ChiMu Kernel Frameworks

v0_6 [970722]

Mark L. Fussell

ChiMu Corporation
1220 N. Fair Oaks Ave, #1314
Sunnyvale, CA 94089
Phone: 408 734-9068
Email: info@chimu.com
www.chimu.com

Table of Contents

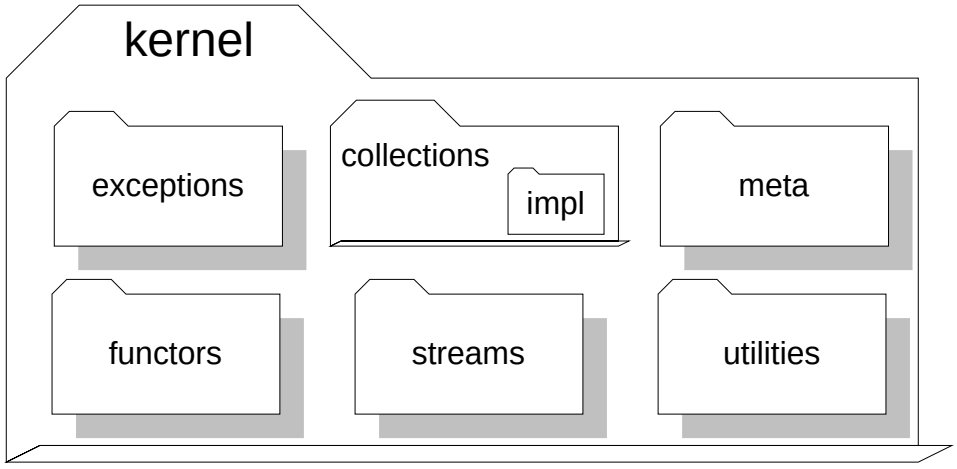
Overview

This document describes ChiMu's Kernel frameworks. These frameworks provide basic functionality that should be available to any part of any application. Although ChiMu considers the Kernel to be a product, it is both open and free. You can do anything you want with the Kernel's design and code as long as you maintain the copyright and warranty notices.

ChiMu's Java Development Standards [Fussell-1] is a prerequisite for understanding some of the naming conventions and notations used in this document.

Introduction

There are many packages within Kernel and they extend the base functionality of Java in very general and useful ways. The sub-packages within Kernel build upon each other and are also expected to be "very public" (highly referenced) throughout other subsystems. The current documentation is incomplete, but it includes some information on Kernel itself (Library and Package), Exceptions, Functors, Streams, Utilities, and Collections.



Kernel

Almost all the functionality in the Kernel Framework is in the sub-frameworks discussed in the subsequent chapters. The Kernel package itself contains just two concepts: Libraries and Packages. These are important to define now because they are useful in discussing the functionality of all Packages. Every ChiMu package has a "Pack" class that documents the functionality of that package.

Library

A Library is a Java "Class" that never has instances, so all functionality is in static members. A Library is really not a class at all, but Java combines object (non-static) and non-object (static) functionality into a single unit of a Class. The Library throws away the object portion of this and only uses the "Class" as a unit of modularity.

Libraries do not have "methods", they have "procedures" and "functions". The distinguishing property is that method calls are bound to methods at runtime but procedures are completely bound at compile time. This is true of static members.

By default, all Libraries are suffixed with Lib to distinguish them from interfaces (no suffix) and normal classes (which are suffixed with "Class").

Package

A Package is an object that understands the functionality of a Java Package (which is not an object). This is useful for consistency (everything "should" be an object), but more importantly it provides an entity that can:

1. Document the functionality of the Package as a whole
2. Provide Package scoped Singleton objects
3. Create Objects of the Types (Interfaces) specified in the package

Of these, probably only the last might be considered strange: Why not just use the Class to create objects of that Class's type? The reasons are:

1. **A Package creation method or function can be more intelligent than a constructor can be.** A Package can reuse an existing object if needed, and can branch on the creation parameters to select the appropriate class to use. Although this could also be done with a Class static method, it does not make as much sense (why branch to a different class) and does not work as well because the static functionality will be inherited by subclasses even when it is not appropriate.
2. **If you expose the Class for creation, you expose the class.** All the public methods are now visible even if some of them are only for intra-subsystem functionality.
3. **This produces a nice separation** of instance behavior (via types) of existing objects and creation behavior (via Packages or other objects) to make those objects.
4. **This also produces a nice cohesion**: all the creation methods of similar types can be placed together or separated (if multiple Packages are used).

