The Inflection Language Specification

John Brush, ZHAW, brsh@zhaw.ch

1. December 2015

Chapter 1

Language

1.1 Lexical Structure

The lexical structure generally follows the Java conventions.

1. White space as described in [1], §3.6.
2. Comments as described in [1], §3.7.
3. Identifiers as described in [1], §3.8.
4. Keywords include all Java keywords (see [1], §3.9) as well as the following keywords exclusive to Inflection: `default`, `taxonomy`, `view`, `use`, `property`, `field`, `include`, `exclude`, `as`.
5. Separators are a subset of [1], §3.11: `( ) { } ; , .`.
6. There is a single operator, the wildcard operator `*`.

1.2 Compilation Units

1. Compilation units are defined in files ending in `.inflect` (TBD: clarify this).
2. Each compilation unit is structured as zero or one `package` declarations, followed by zero or more `import` declarations, followed by zero or more `taxonomy` declarations.
3. The name specified by the `package` declaration must correspond to the file name as described in [1], §7.2 (TBD: clarify this).
4. Inflection also supports unnamed packages as described in [1], §7.4.2.
5. Inflection supports Single-Type-Import and Type-Import-On-Demand declarations as described in [1], §7.5.1 and §7.5.2, respectively.

6. Single-Type-Imports may not point to types in different packages that share a common same simple name.

7. As in Java, the unnamed and the `java.lang` packages are always automatically imported. Additionally the package `ch.liquidmind.inflection` is also always automatically imported. These reserved package prefixes cannot be used for user taxonomies.

8. Compilation units result in zero or more compiled taxonomies, each in its own `.tax` file; these files are analogous to `.class` files in regards to packaging and linking/loading.

1.3 Taxonomies

1. Taxonomy names must follow the rules for identifiers (§1.1, 3).

2. Taxonomy names must be unique within the package of the compilation unit.

3. Taxonomies may declare zero or more *views* (§1.4).

4. For every super class of a class referenced by a view there must be corresponding views in the taxonomy or super taxonomy. If such views are not defined by the user, empty default views are automatically inserted by the compiler[1].

5. Taxonomies may inherit from one or more other taxonomies. The syntax is `extends T1, T2, ... Tn`, where `T1` to `T2` are super-taxonomies. If no specific super-taxonomy is specified then the default is the root taxonomy `ch.liquidmind.inflection.Taxonomy`. The order of taxonomies in the declaration is directly related to the precedence of inheritence, thus: given the features `f1` in `T1` and `f2` in `T2`, where `f1` and `f2` refer to the same thing (i.e., same view or are both default declarations), `f1` takes precedence because `T1` is listed before `T2`. It is illegal to inherit from the same taxonomy more than once in a single taxonomy declaration.

6. Taxonomies may define a default *access method* (`property`|`field`) for members occuring within it. The syntax is `default (property|field);`. Taxonomies that do not declare a default inherit the default of the first super-taxonomy. The default of the root taxonomy is `property`.

7. For each view in a given taxonomy, there must be a set of views in that taxonomy or some super taxonomy that refer to the set of super classes of the viewed class.

8. If a taxonomy is declared `abstract` then it is not possible to create instances of views of that taxonomy. Furthermore, the rule defined by §7 is suspended.

[1] Not yet implemented.

9. It is legal for a taxonomy's fully qualified name to be the same as a known package.

10. It is not legal for a taxonomy's fully qualified name to be the same as a known type (class, interface, enum, basic type, taxonomy).

1.4 Views

1. View declarations specify one or more referenced classes using the syntax `view R1, R2, ... Rn`, where `R1` to `Rn` are either specific class references or class selectors (§1.7).~~:~~

2. Classes refered to in view declarations may generic or non-generic, static or non-static, top-level or member classes and must be public[2]. Local and anonymous classes as well as interfaces are not supported.

