Creating Graphs of Objects

At runtime, OO programs create object graphs in many ways.

class Server {
    Protocol _protocol;
    Authorizer _auth;
    Logger _logger;

    // field initialization
    ErrorHandler _eh = new ErrorHandler();

    // constructor initialization
    public Server(Protocol p) {.... }

    // setter initialization
    public setAuthorizer(Authorizer a) { this._auth = a;}

    // method initialization
    public run() {
        _logger = new Logger();
        ...
    }
}
For improving reusability, it is important to keep the class open.

// Rule #1: No dependency to concrete types
class Server {

    IProtocol _protocol;
    IAuthorizer _auth;
    ILogger _logger;

    // Rule #2: No hard coded types
    IErrorHandler _eh = new ErrorHandler();

    public Server(IProtocol p) {... }

    public setAuthorizer(IAuthorizer a) { this._auth = a;}

    // method initialization
    public run() {
        _logger = new Logger();
        ...
    }
}
The Dependency Injection Design Pattern (Fowler)

- Facilitates reuse and testing
- Concrete types are "injected" using constructors or methods
- Most used is Fowler's Constructor Injection

Problem #1: **Long** chains of constructors
Problem #2: error-prone
Google Guice

- is a framework for dependency injection developed at Google
- Component is called Module
- http://code.google.com/p/google-guice/

Manually injected dependency versus Automatically injected dependency
First Guice Example

- A Webserver is composed of one scheduler and one handler
  - Scheduler: Sequence, MultiThread
  - Handler: Constant, File, Dispatcher

Module webserver = new AbstractModule() {
    @Override
    protected void configure() {
        bind(IRequestHandler.class).to(HelloWorldRequestHandler.class);
        bind(IScheduler.class).to(MultiThreadScheduler.class);
    }
};

Guice.createInjector(webserver).getInstance(RequestReceiver.class).run();

No constructors and Automated bindings.
Behind the scene

```java
public class RequestReceiver implements Runnable {
    @Inject
    private IScheduler s;
    @Inject
    private IRequestHandler rh;
}
```

```java
public class RequestAnalyzer implements IRequestHandler {
    @Inject (optional = true)
    private ILogger l;
}
```

- One single annotation
- If optional, bindings are not required
- All fields can be made private with no constructor
Constructor injection

```java
public interface ILogHeader {
    public String getLogHeader();
}

public class DynConfigurableLogger implements ILogger {
    private ILogHeader _header;
    @Inject
    public DynConfigurableLogger(ILogHeader o) {
        _header = o;
    }
    public void log(String msg) {
        System.err.println(_header.getLogHeader()+msg);
    }
}

bind(ILogHeader.class).to(DateLogHeader.class);
bind(ILogger.class).to(DynConfigurableLogger.class);

Pattern and Guice can co-exist.
```
Linked Bindings

Linked bindings map a type to its implementation.

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

You can even link the concrete `DatabaseTransactionLog` class to a subclass:

```java
bind(DatabaseTransactionLog.class).to(MySqlDatabaseTransactionLog.class);
```

Linked bindings can also be chained:

```java
public class BillingModule extends AbstractModule {
    @Override
    protected void configure() {
        // TransactionLog instances will be MySqlTransactionLog
        bind(TransactionLog.class).to(DatabaseTransactionLog.class);
        bind(DatabaseTransactionLog.class).to(MySqlTransactionLog.class);
    }
}
```
Problem: No all objects are similar, esp. in the presence of decorated objects.

```java
/** Extracts the requested URI from HTPP */
public class RequestAnalyzer implements IRequestHandler {
    // this should be a FileAnalyzer
    @Inject
    private IRequestHandler rh;
}

public class RequestReceiver implements Runnable {
    // this should be a RequestAnalyzer
    @Inject
    private IRequestHandler rh;
}
Tagged bindings (annotatedWith)

```java
class RequestAnalyzer implements IRequestHandler {
    @Inject @Named("RequestAnalyzerBinding")
    private IRequestHandler rh;
}

Module webserver = new AbstractModule() {
    @Override
    protected void configure() {
        bind(IRequestHandler.class).to(RequestAnalyzer.class);
        bind(IRequestHandler.class)
            .annotatedWith(Named.named("RequestAnalyzerBinding"))
            .to(FileRequestHandler.class);
        bind(IScheduler.class).to(MultiThreadScheduler.class);
    }
}
```

Bindings can be specialized
Tagged bindings (annotatedWith)

