Vanilla Users Guide

Christopher J. Stehno

Version 0.1.0, December 2015

Table of Contents

Introduction

The Vanilla library is a multi-purpose utility library containing a lot of "what if" and experimentation code that actually seemed to have a potential use, so it was made publicly available.

The project web site is: http://stehno.com/vanilla

The source code for the project is hosted on GitHub at: https://github.com/cjstehno/vanilla

Vanilla is licensed under the Apache 2 open source license.

Lazy Immutable

The Groovy `@Immutable` annotation is quite useful; however, it is an all-or-nothing object, meaning that you create it once and then you can't touch it again (other than copying it with different properties). The `@LazyImmutable` annotation in Vanilla allows for a slighly more flexible configuration of immutable objects.

Consider the case of of an immutable Person object:

```
@Immutable
class Person {
    String firstName
    String middleName
    String lastName
    int age
}
```

With `@Immutable` you have to create the object all at once:

```
def person = new Person('Chris','J','Stehno',42)
```

and then you're stuck with it. You can create a copy of it with one or more different properties using the `copyWith` method, but you need to specify the `copyWith=true` in the annotation itself, then you can do something like:

```
Person otherPerson = person.copyWith(firstName:'Bob', age:50)
```

With more complicated immutables, this "all at once" requirement can be annoying. This is where the `@LazyImmutable` annotation becomes useful. With a similar Person class:

```
@LazyImmutable @Canonical
class Person {
    String firstName
    String middleName
    String lastName
    int age
}
```

using the new annotation, you can create and populate the instance over time:

```
def person = new Person('Chris')
person.middleName = 'J'
person.lastName = 'Stehno'
person.age = 42
```

Notice that the `@LazyImmutable` annotation does not apply any other transforms (as the standard `@Immutable` does). It's a standard Groovy object, but with an added method: the `asImmutable()` method is injected via AST Transformation. This method will take the current state of the object and create an immutable version of the object - this does imply that the properties of lazy immutable objects should follow the same rules as those of the standard immutable so that the conversion is determinate. For our example case:

```
Person immutablePerson = person.asImmutable()
```

The created object is the same immutable object as would have been created by using the `@Immutable` annotation and it is generated as an extension of the class you created so that it's type is still valid. The immutable version of the object also has a useful added method, the `asMutable()` method is used to create a copy of the original mutable object.

```
Person otherMutable = immutablePerson.asMutable()
```

You can swap back and forth between immutability and mutability as needed without loss of functionality or data integrity.

Property Randomizers

Generating large amounts of meaningful test data can be tedious and sometimes error prone if your test values arbitrarily fit your code and don't properly exercise it. Randomly generating test data objects can be a simple way to test different scenarios across the same tests and also simplify your test writing.

The `PropertyRandomizer` can be used as a builder by using the classes directly or as a DSL.

Operations

The `PropertyRandomizer` instances have six configuration methods and three builder methods. The configuration methods are:

`ignoringTypes(Class…)`
: Specifies the given property types should be ignored and not randomized.

`ignoringProperties(String…)`
: Specifies that the given property names should be ignored and not randomized.

`typeRandomizers(Map<Class,Object>)`
: Allows the specification of multiple type randomizers, where the map is property type to the appropriate randomizer.

`typeRandomizer(Class, Object)`
: Specifies a single randomizer for a property type.

`propertyRandomizers(Map<String,Object>)`
: Allows the specification of multiple property randomizers, where the map is property name to the appropriate randomizer.

`propertyRandomizer(String, Object)`
: Specifies a single randomizer for a property name.

The `Object` configured as a "randomizer" may be either a `PropertyRandomizer` instance or a `Closure`. If a `Closure` is used, it will allow up to two arguments to the `Closure`:

- If no arguments are specified, the returned value must be built using external code
- If one argument is specified, it will be the `Random` instance used by the randomizer for generating random data.
- If two arguments are passed in, the first will be the `Random` as above, and the second will be the map of properties which will be used to populate the target object.

The allowed builder methods are:

`one()`
: Instantiates a single randomized instance of the target object.

`times(int)`
: Instantiates a collection of multiple randomized instances of size equal to the provided count.

`*(int)`
: An alias to the `times(int)` method for operator-overridden shorthand notation.