3. The sets of classes specified by the references in a given view declaration may overlap; in this case precedence is determined by the ~~order in the declaration~~declaration order. For example, if class `V1` and `V2` occur~~s~~ in `R1` and `R2`, respectively, and refer to the same class, then `V2` ~~from~~ `R2` has precedence. This is particularly relevant for aliases (§~~TBD~~7) and annotations (§TBD).

4. The sets of classes specified by the references in two or more view declarations may overlap; as within a single declaration, precedence is determined by the order. For example, given the code fragment `view V1; view V2;`, where `V1` and `V2` refer to the same class, `V2` has precedence.

5. A single view declaration may result in zero, one or more views in the compiled taxonomy.

6. Views may use either the `include` or `exclude` modifier to specify inclusion in or exclusion from the containing taxonomy. All view delcarations with the include modifier are evaluated before those with the exclude modifier. Excluding views cannot declare annotations (§1.6), aliases (§7) or used classes (§TBD).

7. Views may declare *aliases* such as `view V1 as A1` where `V1` is a class reference and `A1` is a legal identifier. Aliases must be defined as part of an *included* view and must be *unique* within the taxonomy hierarchy. Aliases cannot be referenced within taxonomy declarations but can be used to create view instances.

8. Views may declare *used classes* such as `view V1 use C1` where `V1` is a class reference and `C1` is a specific type reference. Used classes allow for calculated fields (§TBD). Views may specify multiple used classes such as

[2] This is mainly due to the fact that views and classes can never be in the same namespace, since the containing taxonomy is always part of a view's namespace.

`view V1 use C1, C2, ... Cn`. It is illegal to specify the same used class more than once in a single view declaration.

1.5 Members

1.6 Annotations

1.7 Type References

1.7.1 Specific Type References

1. Analogous to Java type references; resolution of unqualified (simple) type names is performed by comparing that name with any imported types or packages:

   (a) An error occurs if the simple name does not match any type.

   (b) Any error occurs of the simple name matches more than one type.

1.7.2 Type Selectors

1. Similar to Java type references, but includes the use of the *wildcard operator* (`*`):

   (a) Imported packages are first resolved into (virtually) imported types by identifying any matching types found amoung the set of known types (i.e., compiled types from the class path or uncompiled types in the (known) source code). These virtual imported types are combined with the non-virtual imported types, and the results are compared one by one with the type selector, resulting in a set of matching types.

   (b) Type selectors may match zero, one or more types.

1.8 View Instances

Chapter 2

Errors and Warnings

2.1 Errors

1. If the file name of the compilation unit does not end with `.inflect` ().

2.2 Warnings

Chapter 3

Tests

1. Test single- and multi-line comments (§1.1, 2).
2. Test identifiers (§1.1, 3).
3. Test reserved keywords (parameterized tests) (§1.1, 4).
4. Test empty compilation unit (should "compile") (§1.2, 2).
5. Look into how javac handles suffixes and package to file name correlations and formulate tests correspondingly (§1.2, 1 & 3).
6. Test unnamed package (§1.2, 4); create taxonomy in unnamed package and reference from a) taxonomy in named package and b) taxonmy in unnamed package. What happens when two taxonomies with the same simple name are in both named and unnamed packages? Which has precedence? Does this cause an error due to ambiguity? What does the Java spec say?
7. Test single-type-import naming conflicts (§1.2, 6).
8. Test resolution of specific type references:
   (a) Resolution of simple names versus fully-qualified names.
   (b) Resolution in conjunction with type imports versus package imports.
   (c) Resolution of compiled (class path) versus uncompiled types (compilation units).
9. Test automatic imports and reserved package names (§1.2, 6). Consider introducing mechanism for taking source code from test classes directly (see `IndentingPrintWriter`).
10. Test taxonomy name uniqueness (§1.3, 2). Taxonomy must be unique across compiled (class path) and uncompiled types (compilation units).
11. Test default views (§1.3, 4).

12. Test empty taxonomy (§1.3, 3).
13. Test taxonomy inheritence (including views and default access method) (§1.3, 5 & 6).

Bibliography

[1] The Java Language Specificiation (Java SE 7 Edition), Gosling et al.,