



# Advanced Spring IoC Container, Google Guice, and Dagger

Charles Zhang

Fall 2019

# What we covered in last lecture?

- What is software architecture?
- How to measure the performance of applications?
- Why do we layer enterprise applications?
- What is SOA and what are the pros and cons?
- What does Spring provide and why is it popular?
- What are IoC and DI?
- How do we configure and wire up beans in Spring?

# Outline

- Advanced Spring IoC topics
- Google Guice
- Dagger

# Factory bean in action (Review)

```
<beans ...>
  <bean id="aaa"
    class="com.apress.springrecipes.shop.DiscountFactoryBean">
    <property name="product">
      <bean class="com.apress.springrecipes.shop.Battery">
        <constructor-arg value="AAA" />
        <constructor-arg value="2.5" />
      </bean>
    </property>
    <property name="discount" value="0.2" />
  </bean>

  <bean id="cdrw"
    class="com.apress.springrecipes.shop.DiscountFactoryBean">
    <property name="product">
      <bean class="com.apress.springrecipes.shop.Disc">
        <constructor-arg value="CD-RW" />
        <constructor-arg value="1.5" />
      </bean>
    </property>
    <property name="discount" value="0.1" />
  </bean>
```

# Create beans by a static factory method

```
public class ProductCreator {  
    public static Product createProduct(String productId) {  
        if ("aaa".equals(productId)) {  
            return new Battery("AAA", 2.5);  
        } else if ("cdrw".equals(productId)) {  
            return new Disc("CD-RW", 1.5);  
        }  
        throw new IllegalArgumentException("Unknown product");  
    }  
}
```

```
<beans ...>  
<bean id="aaa" class="com.apress.springrecipes.shop.ProductCreator"  
    factory-method="createProduct"  
    <constructor-arg value="aaa" />  
</bean>  
  
<bean id="cdrw" class="com.apress.springrecipes.shop.ProductCreator"  
    factory-method="createProduct"  
    <constructor-arg value="cdrw" />  
</bean>  
</beans>
```

What if the  
*factory-method*  
property is not  
there?

Factory method *or* factory class?

# Create beans by an instance factory method

```
public class ProductCreator {  
    private Map<String, Product> products;  
  
    public void setProducts(Map<String, Product> products) {  
        this.products = products;  
    }  
  
    public Product createProduct(String productId) {  
        Product product = products.get(productId);  
        if (product != null) {  
            return product;  
        }  
        throw new IllegalArgumentException("Unknown product");  
    }  
}
```

```
<beans ...>  
  <bean id="productCreator"  
    class="com.apress.springrecipes.shop.ProductCreator">  
    <property name="products">  
      <map>  
        <entry key="aaa">  
          <bean class="com.apress.springrecipes.shop.Battery">  
            <property name="name" value="AAA" />  
            <property name="price" value="2.5" />  
          </bean>  
        </entry>  
        <entry key="cdrw">  
  
          <bean id="cdrw" factory-bean="productCreator"  
            factory-method="createProduct">  
              <constructor-arg value="cdrw" />  
            </bean>  
        </entry>  
      </map>  
    </property>  
  </bean>  
</beans>
```

- Create a bean for the *factory* class
- Specify *factory-bean* and *factory-method* for the actual bean

# Create beans from *static fields* - FieldRetrievingFactoryBean

```
public abstract class Product {  
    public static final Product AAA = new Battery("AAA", 2.5);  
    public static final Product CDRW = new Disc("CD-RW", 1.5);  
    ...  
}
```

```
<beans ...>  
  <bean id="aaa" class="org.springframework.beans.factory.config.↵  
    FieldRetrievingFactoryBean">  
    <property name="staticField">  
      <value>com.apress.springrecipes.shop.Product.AAA</value>  
    </property>  
  </bean>
```

```
<util:constant id="aaa"  
  static-field="com.apress.springrecipes.shop.Product.AAA" />
```

- How is the bean aaa created?
  - Identify the class
  - Identify the field
  - What's the type of aaa?



