With DI we take a completely different approach that stresses testability and cohesive code that will be easy to read and maintain. Ahh, how we like those words, EASY!

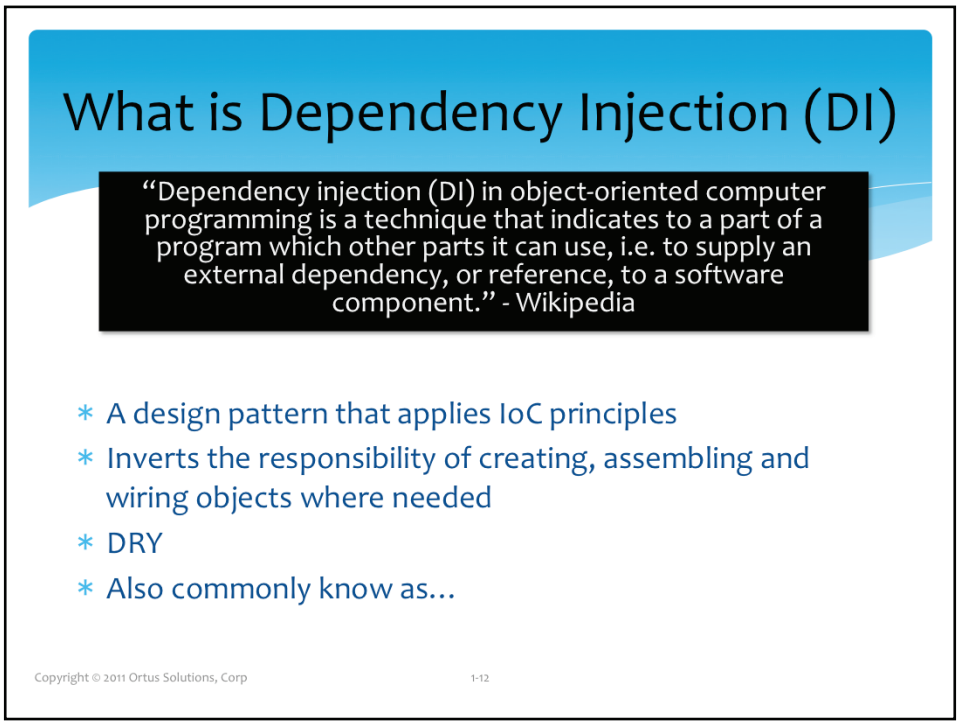


Our Silver Bullet: DI

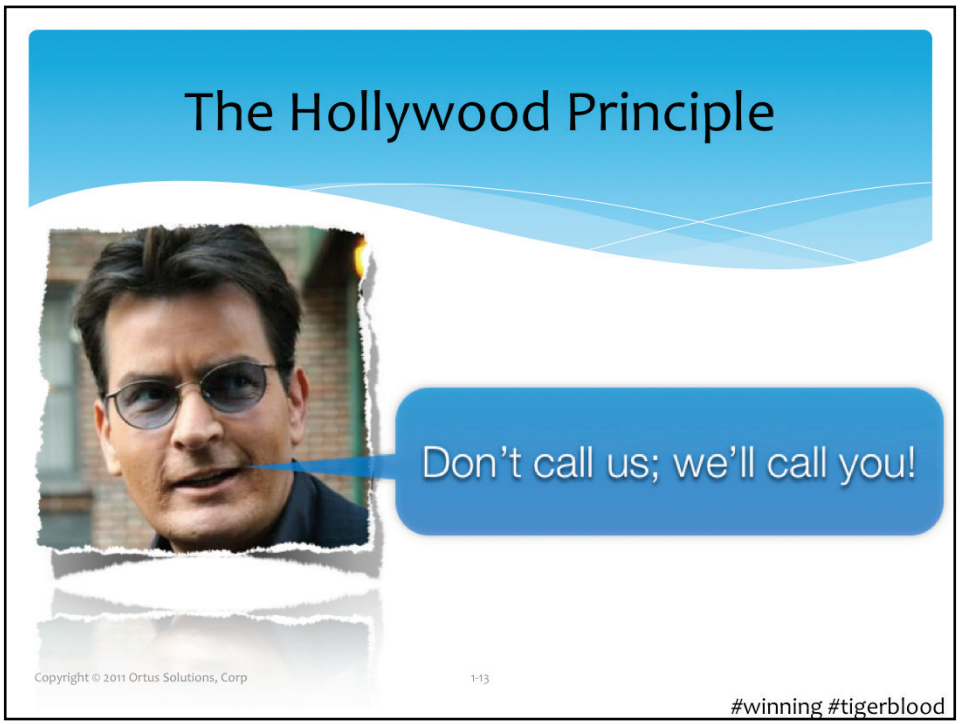
Dependency Injection or DI for short is a mixture of all the solutions we have seen (except service locator pattern, please Google if intrigued). DI will concentrate behavior on testability of objects by having usage of constructor arguments, setter methods, and even *cfproperty* metadata (more to come soon). DI can also be considered to be a big object factory or service locator and thus encapsulation of object creation and assembling is achieved. Thus, client code now remains cleaner with no more direct references to any concrete or abstract factories. This idea of not knowing where things come from is the cornerstone of dependency injection. So let's start getting messy!