Approaches

After deciding to use Package objects there are two approaches to implementing them. Either use a real singleton object that implements the Package interface or create a pseudo-singleton object via a Library (a Pack). Although ChiMu uses primarily the latter approach, we will cover the real object approach first (Package and PackageClasses).

Package

Package defines an interface that all Package objects should be able to support. Each package can add a new subtype to Package if they want to specify new functionality beyond the standard Package functionality. They could also subclass to document the Package although that would more likely be done in either the Pack or the PackageClass.

PackageClass

To implement a Package interface requires a class. PackageAbsClass provides some inheritable functionality to build the standard Package interface and can be extended to provide the rest of the functionality for your specific package. You can also use this class to manage your Singleton Package object, although we normally use a Pack to do that.

Pack (PackageLibrary)

A Pack (short for PackageLibrary) is a simplification and "cheat" for creating a Package object: use a Library to simulate a singleton object. This is far easier to use for most clients. Instead of trying to find the Singleton object ("somebody.soleKernelPack().doSomething()"), clients can directly refer to the Pack class in their code ("KernelPack.doSomething()").

The trade off is flexibility: you can not pass around the "KernelPack" library to other parts of the system for them to work with. You also can't "alias" a Pack so either the Pack name does not collide with other Pack names (used by a particular client) or you have to refer to the Pack class using the full path name.

All Packs should be suffixed with "Pack". This documents that they are a Package Library and prevents name collisions with the normal types and classes within the package.

ChiMu Standard Approach for Packages

ChiMu code standardizes on having Packs in all Packages and only providing real Package objects when it is necessary. Documentation is placed on the Packs.

Examples

Using PackageLibs for documentation is shown in COM.chimu.kernel.KernelPack itself. Using PackageLibs to provide package functionality is shown in COM.chimu.kernel.collections.impl.CollectionsImplPack. Almost all ChiMu classes are invisible outside the package (unless they can be inherited from), so all object constructions are done through "Pack" libraries.

Exceptions

Kernel Exceptions adds a few useful exception subclasses to the standard Java Exception classes. These are currently all development exceptions.

Development Exceptions

DevelopmentExceptions are errors that occur at Runtime but are caused solely by a malformed program. The following table documents some of the current Development exceptions, and gives a feel for what the other ones would be like.

NotImplementYetException	Used to express that some functionality is planned but not implemented
ShouldNotImplementException	A subclass has a more restrictive interface than it "claims" to have (by inheritance or type implementation) and should not (or at least does not) implement a particular method.
FailedRequireException	A method requires (in the Eiffel sense) something from the caller (e.g. non-null parameters or being in a certain state) which was not satisfied

ThrowableWrapper and WrappedExceptions

ThrowableWrappers enable you to throw a new exception that wraps another exception, usually the exception that you are currently handling. Although most clients will only "see" the newly thrown exception, other clients (and the #toString and #printStackTrace methods) can look into what other exception lower in the system caused the current exception. This structure allows you to produce high-level exceptions but still enable a client to see the details of a low level exception.

ThrowableWrapper is the interface for wrapped exceptions. If you want to test whether a given exception may be wrapping another exception, test whether it is an instanceof (implements) ThrowableWrapper. Then you can ask what the wrappedThrowable was (which could be null for an originating exception) and continue the process through the next exception, if any.

WrappedException and RuntimeWrappedException are two implementation classes that can either be used directly or subclassed.

Functors

Functors are objects that model operations that can be performed. In their simplest form they are somewhat like function pointers: they allow a client to call an unknown method with a standard interface. Because functors are full-blown objects they can do quite a bit more than function pointers: they can record and take advantage of their own state as well as allowing references to other objects be used in the method execution. "Smart" functors can also allow the client to ask questions about what operations are available and what the functor requires for proper execution. Basic functors don't provide this functionality but are easier to create which makes them more useful.

Basic Functors

The basic functors are Interfaces with a single-method protocol that makes them easy to implement. They are about as close to having Smalltalk block simplicity as is possible with Java right now. Functors vary on two axes: what type of value they return and how many arguments they take. There are three types of Functors based on return value:

Type	**Returns**	**Method Prefix**
Procedure	nothing (void)	execute
Function	An Object	value
Predicate	A boolean	isTrue

There can be any number of parameters, but currently there is support for only 0, 1, and 2 argument functors. All functors only take Objects as parameters.