# Create beans from *object properties* - PropertyPathFactoryBean

```
public class ProductRanking {  
    private Product bestSeller;  
  
    public Product getBestSeller() {  
        return bestSeller;  
    }  
}
```

- Steps:
  - Create the reference bean
  - Specify the reference ID and path

```
<beans ...>  
    <bean id="productRanking"  
        class="com.apress.springrecipes.shop.ProductRanking">  
        <property name="bestSeller">  
            <bean class="com.apress.springrecipes.shop.Disc">  
                <property name="name" value="CD-RW" />  
                <property name="price" value="1.5" />  
            </bean>  
        </property>  
    </bean>  
  
    <bean id="bestSeller"  
        class="org.springframework.beans.factory.config.PropertyPathFactoryBean">  
        <property name="targetObject" ref="productRanking" />  
        <property name="propertyPath" value="bestSeller" />  
    </bean>  
</beans>
```

The simpler way with schema/util

```
<util:property-path id="bestSeller" path="productRanking.bestSeller" />
```



# Bean scopes: default to singleton

Default

When to use  
Singleton and  
Prototype?

Scope	Description
Singleton	Creates a single bean instance per Spring IoC container
Prototype	Creates a new bean instance each time when requested
Request	Creates a single bean instance per HTTP request; only valid in the context of a web application
Session	Creates a single bean instance per HTTP session; only valid in the context of a web application
GlobalSession	Creates a single bean instance per global HTTP session; only valid in the context of a portal application

# Bean scopes, continued

Same cart or not?

```
<beans ...>
  <bean id="aaa" class="com.apress.springrecipes.shop.Battery">
    <property name="name" value="AAA" />
    <property name="price" value="2.5" />
  </bean>

  <bean id="cdrw" class="com.apress.springrecipes.shop.Disc">
    <property name="name" value="CD-RW" />
    <property name="price" value="1.5" />
  </bean>

  <bean id="dvdrw" class="com.apress.springrecipes.shop.Disc">
    <property name="name" value="DVD-RW" />
    <property name="price" value="3.0" />
  </bean>

  <bean id="shoppingCart" class="com.apress.springrecipes.shop.ShoppingCart" />
</beans>
```

```
ShoppingCart cart1 = (ShoppingCart) context.getBean("shoppingCart");
cart1.addItem(aaa);
cart1.addItem(cdrw);
System.out.println("Shopping cart 1 contains " + cart1.getItems());

ShoppingCart cart2 = (ShoppingCart) context.getBean("shoppingCart");
cart2.addItem(dvdrw);
System.out.println("Shopping cart 2 contains " + cart2.getItems());
```

```
<bean id="shoppingCart"
      class="com.apress.springrecipes.shop.ShoppingCart"
      scope="prototype" />
```

What about the name uniqueness?

# Singleton or Non-singleton or Static class?

- Natural examples of singleton
- Pros of singleton
  - Natural, conceptual simplicity
  - Memory friendly
- Cons of singleton
  - May not simplify concurrency control
  - Not allowing inheritance
- What about static classes?

# How is Spring's singleton different from Java's singleton?

- Conventional way of creating singletons in Java
  - Private constructor
  - public *static* getInstance() method
- When do you create the instance?
  - Static instance
  - Lazy creation
- Spring's way is simpler, but with caveats
  - Singleton within the application context only
  - How do you break it?