Randomizer Builder

The `PropertyRandomizer` class allows randomizers to be built directly from the mutator methods of the class. Pulling an example from the unit tests we can see how the builder uses the configuration and builder operations:

```
PropertyRandomizer rando = PropertyRandomizer.randomize(Person)
    .ignoringProperties('bankPin')
    .typeRandomizers(
        (Date)    : { new Date() },
        (Pet)     : { randomize(Pet).one() },
        (String[]): forStringArray()
    )
    .propertyRandomizers(name: { 'FixedValue' })
```

The code above creates a `PropertyRandomizer` for the `Person` class where the `bankPin` property is ignored, the `name` property is "randomized" to a fixed value "FixedValue", and the properties of type `Date`, `Pet`, and `String[]` are randomized using the provided custom randomizers. To generate random instances using this randomizer you could do any of the following:

```
Person personA = rando.one()
Collection<Person> fourPeople = rando.times(4)
Collection<Person> tenPeople = rando * 10
```

Randomizer DSL

Using the DSL version of the randomizer is not really any different than using the builder style. For the example specified above, we would have the following code to achieve the same randomizer:

```
PropertyRandomizer rando = PropertyRandomizer.randomize(Person){
    ignoringProperties 'bankPin'
    typeRandomizers(
        (Date)    : { new Date() },
        (Pet)     : { randomize(Pet).one() },
        (String[]): forStringArray()
    )
    propertyRandomizers name: { 'FixedValue' }
}
```

The main difference being that the operations are specified within a `Closure`, which could be shared or configured externally to the rest of the code.

Provided Randomizers

The library provides a collection of useful randomizers, some of which are configured by default in the `PropertyRandomizer`, in the `Randomizers` utility class. These may be used directly or as randomizer objects for the `PropertyRandomizer` configurations.

Randomizing Simple Types

The `PropertyRandomizer` is not just useful for complex objects, it will also provide random instances of more simple types, such as `String` and `Date` objects.

If the type to be randomized is one of the built-in class randomizers (`String` and the primitive types and their wrapper classes) you can randomize them directly:

```
PropertyRandomizer stringRando = PropertyRandomizer.randomize(String)
PropertyRandomizer dateRando = PropertyRandomizer.randomize(Date)
```

Other, non-default types may also be configured in a similar manner, by adding a `typeRandomizer(Class,Object)`:

```
Class byteArray = ([] as byte[]).class
PropertyRandomizer stringRando = PropertyRandomizer.randomize(byteArray){
    typeRandomizer byteArray, Randomizers.forByteArray()
}
```

This code will create random instances of `byte` arrays. The extra code for determining the `Class` for the `byte` array is a work-around for an odd issue with Groovy typing and byte arrays - not really related to the example at hand.

Generally, this approach should be left to very simple value-type objects. In many cases, you could also just use the randomizers provided in the `Randomizers` class directly.

Object Mappers

When working with legacy or external code-bases you may run into an API where you end up doing a lot of domain object mapping, from one object format to another. The Vanilla Object Mappers framework can help simplify the mappings. It comes in two forms, a dynamic runtime implementation and a compile-time generated implementation.

Generated `ObjectMapper` instances are stateless and thread-safe.

Object Mapper DSL

The DSL used by the `ObjectMapper` framework is shared between both the runtime and compiled implementations. There are four supported mapping statements:

`map <source-name>`
: Maps the specified source property into the destination property of the same name, with no conversion.

`map <source-name> into <destination-name>`
: Maps the specified source property into the specified destination property, with no conversion.

`map <source-name> into <destination-name> using <closure>`
: Maps the specified source property into the specified destination property, with the given conversion closure.

`map <source-name> using <closure>`
: Maps the specified source property int the destination property of the same name, with the given conversion closure.

The conversion closures configured by the `using` clause are standard Groovy closures that are allowed to accept up to three arguments:

- Argument one - source property value
- Argument two - source object reference
- Argument three - destination object reference

A sample of the DSL pulled from the unit tests is shown below:

```
map 'name'
map 'age' into 'years'
map 'startDate' using { Date.parse('MM/dd/yyyy', it) }
map 'birthDate' into 'birthday' using { LocalDate d -> d.format(BASIC_ISO_DATE) }
```

Runtime Mappers

The `RuntimeObjectMapper` is created using its static `mapper(Closure)` method, where the `Closure` contains the configuration DSL. An example would be something like:

```
ObjectMapper inividualToPerson = RuntimeObjectMapper.mapper {
    map 'id'
    map 'givenName' into 'firstName'
    map 'familyName' into 'lastName'
    map 'birthDate' using { d-> LocaleDate.parse(d) }
}
```

Compiled Mappers

Sometimes you need to squeeze a bit more performance out of a mapping operation, or you just want to generate cleaner code. The `InjectCompiledObjectMapper` annotation is used to annotate a field or property to inject a compile-time generated `ObjectMapper` implementation (using AST Transformations) based on the supplied DSL configuration.