Inversion of Control is more of a principle than an action that applies to dependency injection. Dependency injection is actually a practical use of Inversion of Control principles.

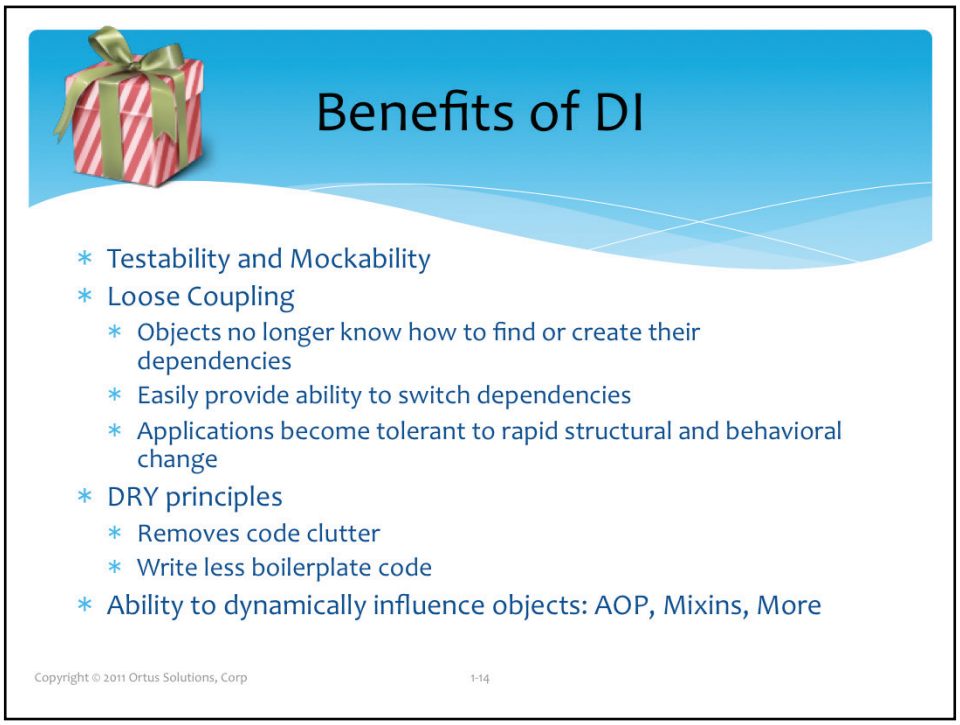


As we saw in our previous slide, dependency injection is an application of the inversion of control principles. The task of creating, wiring and assembling dependencies into objects are performed by an external framework or library commonly know as a dependency injection framework, commonly know as the DI acronym.