#	**Type Suffix**	**Method Suffix**
0	0Arg	() [none]
1	1Arg	With(Object arg1)
2	2Arg	With_with(Object arg1, Object arg2)

Examples

The following are some declarations of the different basic functors.

```
public interface Procedure0Arg {
  public void execute();
}
```

```
public interface Function1Arg {
  public Object valueWith(Object arg1);
}
```

```
public interface Predicate2Arg{
  public boolean isTrueWith_with(Object arg1, Object arg2);
}
```

Calling the above functors might look like this:

```
boolean exampleMethod (Procedure0Arg procedure, Function1Arg function, Predicate2Arg
predicate) {
    procedure.execute();
    return predicate.isTrueWith_with("Hi There", function.valueWith("  Hi  "))
}
```

Using a BasicFunctor

To use a BasicFunctor you can either implement the functor's interface in a standard Class or use Java 1.1's anonymous classes to create functors where you need them.

Named Functor Creation

Using the standard Class implementation looks like this:

```
class MyProcedure implements Procedure0Arg {
   public void execute() {
      System.out.println("We're here");
   }
}
```

```
class MyTrimFunction implements Function1Arg {
   public Object valueWith(Object arg1) {
      return ((String) arg1).trim();
   }
}
```

```
class MyStartsWithPredicate implements Predicate2Arg {
   public boolean isTrueWith_with(Object arg1, Object arg2) {
      return ((String) arg1).startsWith((String) arg2);
   }
}
```

And creating one of these functors would look like this:

```
Procedure0Arg myProcedure1 = new MyProcedure();
```

Anonymous Functor Creation

Using the anonymous class method to define and create the Functors in one statement is a bit more obscure but is also much shorter and more convenient. It looks like this:

```
new Procedure0Arg () {public void execute() {
      System.out.println("We're here");
}};
```

```
new Function1Arg () {public Object valueWith(Object arg1) {
      return ((String) arg1).trim();
}};
```

```
new Predicate2Arg () {public boolean isTrueWith_with(Object arg1, Object arg2) {
      return ((String) arg1).startsWith((String) arg2);
}};
```

Functor recompiler

Because using anonymous classes for Functors is still a bit noisy, we have a recompiler for Java that allows you to use "Blocks" (very similar to Smalltalk blocks) as a more concise functor creation expression. A block's syntax is:

```
'[' FunctorArgumentList '|' BlockStatements ']'
```

where a '^' in front of an argument specification indicates the return type of the "block".

Returning to the above examples, we could call the original “exampleMethod” as follows:

```
if (exampleMethod (
      [^void | System.out.println("We're here");],
      [String arg1 | return arg1.trim();],
      [^boolean, String arg1, String arg2 | return arg1.startsWith(arg2)]
      )
  ) System.out.println("Found the string!");
```

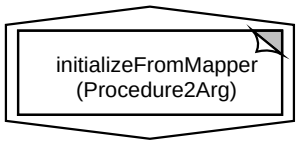
where each block would expand to the above anonymous functor creation statement respectively (except there is some auto casting going on).

Smart Functors

Smart functors have the same basic abilities as basic functors but have a richer protocol that allows the client of the functor to inquire about the functors capabilities and requirements. For example, a SmartFunctor can answer the question of how many arguments it requires and whether it returns a value.

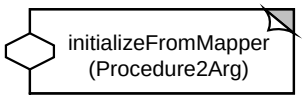
Notation

Since functors are a combination of an object and a function (block of code), the notation for a functor should be a combination of an object and a block of code. Because the function is within the object we get the notation to the right.



The first text line describes the Class (the functionality) of the Functor and the second line describes the parent type (the protocol) of the Functor. This symbol is frequently unwieldy because it is large to begin with and enlarges significantly as text is added.

The second notation to the right is also acceptable. For this notation the objectness of the functor is visible but less dominant in area. This is generally a better symbol because it is more visually distinct and scales much better. The ‘object’ can be moved around along the edge of the ‘function’.



Streams

The package “streams” currently contains only one new concept, Generators. A Generator can generate new values endlessly so it is similar to an enumeration that always return “true” to hasMoreElements. Generators also know their current value and can return collections of next values. Currently Generators are used with the database for generating unique identifiers.

Utilities

The Utilities package currently contains only a Translation library with Functors for doing translation between Java types.