# Initialization and destruction

- Implementing InitializingBean and DisposableBean

```
public class Cashier implements InitializingBean, DisposableBean {  
    ...  
    public void afterPropertiesSet() throws Exception {  
        openFile();  
    }  
  
    public void destroy() throws Exception {  
        closeFile();  
    }  
}
```

# Easier ways for init and destroy

- Setting the init-method and destroy-method  
Attributes

```
<bean id="cashier1" class="com.apress.springrecipes.shop.Cashier"  
    init-method="openFile" destroy-method="closeFile">  
    <property name="name" value="cashier1" />  
    <property name="path" value="c:/cashier" />  
</bean>
```

- Use @PostConstruct and @PreDestroy

## @PostConstruct

```
public void openFile() throws IOException {  
    File logFile = new File(path, name + ".txt");  
    writer = new BufferedWriter(new OutputStreamWriter(  
        new FileOutputStream(logFile, true)));  
}
```

## @PreDestroy

```
public void closeFile() throws IOException {  
    writer.close();  
}
```



# Reducing XML with annotation

```
@Configuration
public class PersonConfiguration {
    @Bean(name = "theProgrammer")
    public Person josh() {
        Person josh = new Person();
        josh.setName("Josh");
        return josh ;
    }
}
```

```
@Bean( initMethod = "startLife", destroyMethod = "die")
public Person companyLawyer() {
    Person companyLawyer = new Person();
    companyLawyer.setName("Alan Crane");
    return companyLawyer;
}
```

# Making Beans Aware of the Container

Aware Interface	Target Resource
BeanNameAware	The bean name of its instances configured in the IoC container
BeanFactoryAware	The current bean factory, through which you can invoke the container's services
ApplicationContextAware*	The current application context, through which you can invoke the container's services
MessageSourceAware	A message source, through which you can resolve text messages

```
public class Cashier implements BeanNameAware {  
    ...  
    public void setBeanName(String beanName) {  
        this.name = beanName;  
    }  
}
```

# Google Guice

- Disclaimer: the Guice slides here are based on the content in the Google Guice Documentation linked below:

<https://github.com/google/guice/wiki/Motivation>

# What is Guice?

- Google's open source dependency injection framework for Java
- First generic framework for dependency injection using Java annotations
- Supports programmatically binding interfaces to implementations
- Supports injection through constructors, methods, and fields

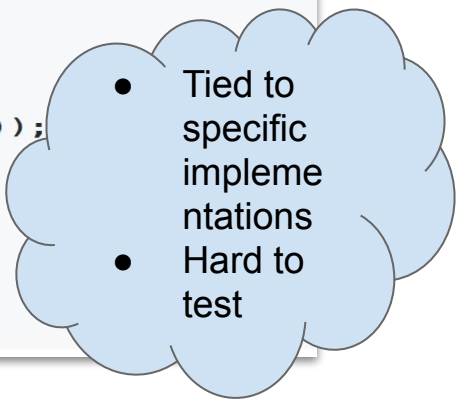
# The Case for Guice (or Other Dependency Injection Frameworks)

```
public interface BillingService {  
  
    /**  
     * Attempts to charge the order to the credit card. Both successful and  
     * failed transactions will be recorded.  
     *  
     * @return a receipt of the transaction. If the charge was successful, the  
     *         receipt will be successful. Otherwise, the receipt will contain a  
     *         decline note describing why the charge failed.  
     */  
    Receipt chargeOrder(PizzaOrder order, CreditCard creditCard);  
}
```

Start with a billing service interface that we want to implement and test

# World without Dependency Injection

```
public class RealBillingService implements BillingService {  
    public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {  
        CreditCardProcessor processor = new PaypalCreditCardProcessor();  
        TransactionLog transactionLog = new DatabaseTransactionLog();  
  
        try {  
            ChargeResult result = processor.charge(creditCard, order.getAmount());  
            transactionLog.logChargeResult(result);  
  
            return result.isSuccessful()  
                ? Receipt.forSuccessfulCharge(order.getAmount())  
                : Receipt.forDeclinedCharge(result.getDeclineMessage());  
        } catch (UnreachableException e) {  
            transactionLog.logConnectException(e);  
            return Receipt.forSystemFailure(e.getMessage());  
        }  
    }  
}
```

- 
- Tied to specific implementations
  - Hard to test

Make direct constructor calls - what are the potential problems?