```java
public class RequestAnalyzer implements IRequestHandler {
    @Inject @RequestAnalyzerBinding
    private IRequestHandler rh;
}

@Retention(RetentionPolicy.RUNTIME) @BindingAnnotation
public @interface RequestAnalyzerBinding { }

Module webserver = new AbstractModule() {
    @Override
    protected void configure() {
        bind(IRequestHandler.class).to(RequestAnalyzer.class);
        bind(IRequestHandler.class)
            .annotatedWith(RequestAnalyzerBinding.class)
            .to(FileRequestHandler.class);
        bind(IScheduler.class).to(MultiThreadScheduler.class);
    }
}
```

Bindings can be specialized
public class ConfigurableLogger implements ILogger {
    @Inject @Named("ConfigurableLoggerHeader")
    private String header;

    public void log (String msg) {
        System.err.println(header+msg);
    }
}

Module webserver = new AbstractModule() {
    @Override
    protected void configure() {
        bind(ILogger.class).to(ConfigurableLogger.class);
        bind(String.class)
            .annotatedWith(Names.named("ConfigurableLoggerHeader"))
            .toInstance(">>> ");
        // equivalent to
        bindConstant()
            .annotatedWith(Names.named("ConfigurableLoggerHeader")).to(">>> ");
    }
}

Fields can be initialized. Useful for CONSTANTS.
(Mind the private)
Guice Component Composition Operators

• In class Modules:
  • combine (Module... modules) -> Module: Returns a new module that combines the bindings of \( m_1 \) \( \ldots \) \( m_n \). Crashes with CreationException if concurrent bindings.
  • override (Module... modules) -> ModuleBuilder: Returns a builder that creates a module that overlays override modules over the given modules. "with" must be called on the returned object.

Module functionalTestModule = Modules.override(new ProductionModule()).with(new TestModule());
Main Concepts

A **module** binds abstract types to concrete types.

```java
class ServerModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(IRequestHandler.class).to(RequestAnalyzer.class);
    }
}
```

An **injector** is a module transformed into a factory for:

- creating new instances
- enriching existing instances

```java
Injector injector = Guice.createInjector(new ServerModule());
IRequestHandler obj = injector.getInstance(IRequestHandler.class);
//set all injectable fields
injector.injectMembers(new RequestAnalyzer());
```
Main Concepts

A field can be **injectable** using an annotation `@Inject`. The injection may be optional.

```java
@Inject (optional = true)
private ILogger l;
```

The scope of an injection can be restricted using an annotation `@Named`.

```java
@Inject    @Named("RequestAnalyzerBinding")
private IRequestHandler rh;
```

```java
bind(IRequestHandler.class)
    .annotatedWith(Names.named("RequestAnalyzerBinding"))
    .to(FileRequestHandler.class);
```
Intermediate Concepts

Default implementation classes can be specified using the annotation `@ImplementedBy` (no more `bind().to()`)

```java
@ImplementedBy(BasicLogger.class)
public interface ILogger {
    void log (String msg);
}
```

Note: the default implementation may be overridden by `bind().to()`
Intermediate Concepts

A Singleton can be specified using the annotation @Singleton

```java
@Singleton
public class DatabaseConnection implements Connection {
    ...
}
```

Note: this simply discards all uses of the Singleton implementation pattern.
A tailored object can be created with a **provider** method using the `@Provides` annotation

```java
Module webserver = new AbstractModule() {
    @Override
    protected void configure() {
        bind(IRequestHandler.class).to(HelloWorldRequestHandler.class);
    }

    @Provides
    IScheduler newIScheduler() {
        return new MultiThreadScheduler() {
            @Override
            public Thread configure(Thread thread) {
                thread.setUncaughtExceptionHandler(
                    new UncaughtExceptionHandler() {
                        @Override
                        public void uncaughtException(Thread t, Throwable e) {
                            Log.debug(e.getLocalizedMessage());
                        }
                    });

                return super.configure(thread);
            }
        }
    }
};
```

**Note:** `@Provides` necessarily appears in Module class.
Guice definition of software component

• In Guice, a software component definition is a class implementing Module (or extending AbstractModule).

• A component instance is graph of objects that are bound using automated dependency injection.