Collections

ChiMu's Kernel Collections provides an interface-based set of collection classes. Collection users are provided with the protocol for interaction with many different (but related) types of collections, and they are only tied to those interface not the specific implementation. Collection implementations only need to conform to the interface and can then choose what are the most important performance and space criteria for them. This also allows many interesting implementations like Futures or Tracers to work within the same collection system.

A question to ask would be "Why does ChiMu have its own set of collection classes?" instead of using all the commercially available ones. The previous paragraph is the main reason: none of the current collection classes are fully interface-based. Without this feature we are getting tied into the specific implementations of the classes and getting locked into a single company's licensed product. An added benefit of ChiMu Corporation's collections is it uses woven parameters and a clean Type hierarchy.

The ChiMu collections have many classes that wrap licensed classes (JDK, JGL, or CAL) or are transformations of public domain classes (Doug Lea's) to conform to the type system. This shows a client is not trapped into a single product by using these interfaces.

Concepts

The main concept for the Collection framework is that clients should only be concerned with the Type hierarchy when specifying functionality. All client-typing of objects (classes, methods, and variables) should be done in terms of the type hierarchy. Choosing among the implementations of the types is only relevant at object creation time and should only need to consider performance and code-base preference.

Another concept within the Collection Framework is that the most commonly used interface should have simple names. Ideally Java would allow multiple names for a single Type/Interface so we could have a Scientific long name (Extensible Indexed Collection) and a more convenient short name (Sequence), but since Java does not allow this we prefer the short name when it is very natural.

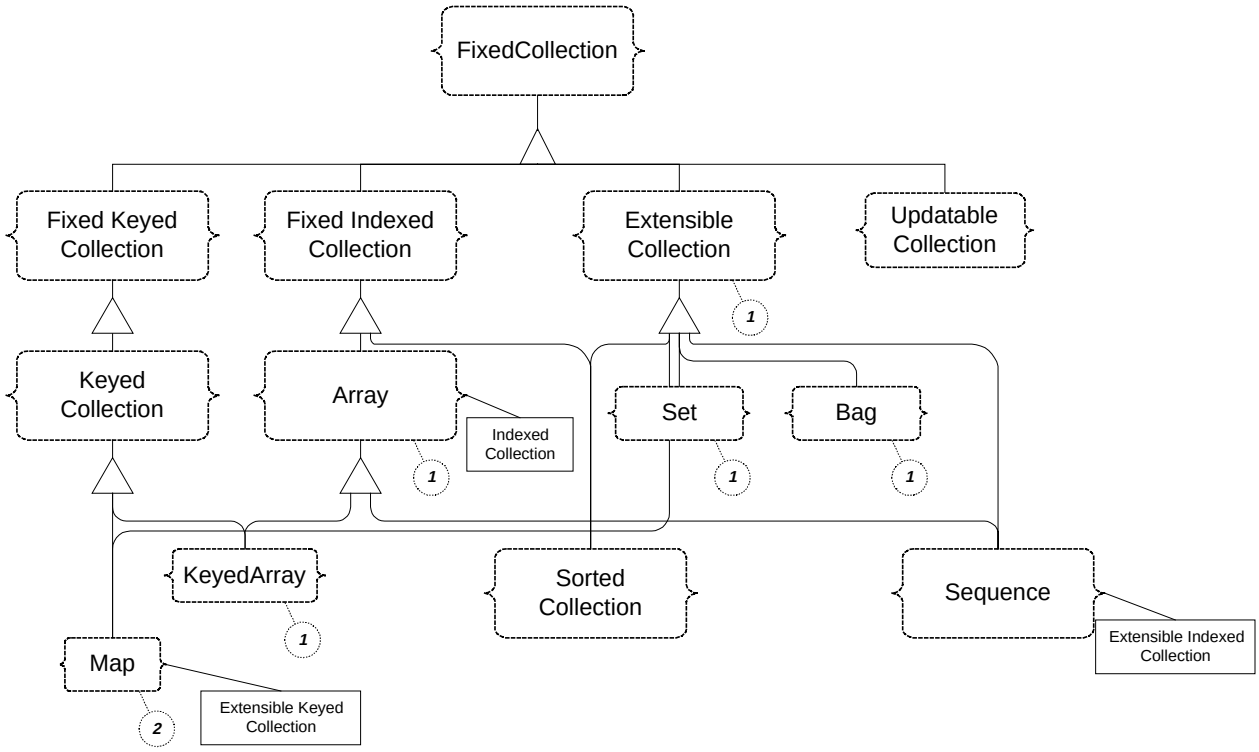
Terms

The Collection framework has precise definition of words which allows collections to be self describing (in the long-name form).