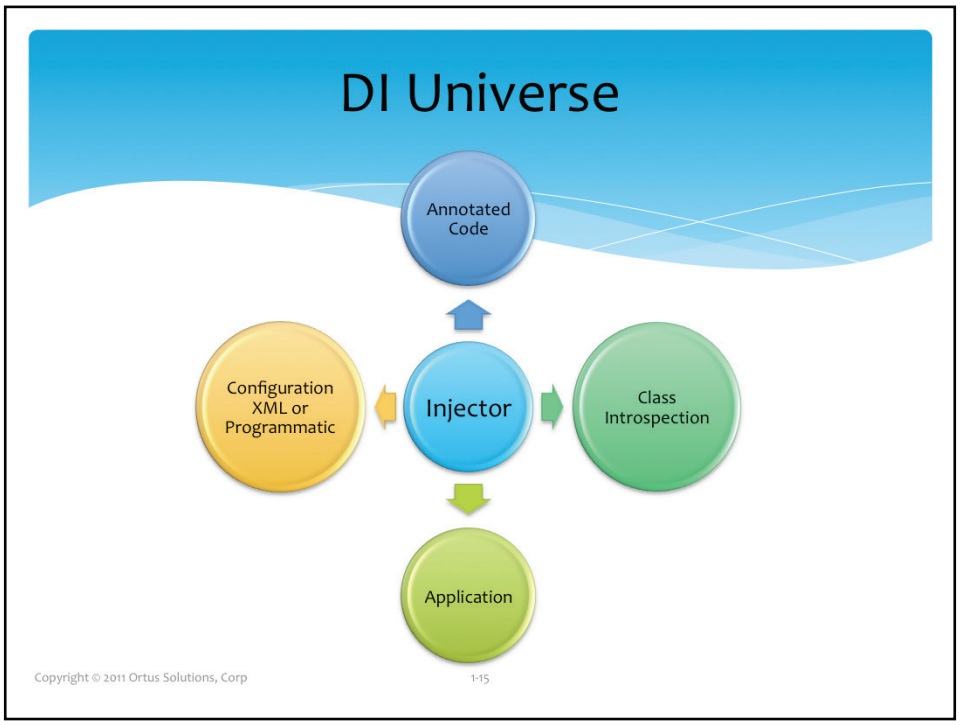
This principle is applied by DI libraries or frameworks in order to provide objects with the necessary dependencies it needs without the target object actually building or assembling these dependencies.

"The task of creating, assembling, and wiring the dependencies into an object graph is performed by an external framework known as a dependency injection framework or simple a dependency injector" – Dhanji Prasanna from Dependency Injection



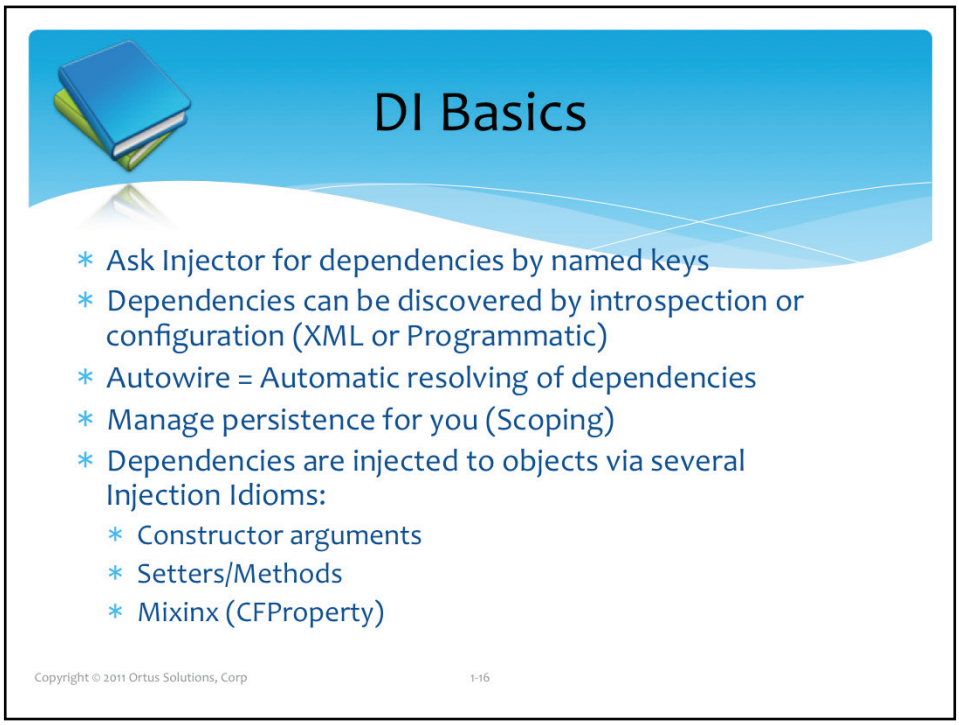
Advantages of a DI Framework
Compared to manual DI, using WireBox can lead to the following advantages:

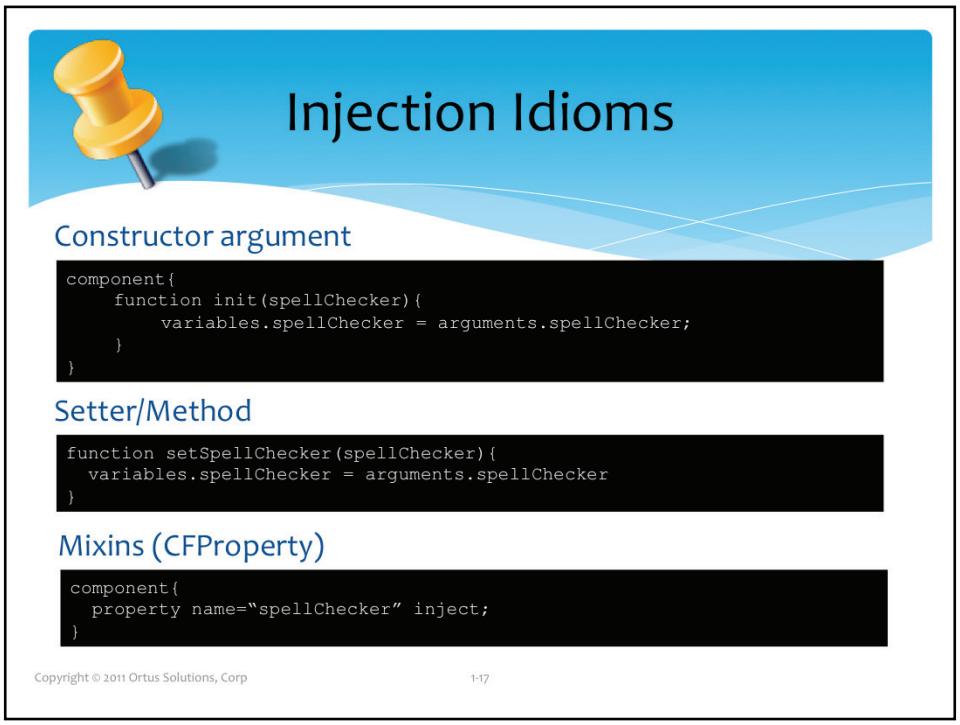
- Objects will become more testable and easier to mock, which in turn can accelerate your development by using a TDD (Test Drive Development) approach
- Loose coupling
    - Objects will become less dependent on each other as they now invert control to the DI library for object creation and resolution
    - Your applications will become more tolerant to rapid structural and behavioral changes as simple DI library changes can completely change behavior and object structures
- DRY Principles
    - You will write less boilerplate code.
    - You will stop creating objects manually or using custom object factories.
- Once WireBox leverages your objects you can take advantage of AOP or other event life cycle processes to really get funky with OO.



Most DI Injectors will have certain features and abilities. But pretty much all of them will have some way of detecting dependencies on target objects or will need to be provided with some type of configuration data in order to know the dependencies an object needs. This configuration approach has been done in XML or programmatically in some of the major DI libraries. As for detecting dependencies, many frameworks rely on class introspection and usage of annotated code or the addition of metadata to code at specific junctions that a DI library has agreed upon beforehand. Each approach has its benefits and drawbacks that we will investigate a more detailed example when dealing with WireBox.

A usual requesting of objects from an application or client code to the injector most likely is done with some type of identifier; much like the service locator pattern. Many DI libraries leverage simple keys or combinatorial keys with different approaches.

