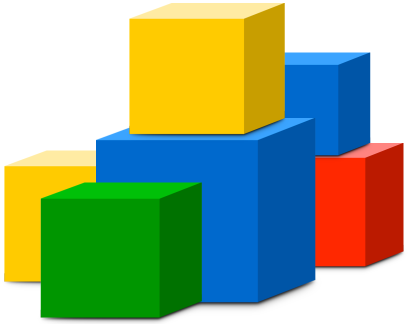
Google®

Google
Developer
Day2007

Java on Guice

Dependency Injection the *Java Way*

Bob Lee

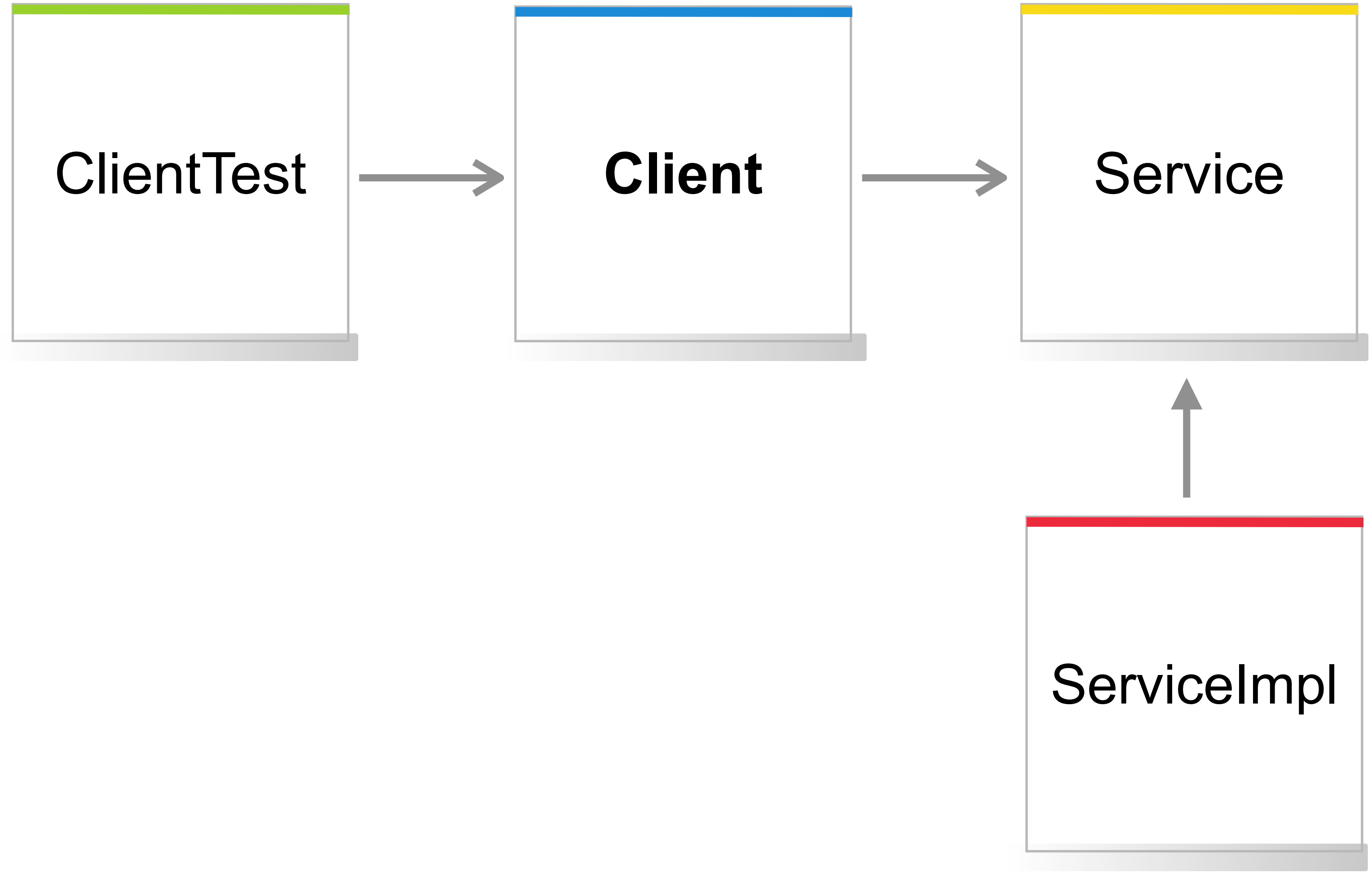


What can **dependency injection** do for me?