# How About Using Factories?

```
public class CreditCardProcessorFactory {  
  
    private static CreditCardProcessor instance;  
  
    public static void setInstance(CreditCardProcessor processor) {  
        instance = processor;  
    }  
  
    public static CreditCardProcessor getInstance() {  
        if (instance == null) {  
            return new SquareCreditCardProcessor();  
        }  
  
        return instance;  
    }  
}
```

```
public class RealBillingService implements BillingService {  
    public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {  
        CreditCardProcessor processor = CreditCardProcessorFactory.getInstance();  
        TransactionLog transactionLog = TransactionLogFactory.getInstance();  
  
        try {  
            ChargeResult result = processor.charge(creditCard, order.getAmount());  
            transactionLog.logChargeResult(result);  
  
            return result.isSuccessful()  
                ? Receipt.forSuccessfulCharge(order.getAmount())  
                : Receipt.forDeclinedCharge(result.getDeclineMessage());  
        } catch (UnreachableException e) {  
            transactionLog.logConnectException(e);  
            return Receipt.forSystemFailure(e.getMessage());  
        }  
    }  
}
```

# Proper Unit Testing with Factories

```
public class RealBillingServiceTest extends TestCase {

    private final PizzaOrder order = new PizzaOrder(100);
    private final CreditCard creditCard = new CreditCard("1234", 11, 2010);

    private final InMemoryTransactionLog transactionLog = new InMemoryTransactionLog();
    private final FakeCreditCardProcessor processor = new FakeCreditCardProcessor();

    @Override public void setUp() {
        TransactionLogFactory.setInstance(transactionLog);
        CreditCardProcessorFactory.setInstance(processor);
    }

    @Override public void tearDown() {
        TransactionLogFactory.setInstance(null);
        CreditCardProcessorFactory.setInstance(null);
    }

    public void testSuccessfulCharge() {
        RealBillingService billingService = new RealBillingService();
        Receipt receipt = billingService.chargeOrder(order, creditCard);

        assertTrue(receipt.hasSuccessfulCharge());
        assertEquals(100, receipt.getAmountOfCharge());
        assertEquals(creditCard, processor.getCardOfOnlyCharge());
        assertEquals(100, processor.getAmountOfOnlyCharge());
        assertTrue(transactionLog.wasSuccessLogged());
    }
}
```

- It works
  - -with drawbacks
- Can we make it even easier?

# Manual Dependency Injection

```
public class RealBillingService implements BillingService {  
    private final CreditCardProcessor processor;  
    private final TransactionLog transactionLog;  
  
    public RealBillingService(CreditCardProcessor processor,  
        TransactionLog transactionLog) {  
        this.processor = processor;  
        this.transactionLog = transactionLog;  
    }  
}
```

```
public class RealBillingServiceTest extends TestCase {  
  
    private final PizzaOrder order = new PizzaOrder(100);  
    private final CreditCard creditCard = new CreditCard("1234", 11, 2010);  
  
    private final InMemoryTransactionLog transactionLog = new InMemoryTransactionLog();  
    private final FakeCreditCardProcessor processor = new FakeCreditCardProcessor();  
  
    public void testSuccessfulCharge() {  
        RealBillingService billingService  
            = new RealBillingService(processor, transactionLog);  
        Receipt receipt = billingService.chargeOrder(order, creditCard);  
    }  
}
```

- Extract dependencies through a constructor
- Manually inject dependencies through explicit constructor invocation
- Can we make it even easier?