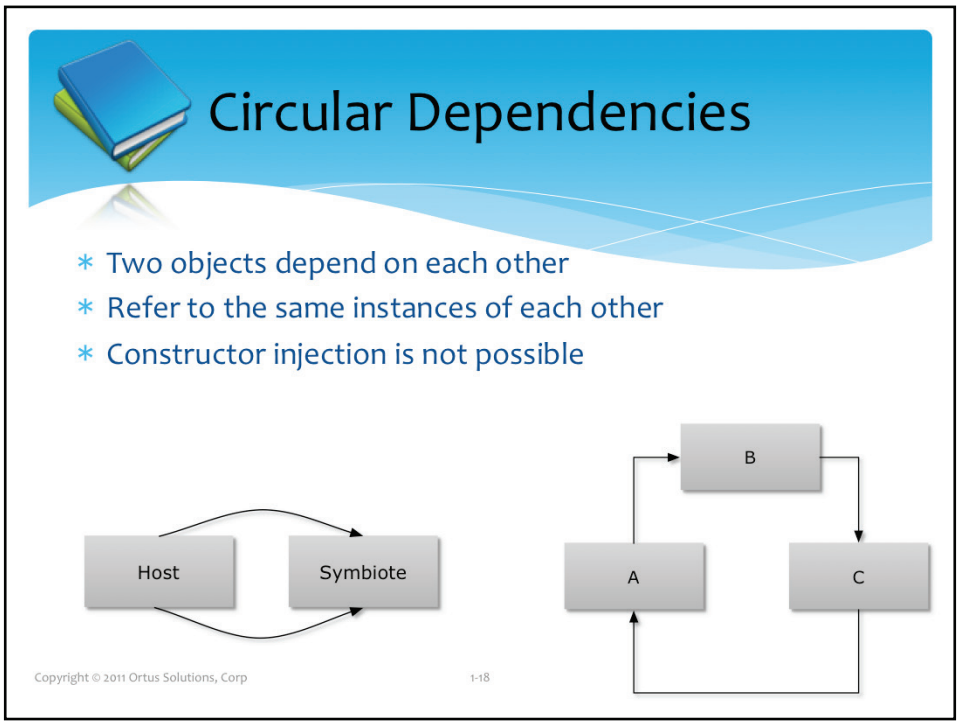

DI Basics
* Ask Injector for dependencies by named keys
* Dependencies can be discovered by introspection or configuration (XML or Programmatic)
* Autowire = Automatic resolving of dependencies
* Manage persistence for you (Scoping)
* Dependencies are injected to objects via several Injection Idioms:
* Constructor arguments
* Setters/Methods
* Mixinx (CFProperty)
Copyright © 2011 Ortus Solutions, Corp
1-16



Almost every DI library or framework supports different injection idioms. Some of the most commonly know injection idioms are:

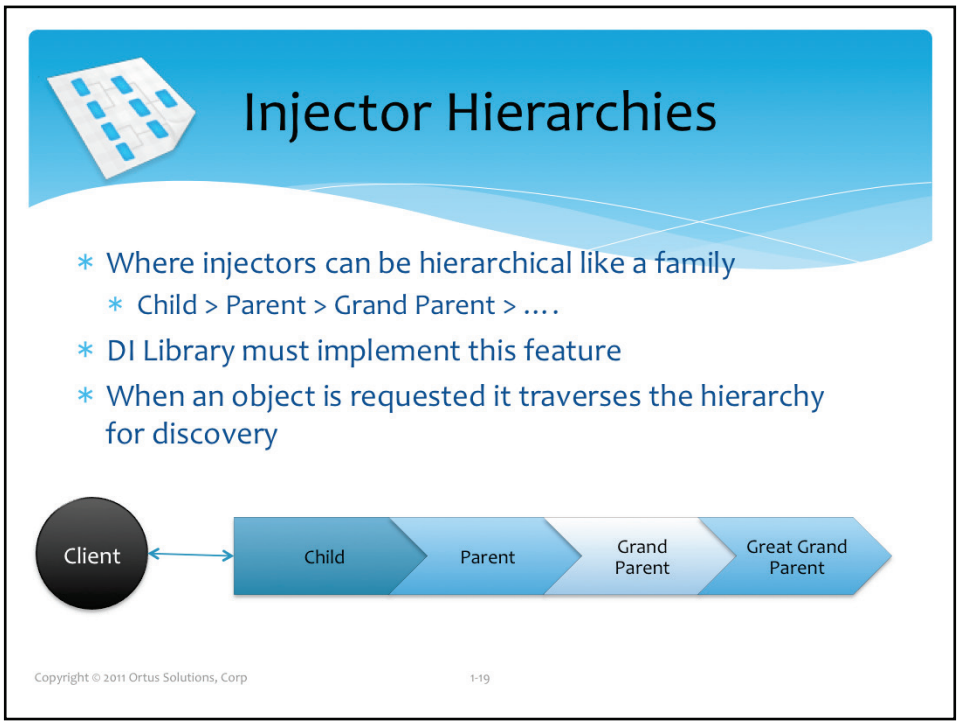
- **Constructor Arguments** : Where dependencies are passed via the constructor method
- **Setter/Method injection**: Where dependencies are passed via agreed method signatures such as setXXX where XXX is the name of the object to inject
- **Mixin Injection**: Where in popular dynamic languages you have the ability to modify classes at runtime by injecting dependencies in certain visibility scopes of a target

Each comes with its set of pros and cons that we will cover in later sections. The important aspect of any DI library is the HOW are dependencies injected into objects.