- Easier testing
- More decoupling
- Less boilerplate
- **Better maintainability**

A **Simple** Example

High Level Design

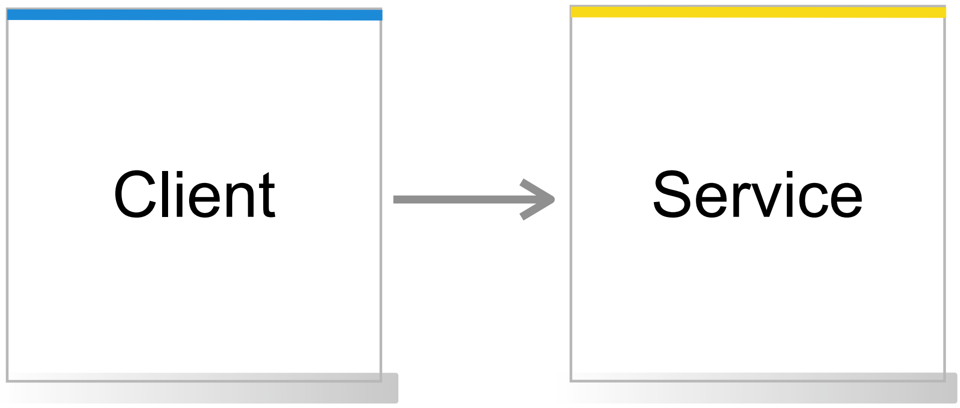


We’ll examine 3 approaches...

1. The Factory Pattern
2. Dependency Injection *by Hand*
3. Dependency Injection *with **Guice***

One Variable

- From approach to approach, how does *Client* get a *Service?*

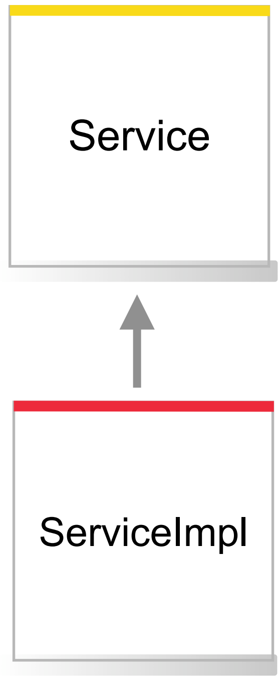


A Few Constants

- Regardless of the approach, *Service* stays the same:

```
public interface Service {
  void go();
}


public class ServiceImpl
    implements Service {
  public void go() {
    // Some expensive stuff.
    ...
  }
}
```



Mock Service

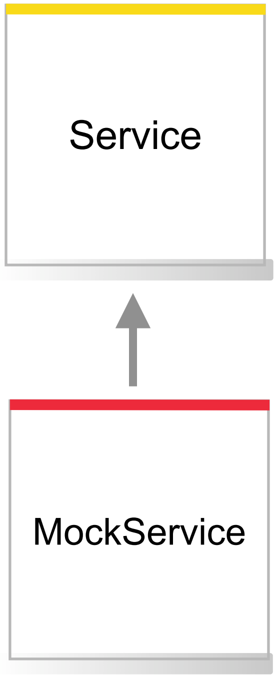
- We also need a mock implementation of *Service* which we can use to test clients:

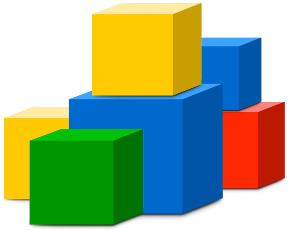
```
public class MockService
    implements Service {

  private boolean gone = false;

  public void go() {
    gone = true;
  }

  public boolean isGone() {
    return gone;
  }
}
```





Approach #1:
The Factory Pattern

The Factory Client

```
public class Client {

  public void go() {
    Service service = ServiceFactory.getInstance();
    service.go();
  }
}
```

Service Factory

```java
public class ServiceFactory {

  private ServiceFactory() {}

  private static Service instance = new ServiceImpl();

  public static Service getInstance() {
    return instance;
  }

  public static void setInstance(Service service) {
    instance = service;
  }
}
```

A Unit Test

```
public void testClient() {
  Service previous = ServiceFactory.getInstance();
  try {
    final MockService mock = new MockService();
    ServiceFactory.setInstance(mock);
    Client client = new Client();
    client.go();
    assertTrue(mock.isGone());
  }
  finally {
    ServiceFactory.setInstance(previous);
  }
}
```

Factory Observations

- Our unit test had to pass the mock to the factory and then clean up afterwards.
- You have to look at the implementation of Client to know it depends on Service.
- Reusing Client in a different context will be difficult.
- We have to write the same factory code for every dependency.
- Client has a compile time dependency on ServiceImpl.