Using the compiled approach is as simple as the runtime approach, you just write the DSL code in the `@InjectObjectMapper` annotation on a method, or field:

```
class ObjectMappers {

    @InjectObjectMapper({
        map 'id'
        map 'givenName' into 'firstName'
        map 'familyName' into 'lastName'
        map 'birthDate' using { d-> LocaleDate.parse(d) }
    })
    static final ObjectMapper personMapper(){}
}
```

When the code compiles, a new implementation of ObjectMapper will be created and installed as the return value for the personMapper() method. The compiled version of the DSL has all of the same functionality of the dynamic version except that it does not support using ObjectMappers directly in the using command; however, a workaround for this is to use a closure to wrap another `ObjectMapper`.

Usage

Once you have created an `ObjectMapper` using either the runtime or compile-time implementations, objects can be copied using the `copy(Object,Object)` method, which will copy the properties of the source object into the destination object according to the configured mapping information in the DSL.

```
def people = individuals.collect { indiv->
    Person person = new Person()
    individualToPerson.copy(indiv, person)
    person
}
```

The `Person` objects will now contain the correctly mapped property values from the `Individual` objects. The `ObjectMapper` also has a `create(Object, Class)` method to help simplify the use case where you are simply creating a new populated instance of the destination object:

```
def people = individuals.collect { indiv->
    individualToPerson.create(indiv, Person)
}
```

When using the `create()` method, the destination object must have a default constructor.

The third, slightly more useful option in this specific collector case is to use the `collector(Class)` method, which again takes the type of the destination object (with a default constructor):

```
def people = individuals.collect( individualToPerson.collector(Person) )
```

The collector(Class) method returns a Closure that is also a shortcut to the conversion code shown previously. It's mostly syntactic sugar, but it is nice and clean to work with.

ResultSet Mappers

A common operation while working with a databases is the mapping of row data to objects in code. This has always seemed very repetitive and error prone. The Vanilla library provides a simple DSL-based solution for quickly implementing code that extracts, and optionally transforms, the data from the database table (via a `ResultSet`) and uses it to populate the object representation of your choice.

There are two implementations available, one that is dynamic and does all the extraction and conversion work at runtime, and a second which pre-compiles the extraction code at compile time using AST transformations.

Mapping Style

There are two styles of mapping available to both implementations of the mapping framework:

- `MappingStyle.IMPLICIT` - All properties of the mapped object are mapped by default, though the DSL may be used to alter property extraction or ignore properties.
- `MappingStyle.EXPLICIT` - No properties of the mapped object are mapped by default. All mapping must be explicitly configured by the DSL.

DSL

The DSL used by the `ResultSetMapper` framework is shared between both the runtime and Compiled Implementations. There are five supported mapping statements:

`ignore <property-name>[,<property-name>…]`
: Used to ignore one or more of the mapped objects properties. This operation is meaningless for explicit mappers.

`map <property-name>`
: Maps the specified object property with the default extraction type (uses `from <field-name>`) and no conversion.

`map <property-name> from[TYPE] <field-name>`
: Maps the specified object property with the specified extraction type and field name with no conversion.

`map <property-name> from[TYPE] <field-name> using <closure>`
: Maps the specified object property with the specified extraction type with the provided conversion closure.

`map <property-name> using <closure>`
: Maps the specified object property with the default extraction type (uses `from <field-name>`) with the provided conversion closure.

Auto-conversions of the property name to a database field name are done as camel-case to underscore-based notation similar to the following:

```
something            --becomes--> something
somethingInteresting --becomes--> something_interesting
anotherPropertyName  --becomes--> another_property_name
```

The extraction types are defined analogous to the getter methods provided in the `ResultSet` but with the "from" prefix in place of "get". The supported extractors are: `from`, `fromObject`, `fromString`, `fromBoolean`, `fromByte`, `fromShort`, `fromInt`, `fromLong`, `fromFloat`, `fromDouble`, `fromBytes`, `fromDate`, `fromTime`, `fromTimestamp`, `fromAsciiStream`, `fromUnicodeStream`, `fromBinaryStream`, `fromCharacterStream`, `fromBigDecimal`, `fromRef`, `fromBlob`, `fromClob`, `fromArray`, `fromURL`, `fromMapper`.