Collection	An object which holds onto a number of other objects
Extensible	Able to add and remove objects
Updateable	Able to replace objects currently in the collection
Fixed	Unable to change in state. A FixedCollection is one that cannot grow in size or replace any of its elements. A client can only ask questions of the collection. In certain contexts the Fixed is not as restrictive and allows replacement but not growth.
Keyed	Having a key (Object) that can return a value
Indexed	Having a sequential index number (zero based) that can return a value
Zero-based	Having an index value of '0' return the first value in an indexed collection. This is one of the many horrible Java concepts that came from 'C' (for which it makes sense) and should be instantly obliterated. But barring that, all the collection classes are Zero based for consistency with (most of) the rest of Java.

Array	*Indexed Collection* (Non Extensible)
Bag	An unordered collection for which an object can be added more than once and the collection will remember each occurance.
KeyedArray	*Indexed and Keyed Collection* (Non Extensible)
Map	*Extensible Keyed Collection.* Can replace, add, or remove keyed entries
Sequence	*Extensible Indexed Collection.* A collection where entries can be added to the end or beginning and all members can be retrieved by index (zero based)
Set	An unordered collection for which an object can only be added once. Subsequent additions of the same object are ignored.

Collection Types



Definitions

Collection	An object which holds onto a number of other objects
Fixed	Unable to change in state. A FixedCollection is one that cannot grow in size or replace any of its elements. A client can only answer questions to the collection. In certain contexts the Fixed is not as restrictive.
Extensible	Able to add and remove objects
Updateable	Able to replace objects currently in the collection
Keyed	Having a key (Object) that can return a value
Indexed	Having a sequential index number (zero based) that can return a value
Zero-based	Having an index value of '0' return the first value in an indexed collection. This is one of the many horrible Java concepts that came from 'C' (for which it makes sense) and should be instantly obliterated. But barring that, all the collection classes are Zero based for consistency with (most of) the rest of Java.
Array	*Indexed Collection* (Non Extensible)
Bag	An unordered collection for which an object can be added more than once and the collection will remember each occurance.
KeyedArray	*Indexed and Keyed Collection* (Non Extensible)
Map	*Extensible Keyed Collection.* Can replace, add, or remove keyed entries
Sequence	*Extensible Indexed Collection.* A collection where entries can be added to the end or beginning and all members can be retrieved by index (zero based)
Set	An unordered collection for which an object can only be added once. Subsequent additions of the same object are ignored.
KeyedCollection	Can replace keyed entries but not add any new ones or remove old ones
FixedKeyedCollection	Can be inquired as to the value of a key
Functor	"An object that models an operation" [Firesmith+E 95]. In the framework's case a basic functor is a type with a single method that performs an operation.
Procedure	A functor that does not return a value
Function	A functor that returns an Object
Predicate	A functor that returns a boolean
Getter	A role of a Function1Arg when it is designed to retrieve a value from an object (the first parameter)
Setter	A role of a Procedure2Arg when it stores a value (the second parameter) into an object (the first parameter)

References

Cook 92 William R. Cook. "Interfaces and Specifications for the Smalltalk-80 Collection Classes" *OOPSLA 92 Proceedings* Association for Computer Machinery, New York, NY, 1992

Coplien+S 95 James Coplien and Douglas Schmidt, Editors. *Pattern Languages of Program Design*. Addison-Wesley, Reading, MA, 1995.

Firesmith+E 95 Donald Firesmith, Edward Eykholt. *Dictionary of Object Technology: The Definitive Desk Reference*. SIGS Books, Inc., New York, NY, 1995.

Fussell 97-1 Mark L. Fussell. "Java Development Standards". http://www.chimu.com/publications/javaStandards/

Gamma+HJV 95 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Object-Oriented Architecture.* Addison-Wesley, Reading, MA, 1995.

Meyer 97 Bertrand Meyer. *Object Oriented Software Construction, 2nd Edition*. Prentice-Hall, Englewood Cliffs, NJ, 1997.

Vlissides+CK 96 John Vlissides, James Coplien, and Norman Kerth, Editors. *Pattern Languages of Program Design 2.* Addison-Wesley, Reading, MA, 1996.