# Do it Guice's way - Use @Inject to declare dependencies

```
class BillingService {  
    private final CreditCardProcessor processor;  
    private final TransactionLog transactionLog;  
  
    @Inject  
    BillingService(CreditCardProcessor processor,  
        TransactionLog transactionLog) {  
        this.processor = processor;  
        this.transactionLog = transactionLog;  
    }  
  
    public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {  
        ...  
    }  
}
```

# Specify Bindings within Modules

- Bindings map types to implementations
- A module is a collection of bindings
  - using fluent method calls

```
public class BillingModule extends AbstractModule {  
    @Override  
    protected void configure() {  
  
        /*  
         * This tells Guice that whenever it sees a dependency on a TransactionLog,  
         * it should satisfy the dependency using a DatabaseTransactionLog.  
         */  
        bind(TransactionLog.class).to(DatabaseTransactionLog.class);  
  
        /*  
         * Similarly, this binding tells Guice that when CreditCardProcessor is used i  
         * a dependency, that should be satisfied with a PaypalCreditCardProcessor.  
         */  
        bind(CreditCardProcessor.class).to(PaypalCreditCardProcessor.class);  
    }  
}
```

# Injector - Guice's Object-Graph Builder

```
public static void main(String[] args) {  
    /*  
     * Guice.createInjector() takes your Modules, and returns a new Injector  
     * instance. Most applications will call this method exactly once, in their  
     * main() method.  
     */  
    Injector injector = Guice.createInjector(new BillingModule());  
  
    /*  
     * Now that we've got the injector, we can build objects.  
     */  
    BillingService billingService = injector.getInstance(BillingService.class);  
    ...  
}
```



# Linked Bindings

```
public class BillingModule extends AbstractModule {  
    @Override  
    protected void configure() {  
        bind(TransactionLog.class).to(DatabaseTransactionLog.class);  
    }  
}
```

Linked bindings can be chained

```
public class BillingModule extends AbstractModule {  
    @Override  
    protected void configure() {  
        bind(TransactionLog.class).to(DatabaseTransactionLog.class);  
        bind(DatabaseTransactionLog.class).to(MySqlDatabaseTransactionLog.class);  
    }  
}
```

# have multiple credit card processors?

```
@BindingAnnotation @Target({ FIELD, PARAMETER, METHOD }) @Retention(RUNTIME)  
public @interface PayPal {}
```

- @BindingAnnotation declares that this is a binding annotation
- @Target({FIELD, PARAMETER, METHOD}) specifies where the binding can be applied
- @Retention(RUNTIME) makes the annotation available at runtime

We can also have  
ApplePay, GooglePay,  
etc.

# Make Use of Binding Annotations

Apply the annotation to the parameters to be injected

```
public class RealBillingService implements BillingService {  
  
    @Inject  
    public RealBillingService(@PayPal CreditCardProcessor processor,  
        TransactionLog transactionLog) {  
        ...  
    }  
}
```

Use the *annotatedWith* clause in the *bind()* statement


```
bind(CreditCardProcessor.class)  
    .annotatedWith(PayPal.class)  
    .to(PayPalCreditCardProcessor.class);
```

# @Named - Built-in Binding Annotation

```
public class RealBillingService implements BillingService {  
  
    @Inject  
    public RealBillingService(@Named("Checkout") CreditCardProcessor processor,  
        TransactionLog transactionLog) {  
        ...  
    }  
}
```

Use `Names.named()` to bind a specific name

```
bind(CreditCardProcessor.class)  
    .annotatedWith(Names.named("Checkout"))  
    .to(CheckoutCreditCardProcessor.class);
```



Pros &  
Cons?