Each extractor takes either the database field name or position value as a required parameter and extracts the database value as you would expect from its name, though you may want to consult the `ResultSet` documentation for more details on the behavior of a specific getter method.

The exceptional extractor case is the `fromMapper` extractor. This extractor accepts a `ResultSetMapper` instance which will be used to extract the object for the specified target property. An example of such a mapping would be the following:

```
map 'foo' fromMapper Foo.mapper('foo_')
```

where `Foo.mapper(String)` is a mapper factory method.

The conversion closures configured by the `using` clause are standard Groovy closures that are passed in one or two arguments:

- 1-argument: the extracted database value at runtime which allows the closure to do additional conversion of the database value.
- 2-argument: the reference to the `ResultSet` being extracted.

The closure should return the desired mapped value for the property.

A sample of the DSL pulled from the unit tests is shown below:

```
ignore 'bankPin', 'pet'
ignore 'children'
map 'birthDate' fromDate 'birth_date'
map 'age' from 2 using { a -> a - 5 }
map 'name' from 'name'
```

Runtime Implementation

The runtime implementation uses the static `mapper(Class, MappingStyle, Closure)` method from `ResultSetMapperBuilder` as its entry point. The object type being mapped is provided, as well as the mapping style and an optional configuration closure (DSL). Using our example code from the DSL section, you would have something like the following for an `IMPLICIT` mapping:

```
ResultSetMapper mapper = ResultSetMapperBuilder.mapper(Person){
    ignore 'bankPin', 'pet'
    ignore 'children'
    map 'birthDate' fromDate 'birth_date'
    map 'age' from 2 using { a -> a - 5 }
    map 'name' from 'name'
}
```

An `IMPLICIT` mapping has a very clean format for cases where your mapped object aligns with the database fields and types. You can simply map it with no configuration DSL.

```
ResultSetMapper mapper = ResultSetMapperBuilder.mapper(Person)
```

An `EXPLICIT` mapping for the same configuration could be something like:

```
ResultSetMapper mapper = ResultSetMapperBuilder.mapper(Person, MappingStyle.EXPLICIT){
    map 'birthDate' fromDate 'birth_date'
    map 'age' from 2 using { a -> a - 5 }
    map 'name' using { n-> "Name: $n"}
}
```

The mapping, extraction and conversion operations are all performed at runtime. Some pre-computing or caching may be performed; however, the bulk of the mapping is done at runtime with each mapped object creation.

The created mappers are reusable and thread-safe.

TIP While the dynamic runtime implementation is fully supported and maintained, it is generally advisable to use the Compiled Implementation to generation more performant code.

Compiled Implementation

The Compiled Implementation uses the `InjectResultSetMapper` annotation to inject a configured `ResultSetMapper` into a field or method of a class.

Using the example from the Runtime Implementation section, we could have an `IMPLICIT` mapper available as a static method as follows:

```
class Mappers {

    @InjectResultSetMapper(
        value=Person,
        config={
            ignore 'bankPin', 'pet'
            ignore 'children'
            map 'birthDate' fromDate 'birth_date'
            map 'age' from 2 using { a -> a - 5 }
            map 'name' from 'name'
        }
    )
    static ResultSetMapper personMapper(){}
}
```

The `personMapper()` method returns the same `ResultSetMapper` instance for every call and the mapper itself is configured at compile-time via AST transformations so that the extraction calls are generated at compile-time rather than for each mapping call; however, the conversion closures are still executed at runtime.

For `IMPLICIT` mappings where the object property names and type align with the database fields, you can have a very simple `IMPLICIT` mapping configuration:

```
class Mappers {

    @InjectResultSetMapper(Person)
    static ResultSetMapper personMapper(){}
}
```

Creation of `EXPLICIT` mappers follows a similar style:

```
class Mappers {

    @InjectResultSetMapper(
        value=Person,
        style=MappingStyle.EXPLICIT,
        config={
            map 'birthDate' fromDate 'birth_date'
            map 'age' from 2 using { a -> a - 5 }
            map 'name' using { n-> "Name: $n"}
        }
    )
    static ResultSetMapper personMapper(){}
}
```

The method or field used to provide the compiled mapper does not need to be static; however, it is advisable, since the underlying instance created will be a static field of the enclosing class.

