google/guice

How to start doing dependency injection with Guice.

Getting Started

With dependency injection, objects accept dependencies in their constructors. To construct an object, you first build its dependencies. But to build each dependency, you need *its* dependencies, and so on. So when you build an object, you really need to build an *object graph*.

Building object graphs by hand is labour intensive, error prone, and makes testing difficult. Instead, Guice can build the object graph for you. But first, Guice needs to be configured to build the graph exactly as you want it.

To illustrate, we'll start the `BillingService` class that accepts its dependent interfaces `CreditCardProcessor` and `TransactionLog` in its constructor. To make it explicit that the `BillingService` constructor is invoked by Guice, we add the `@Inject` annotation:

```
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) {
    ...
  }
}
```

We want to build a `BillingService` using `PaypalCreditCardProcessor` and `DatabaseTransactionLog`. Guice uses *bindings* to map types to their implementations. A *module* is a collection of bindings specified using fluent, English-like 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 in
      * a dependency, that should be satisfied with a
PaypalCreditCardProcessor.
      */
    bind(CreditCardProcessor.class).to(PaypalCreditCardProcessor.class);
  }
}
```

The modules are the building blocks of an *injector*, which is Guice's object-graph builder. First we create the injector, and then we can use that to build the `BillingService`:

```
 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);
    ...
  }
```

By building the billingService, we've constructed a small object graph using Guice. The graph contains the billing service and its dependent credit card processor and transaction log.