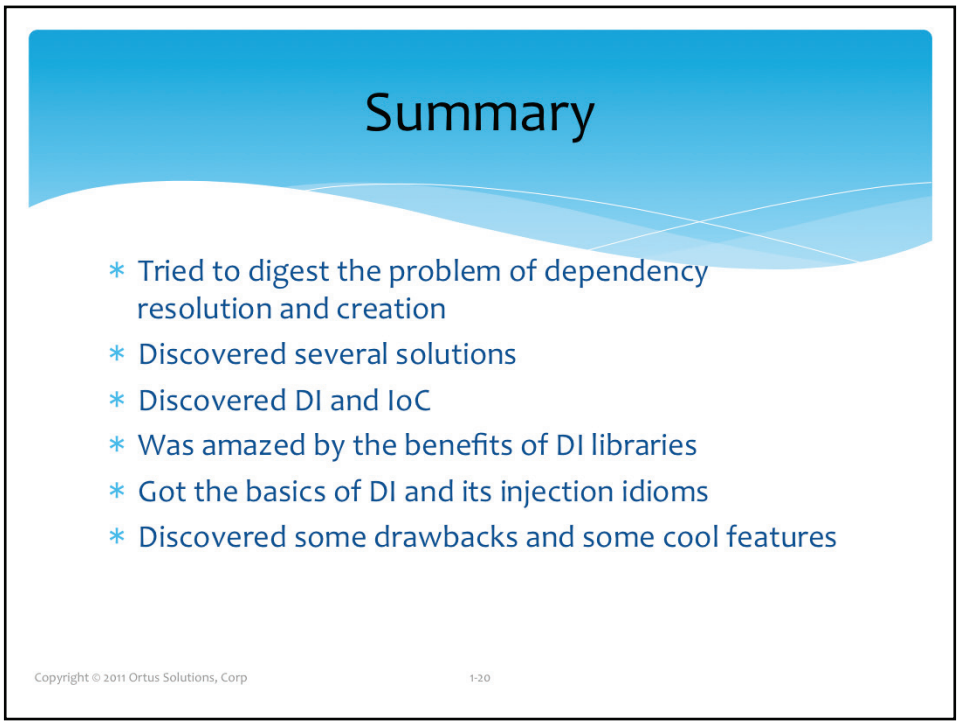
Circular references are something to watch out for when using a DI library as you might bring down a server or two with infinite recursive calls. This side effect is important to mention as it can bring forth catastrophic issues. However, this side effect is only seen when using the constructor argument idiom as it becomes the typical chicken and the egg problem, which one came first? There are various ways to solve these issues and each DI library may implement on or more approaches to it. However, please note that each solution might present other issues as well. We definitely recommend further reading on this topic (Geek Alert!):

- Usage of setter injection
- Usage of proxy objects
- Usage of delayed providers



One of the most useful and undervalued features of a DI library is its capability to provide injector hierarchies. This enables the discovery of objects by traversing a direct chain of ancestry. When a client requests for an object to an injector, the injector then tries to locate it. If unsuccessful and a parent is available, then the injector calls its parent for localization and construction. The parent then starts the process all over again until either the object is found and traversed back down or an exception is thrown. This provides with great ability to:

- Override objects via hierarchy
- Great lookup mechanisms
- Ability to reuse injectors
- Ability to adapt old legacy factories or other DI libraries into each other
- Be an extreme DI nerd!


Summary
* Tried to digest the problem of dependency resolution and creation
* Discovered several solutions
* Discovered DI and IoC
* Was amazed by the benefits of DI libraries
* Got the basics of DI and its injection idioms
* Discovered some drawbacks and some cool features
Copyright © 2011 Ortus Solutions, Corp
1-20