The generated mapper class is created in the same package as the mapped object type. The `InjectResultSetMapper` annotation also provides a `name` property which may be used to provide an alternate name for the generated mapper class. By default, the name of the mapped object type is used with the added "Mapper" suffix.

The generated mapper class may be used directly; however, the method or field injection is required to create the mapper class and it is recommended to use the field or method as your access point to the generated class.

INFO: When using a factory method to create the `ResultSetMapper` instances, note that the method is annotated with `@Memoized` so that multiple calls to the method will return the same instance. This is also true when a prefix is used (method parameter); calls to the method with the same prefix value will return the same instance of the mapper.

Usage

Once you have created a `ResultSetMapper`, you can use it anywhere you have a `ResultSet` or with the `groovy.sql.Sql` class as follows:

```
def sql = Sql.newInstance(db.url, db.user, db.password, db.driver)

def people = []

sql.eachRow('select * from people'){ rs->
    people << Mappers.personMapper().call(rs)
}
```

Where we are using the compiled mapper implementation exposed by the `Mappers.personMapper()` method. The `call(ResultSet)` method performs the mapping of the current `ResultSet` data into a `Person` object.

An alias to the `call(ResultSet)` method is provided as the `mapRow(ResultSet, int)` method, which allows for simple interaction with the Spring Framework as a `RowMapper`, for example:

```
List<Person> people = jdbcTemplate.query(
    'select * from people',
    Mappers.personMapper() as RowMapper
)
```

The `ResultSetMapper` may be cast as a `RowMapper` and then used as one.

Field Name Prefix

Both implementations of the `ResultSetMapper` support an optional field name prefix, which will be applied to all field names on field lookup. So for example a property named "somethingInteresting" with a prefix of "foo_" would look for a database field named "foo_something_interesting". If no prefix is specified an empty string will be used.

The prefix is *not* overridden by the "fromXXX" mapping methods. This is intentional since the main goal of the prefix support is to allow the same mapper to be useful across different mapping scenarios such as in a join where each column name in the query result has been prefixed by some common known prefix.

This does not remove the ability to use numerical from indices. If a number is detected in the "from" statement, the prefix will not be applied.

Mapping Properties from Multiple Fields

It is often necessary to map an object field that is not directly represented in the database fields or is made up of more than on field. This is can be done by using the 2-argument version of the "using" closure. Consider the example of a `GeoLocation` object being mapped:

```
@Canonical
class GeoLocation {
    double latitude
    double longitude
}

class Somewhere {
    GeoLocation location
}
```

The DSL code would look something like the following:

```
{
    map 'location' fromDouble 'latitude' using { lat,rs->
        new GeoLocation(lat, rs.getDouble('longitude'))
    }
}
```

This allows the flexibility to populate the target object from various database fields.

MockResultSet DSL

Mocking interactions with a database can be frustrating; however, the MockRunner JDBC library can really simplify the mocking of `ResultSet` interaction, which is often enough to get basic unit testing done. Vanilla provides an added layer on top of the MockRunner `MockResultSet` class so that the `ResultSet` may be configured using a simple DSL, such as:

```
ResultSet rs = ResultSetBuilder.resultSet {
    columns 'first_name', 'last_name', 'phone_number', 'age'
    data 'Fred', 'Flintstone', '555-123-9876', 56
    object phoneRecord
    map firstName:'Barney', lastName:'Rubble', age:55, phoneNumber:'555-222-3456'
}
```

The resulting `ResultSet` is implemented by the `com.mockrunner.mock.jdbc.MockResultSet` class from MockRunner and should behave like a real `ResultSet` within the scope of the test mocking.

ResultSet DSL

The `ResultSet` DSL consists of four statements:

`columns`
: Accepts a `String…` or `List<String>` argument to provide the names of the columns configured in the `ResultSet`.

`data`
: Accepts a `String…` or `List<String>` argument to provide the data for a single row, in the same order as the columns.

`object`
: Accepts an Object which will be used to populate the row. The column names will be converted to camel-case and used to find properties on the object.