Approach #2:
Dependency Injection *by Hand*

“Don’t call me. I’ll call you.”

```
public class Client {

  private final Service service;

  public Client(Service service) {
    this.service = service;
  }

  public void go() {
    service.go();
  }
}
```

The Factory-based Unit Test (Again)

```
public void testClient() {
  Service previous = ServiceFactory.getInstance();
  try {
    final MockService mock = new MockService();
    ServiceFactory.setInstance(mock);
    Client client = new Client();
    client.go();
    assertTrue(mock.isGone());
  }
  finally {
    ServiceFactory.setInstance(previous);
  }
}
```

The Test With Dependency Injection

```
public void testClient() {
  MockService mock = new MockService();
  Client client = new Client(mock);
  client.go();
  assertTrue(mock.isGone());
}
```

Passing Service to the Client

```
public static class ClientFactory {

  private ClientFactory() {}

  public static Client getInstance() {
    Service service = ServiceFactory.getInstance();
    return new Client(service);
  }
}
```

Further Observations

- In our test, we now pass our mock directly to Client.
  - No middle man
- You can't create a Client without providing a Service.
  - Fewer unexpected surprises.
- We can easily reuse Client with multiple different Service implementations, even in the same application.
- Client no longer depends on ServiceImpl at compile time.
  - We moved the dependency to the application level.
- *We have to write even more factory code.*

Approach #3:
Dependency Injection with **Guice**

Why use a framework?

- Writing factories is tedious
  - Scopes
- We need more up front checking
- We want more flexibility
- Make it easier to do the right thing

In place of factories, we have **modules**.

```
public class MyModule extends AbstractModule {
  protected void configure() {
    bind(Service.class)
      .to(ServiceImpl.class)
      .in(Scopes.SINGLETON);
  }
}
```

And we apply **@Inject**...

```
public class Client {

  private final Service service;

  @Inject
  public Client(Service service) {
    this.service = service;
  }

  public void go() {
    service.go();
  }
}
```

Our test stays exactly the same.

```
public void testClient() {
  MockService mock = new MockService();
  Client client = new Client(mock);
  client.go();
  assertTrue(mock.isGone());
}
```

Conclusions

- Guice requires much less boilerplate code
  - ~20% for this simple example
  - The more you use a dependency, the more you save.
- More startup checks
- Declarative scopes
- More flexibility
- **Easier up front design decisions**

Getting Started with **Guice**

Bootstrapping

- Objects must be “in the club” to be injected.

```
public class MyApplication {
  public static void main(String[] args) {
    Injector injector = Guice.createInjector(new MyModule());
    Client client = injector.getInstance(Client.class);
    client.go();
  }
}

public class MyModule extends AbstractModule {
  protected void configure() {
    bind(Service.class)
      .to(ServiceImpl.class)
      .in(Scopes.SINGLETON);
  }
}
```

Adding a Dependency

- Service is “in the club.”

```
public class ServiceImpl implements Service {
  @Inject Emailer emailer;
  public void go() {
    // Some expensive stuff.
    ...
    // Send confirmation.
    emailer.send(...);
  }
}

public class Emailer {
  ...
}
```

Handling Multiple Implementions

- Use a **binding annotation**

```
public class ServiceImpl implements Service {
  @Inject @Transactional Emailer emailer;
  public void go() {
    ...
  }
}

public class MyModule extends AbstractModule {
  protected void configure() {
    bind(Service.class)
      .to(ServiceImpl.class)
      .in(Scopes.SINGLETON);
    bind(Emailer.class)
      .annotatedWith(Transational.class)
      .to(TransactionalEmailer.class);
  }
}
```

Providing Objects Manually

```
bind(Emailer.class).toProvider(new Provider<Emailer>() {
  @Inject @Named(“email.host”) String emailHost;
  public Emailer get() {
    return new Emailer(emailHost);
  }
}).in(Scopes.SINGLETON);
```

Scopes

- **Scope:** a policy for reusing objects
- Two ways to specify a scope:

```
bind(Emailer.class).in(Scopes.SINGLETON);
```

or

```
@Singleton
class Emailer {
  ...
}
```

Delaying Provision

```
@Inject
void injectAtm(Provider<Money> atm) {
  Money one = atm.get();
  Money two = atm.get();
  ...
}
```

Constructor vs. Method vs. Field Injection

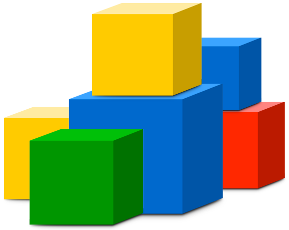
- Prefer constructor injection
  - You can use final fields
- Use method injection when constructor injection won’t work. For example:
  - If you don’t want subclasses to know about your dependencies
  - If Guice can’t create your objects
- Use field injection when you need concision and don’t care about using your class outside of Guice
  - Custom providers
  - *Slides for your talk*

Other Notable Features

- Type conversion for constants
- AOP Alliance-based method interception
- Development stages
- Optional injection
- Integration with:
  - JNDI
  - Spring
  - JMX
  - Struts2

Upcoming Features

- Provider methods
- Mixed automatic and custom injection
- Construction listeners



Questions?

Google®