# Instance Binding

Bind a type to a specific instance of that type - useful only for objects without dependencies of their own, e.g., value objects

```
bind(String.class)
    .annotatedWith(Names.named("JDBC URL"))
    .toInstance("jdbc:mysql://localhost/pizza");
bind(Integer.class)
    .annotatedWith(Names.named("login timeout seconds"))
    .toInstance(10);
```

# @Provides Methods

```
public class BillingModule extends AbstractModule {
    @Override
    protected void configure() {
        ...
    }

    @Provides
    TransactionLog provideTransactionLog() {
        DatabaseTransactionLog transactionLog = new DatabaseTransactionLog();
        transactionLog.setJdbcUrl("jdbc:mysql://localhost/pizza");
        transactionLog.setThreadPoolSize(30);
        return transactionLog;
    }
}
```

- @Provides method must be defined within a module, and it must have the annotation
- Returns the bound type - invoked whenever the injector needs an instance of that type
- Bad practice to allow any kind of exception to be thrown -- runtime or checked



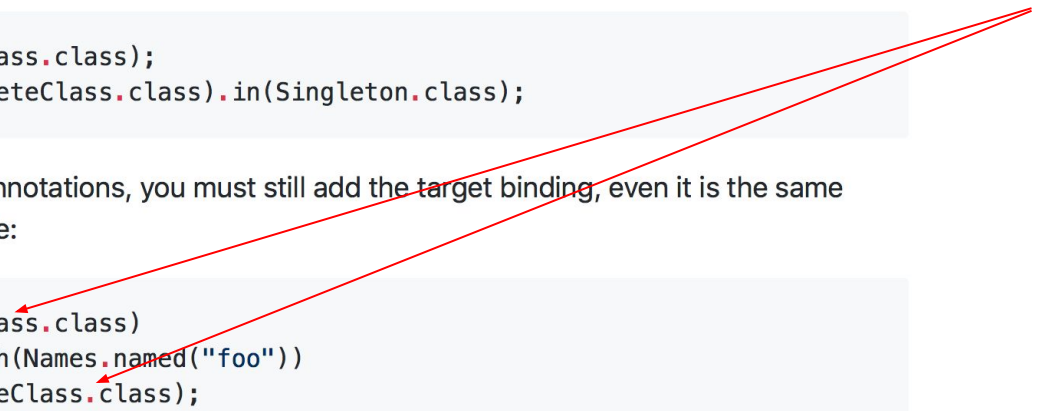
# Untargeted Bindings

You may create bindings without specifying a target. This is most useful for concrete classes and types annotated by either `@ImplementedBy` or `@ProvidedBy`. An untargetted binding informs the injector about a type, so it may prepare dependencies eagerly. Untargetted bindings have no `to` clause, like so:

```
bind(MyConcreteClass.class);  
bind(AnotherConcreteClass.class).in(Singleton.class);
```

When specifying binding annotations, you must still add the target binding, even it is the same concrete class. For example:

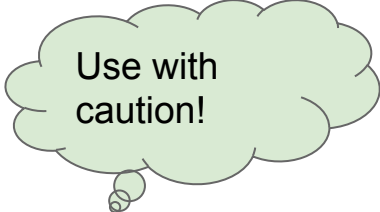
```
bind(MyConcreteClass.class)  
    .annotatedWith(Names.named("foo"))  
    .to(MyConcreteClass.class);  
bind(AnotherConcreteClass.class)  
    .annotatedWith(Names.named("foo"))  
    .to(AnotherConcreteClass.class)  
    .in(Singleton.class);
```



# Constructor Bindings

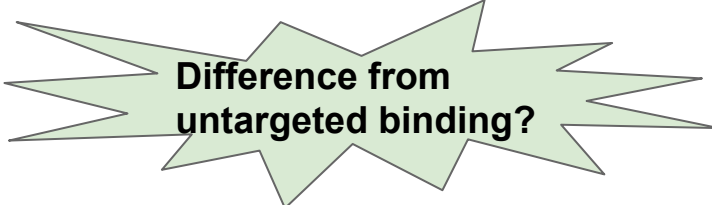
```
public class BillingModule extends AbstractModule {
    @Override
    protected void configure() {
        try {
            bind(TransactionLog.class).toConstructor(
                DatabaseTransactionLog.class.getConstructor(DatabaseConnection.class));
        } catch (NoSuchMethodException e) {
            addError(e);
        }
    }
}
```

- Solve the confusion when there are multiple constructors
- No need to use the *@Inject* annotation on the constructor
- Caveat: each *toConstructor()* binding is scoped independently
- Caveat: manually constructed instances do not participate in



Use with  
caution!