`map`
: Accepts a Map which will be used to populate the row. The column names will be converted to camel-case and used to find properties in the map.

Overlappable

Determining whether or not two objects overlap is pretty straight-forward when there is only a single variable of comparison; however, what about the case where you have multiple variables to consider, for example if we have a group of people "People A" who are males between 15 and 25, weighing between 200 and 300 pounds:

```
People A
Ages: 15-25
Gender: M
Weight: 200-300
```

Now consider the case when you have other search rules in the same search and that for the sake of efficiency we want to find overlapping rules and merge them into one (we will only consider the overlap determination here). With two other rules:

```
People B
Ages: 13-20
Gender: M
Weight: 250-400

People C
Ages: 8-10
Gender: M
Weight: 200-300
```

It’s not hard to look at the rules and determine which ones overlap. "People A" overlaps with "People B", while "People C" does not overlap with either of the other two. In code this can be a difficult comparison to do correctly and efficiently. Also, as your number of comparison axes increases, so does the complexity of determining overlap for any given pair of objects.

What is needed is a simple means of determining the overlap of two objects and the best way I have found to do that is to break each object down into its overlap-comparison components, each of which I will call a "Lobe" from here on out. I chose the term lobe because it is defined as: "any rounded projection forming part of a larger structure". Also, terms like element and node are used far too much already in programming.

When you break each object down into it’s lobes, you will have one lobe for Gender, one for Ages, and one for weights. Now you can build your overlap determination based on whether or not each lobe overlaps with its corresponding lobe on the other object. If all of the lobes overlap those of the other object, then the two objects are considered overlapping, otherwise they are not. This allows for a fail-fast comparison since if the comparison of any given lobe fails, you cannot have an overlapping object and no further comparison is necessary.

In code a lobe can be defined by interface as:

```
interface Lobe {
    boolean overlaps( Lobe other )
}
```

Each `Lobe` implementation defines what it means to overlap another `Lobe` of the same type. Using Groovy and some generic logic we can easily come up with a `ComparableLobe` which is based on single values and ranges of `Comparable` objects such as numbers, strings and ranges. This allows us to do things like:

```
new ComparableLobe( 10..20, 50, 75 )
new ComparableLobe( 'a', 'h'..'j', 'm' )

lobeA.overlaps( lobeB )
```

which can make the overlap determination very flexible.

```
def genderLobe = new ComparableLobe( 'M' )
def agesLobe = new ComparableLobe( 15..25 )
def weightsLobe = new ComparableLobe( 200..300 )
```

The next thing we need is a way of comparing these lobes in a simple and repeatable manner and that's where the `Overlappable` trait comes in. The `Overlappable` trait defines an object that can be compared for overlap. The required method is basically the same as that of the Lobe; however, this trait is for the parent object itself. By providing an abstract implementation of this interface we have a nice clean way of providing overlap detection functionality for an object type. You could create a simple `Overlappable` Person object:

```
class People implements Overlappable {
    String gender
    IntRange ages
    IntRange weights

    @Override Lobe[] getLobes() {
        [
            new ComparableLobe(gender),
            new ComparableLobe(ages),
            new ComparableLobe(weights)
        ]
    }
}
```

The overlaps() method uses the provided Lobes to populate an `OverlapBuilder`, which is basically a helper class for performing the actual Lobe-to-Lobe comparison of a given set of Lobes. The `OverlapBuilder` is inspired by the builder in the Apache Commons - Lang API, such as `EqualsBuilder` and `HashCodeBuilder`. You create an instance and append your Lobes to it, then execute the `overlap()` method to perform the comparison.

```
new OverlapBuilder()
    .appendLobe(new ComparableLobe(1..10), new ComparableLobe(5..15))
    .overlap()
```

It also provides an append method for simple comparable cases:

```
overlapBuilder.appendComparable( 20..25, 15..30 )
```

which just wraps each value in a `ComparableLobe`. Now, given a list of People objects, you can determine if any of them overlap any of the others simply by iterating over the list and comparing each element with the others:

```
def list = [
    new Person( gender:'M', ages:15..25, weights:200..300 ),
    new Person( gender:'M', ages:13..20, weights:250..400 ),
    new Person( gender:'M', ages:8..10, weights:200..300 )
]

list[0..-2].eachWithIndex { self, idx->
    list[(idx+1)..(-1)].each { other->
        if( self.overlaps( other ) ){
            println "$self overlaps $other"
        }
    }
}
```

As a final little bonus feature, there is a `ComparableLobe.ANY` object which denotes a `Lobe` that will always be considered to overlap, no matter what the other value is.