# Just-In-Time Bindings



Difference from  
untargeted binding?

If a type is needed, but no explicit bindings are found, the injector will attempt to create a Just-In-Time binding

```
@ImplementedBy(PayPalCreditCardProcessor.class)
public interface CreditCardProcessor {
    ChargeResult charge(String amount, CreditCard creditCard)
        throws UnreachableException;
}
```

```
@ProvidedBy(DatabaseTransactionLogProvider.class)
public interface TransactionLog {
    void logConnectException(UnreachableException e);
    void logChargeResult(CHargeResult result);
}
```

# Provider Bindings and Lazy Injections

The *Provider* interface

```
public interface Provider<T> {  
    T get();  
}
```

Lazy injection with providers

```
public class DatabaseTransactionLog implements TransactionLog {  
  
    private final Provider<Connection> connectionProvider;  
  
    @Inject  
    public DatabaseTransactionLog(Provider<Connection> connectionProvider) {  
        this.connectionProvider = connectionProvider;  
    }  
  
    public void logChargeResult(ChargeResult result) {  
        /* only write failed charges to the database */  
        if (!result.wasSuccessful()) {  
            Connection connection = connectionProvider.get();  
        }  
    }  
}
```

# Scopes

- Supported scopes
  - `@Singleton`
  - `@RequestScoped`
  - `@SessionScoped`
- How to specify scopes

```
@Singleton
public class InMemoryTransactionLog implements TransactionLog {
    /* everything here should be threadsafe! */
}
```

```
@Provides @Singleton
TransactionLog provideTransactionLog() {
    ...
}
```

- How to make bind singletons?

```
bind(Bar.class).to(Applebees.class).in(Singleton.class);
bind(Grill.class).to(Applebees.class).in(Singleton.class);
```

# Eager Singleton, or Lazy Singleton?

	PRODUCTION	DEVELOPMENT
<code>.asEagerSingleton()</code>	eager	eager
<code>.in(Singleton.class)</code>	eager	lazy
<code>.in(Scopes.SINGLETON)</code>	eager	lazy
<code>@Singleton</code>	eager*	lazy

# Best Practices

- ❑ Minimize mutability
- ❑ Inject only direct dependencies
- ❑ Avoid cyclic dependencies
- ❑ Avoid static state
- ❑ Use @Nullable
- ❑ Modules should be fast and side-effect free
- ❑ Be careful about I/O in Providers
- ❑ Avoid conditional logic in modules
- ❑ Avoid binding Closable resources



# From Guice to Dagger: move toward lightweight injection

- Runtime injection with reflection
  - Slow initialization
  - Late failure
  - Slow injection
  - Non-trivial memory impact.

Guice

- Injection based on static analysis
- Fail as early as possible
  - compile time, not runtime.
- No reflection at runtime
  - Super fast!
- Little memory impact
- Detects cyclic dependencies
- Fits well with Android

Dagger

# Dagger example

```
class CoffeeMaker {  
    @Inject Heater heater;  
    @Inject Pump pump;  
  
    ...  
}
```

```
@Module  
class DripCoffeeModule {  
    @Provides Heater provideHeater() {  
        return new ElectricHeater();  
    }  
  
    @Provides Pump providePump(Thermosiphon pump) {  
        return pump;  
    }  
}
```

- Mostly using `@Inject` and `@Provides`
- No explicit binding command
- All `@Provides` methods belong to a module
- `@Module` marks a module class

# Caveats of Dagger

- No cyclic dependencies (DAG)
- No shortcut to bind interfaces to impls
- No late, dynamic, or conditional binding
- No servlet support, yet (available in 3rd party libs)