Test Fixtures

Unit testing with data fixtures is good habit to get into, and having a simple means of creating and managing reusable fixture data makes it much more likely. The `FixtureBuilder` and `Fixture` class can simplify the creation of reusable test fixtures using a small DSL.

The text fixtures created using the `FixtureBuilder` are the properties required to build new instances of objects needed for testing.

The reasoning behind using Maps is that Groovy allows them to be used as constructor arguments for creating objects; therefore, the maps give you a reusable and detached data set for use in creating your test fixture instances. Two objects instances created from the same fixture data will be equivalent at the level of the properties defined by the fixture; however, each can be manipulated without effecting the other.

One thing to note about the fixtures is that the fixture container and the maps that are passed in as individual fixture data are all made immutable via the asImmutable() method; however, if the data inside the fixture is mutable, it still may have the potential for being changed. Be aware of this and take proper precautions when you create an interact with such data types.

Fixture DSL

The `FixtureBuilder` has three operations available to its DSL:

`define`
: The DSL entry point method, which accpets the DSL closure and creates the `Fixture` container.

`fix(Object,Map)`
: Uses the specified Map data as the content for the fixture with the provided key.

`fix(Object,PropertyRandomizer)`
: Uses the specified `PropertyRandomizer` to generate random content for the fixture with the provided key.

The `build()` method is then used to create the populated `Fixture` container object.

Usage

A fixture for a `Person` class, such as:

```
class Person {
    Name name
    LocalDate birthDate
    int score
}
```

could have fixtures created using the following code:

```
class PersonFixtures {

    static final String BOB = 'Bob'
    static final String LARRY = 'Larry'

    static final Fixture FIXTURES = define {
        fix BOB, [ name:new Name('Bob','Q','Public'), birthDate:LocalDate.of(1952,5,14),
score:120 ]
        fix LARRY, [ name:new Name('Larry','G','Larson'), birthDate:LocalDate.of(1970,2,
8), score:100 ]
    }
}
```

I tend to create a main class to contain my fixtures and to also provide the set of supported fixture keys. Notice that the `define` method is where you create the data contained by the fixtures, each mapped with an object key. The key can be any object which may be used as a `Map` key (proper equals and hashCode implementation).

Once your fixtures are defined, you can use them in various ways. You can request the immutable data map for a fixture:

```
Map data = PersonFixtures.FIXTURES.map(PersonFixtures.BOB)
```

You can create an instance of the target object using the data mapped to a specified fixture:

```
Person person = PersonFixtures.FIXTURES.object(Person, PersonFixtures.LARRY)
```

Or, you can request the data or an instance for a fixture while applying additional (or overridden) properties to the fixture data:

```
Map data = PersonFixtures.FIXTURES.map(PersonFixtures.BOB, score:53)
Person person = PersonFixtures.FIXTURES.object(Person, PersonFixtures.LARRY, score:200)
```

You can easily retrieve field property values for each fixture for use in your tests:

```
assert 100 == PersonFixtures.FIXTURES.field('score', PersonFixtures.LARRY)
```

This allows field-by-field comparisons for testing and the ability to use the field values as parameters as needed.

Lastly, you can verify that an object instance contains the expected data that is associated with a fixture:

```
assert PersonFixtures.FIXTURES.verify(person, PersonFixtures.LARRY)
```

which will compare the given object to the specified fixture and return true of all of the properties defined in the fixture match the same properties of the given object. There is also a second version of the method which allows property customizations before comparison.

One step farther… you can combine fixtures with property randomization to make fixture creation even simpler for those cases where you don't care about what the properties are, just that you can get at them reliably.

```
static final Fixture FIXTURES = define {
    fix FIX_A, [ name:randomize(Name).one(), birthDate:LocalDate.of(1952,5,14), score:120
]
    fix FIX_B, randomize(Person){
        typeRandomizers(
            (Name): randomize(Name),
            (LocalDate): { LocalDate.now() }
        )
    }
}
```

The fixture mapper accepts `PropertyRandomizer` instances and will use them to generate the random content once, when the fixture is created and then it will be available unchanged during the testing.