Generic Ownership Types for Java and the Collections Framework

by

Ahmed Aziz Khalifa

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Doctor of Philosophy
in Computer Science.

Victoria University of Wellington
2014

Abstract

Generic programming has turned out very useful in the development of reusable software. With the Java programming language, genericity is not only meant for reusability, but also for type-safety. Java generics constrain a container object (e.g., list, hash table) to store objects of a pre-specified data type. Nevertheless, safe programming with aliasing (multiple pointers in a program may point to the same object) is still a concern in object-oriented programming language research. A pointing object can mutate the state of the aliased object, reflecting the changes to all of the other pointers (aka aliases) thus affecting their behaviour. As programs grow larger and more complex, such changes in behaviour can be undesirable and difficult to detect and reason about. With respect to container objects, the iterator pattern critically violates encapsulation, allowing aliases to the state (and thereof the components) of its container.

Object ownership is one of the well-researched paradigms in the area of alias management. Ownership types support hierarchical object encapsulation rather than the traditional class-level encapsulation. This thesis introduces an extension of Java 6 with support for ownership types as supplementary generic types. That is, Java generics are extended with the ability of carrying ownership information. This extension provides generic ownership support for all of Java; that is, all major language features are addressed so that programs can safely manage and express their aliasing properties. The resulting language is expressive enough to support common programming idioms, with little programming and runtime overhead. We evaluated the programmability of the language by refactoring a major (the most essential) portion of the Java Collections Framework. We also evaluated the performance impact of our refactoring by conducting a small micro-benchmark study to measure the performance time overhead the refactored collections may impose.

Acknowledgments

I wish to express my gratitude to my supervisors Dr. Alex Potanin and Prof. James Noble. To Alex for his tremendous guidance, patience and support throughout this research. Alex has been habitually knowledgeable, alert and enthusiastic. I am exceedingly thankful to him for keeping me financially secure at all times during the period of this research. James, a million thank you is not enough for you. You enlightened the path forward when it seemed headed for a dead-end. I appreciate the lessons I learned from your red ink. No fair person can fail to see your insightful conjecture.

I wish to express my gratitude to A/Prof. Lindsay Groves for looking after my progress when James and Alex were on another continent; for understanding my stress and press during the final write-up.

A big thank you should go to my examiners for their time, effort and feedback. Another big thank you to the Faculty of Graduate Research for awarding me a Victoria PhD Submission Scholarship, and for the free regular workshops that helped me cope and settle with more ease. I would also like to thank Victoria International for the help given to me since the moment I put my foot on Wellington airport.

My wife Salma has been a fruitful source of love and support throughout my studies. My daughters Ola and Rana, thank you for being quiet (at night) while I was working.

My parents have always supported me to pursue my dreams. They have done everything possible to secure me the best way of life. What I owe them is countless and eternal.

Contents

List of Figures

Chapter 1

Introduction

Since the 1940s to the end of the 1980s, software construction had evolved considerably. Since the 1990s to the present, software construction has evolved miraculously. One of the most prominent advances in the last three decades is the evolution of object-oriented technologies. Object-oriented techniques are mainly meant for reusability, or building reusable components, thus less software needs to be written. Nevertheless, building reliable software components is still a challenge.

The value of some constituent object-oriented techniques such as inheritance, polymorphism and dynamic binding was not well-known when the information hiding principle [54, 55] and the encapsulation technique [40, 41] were introduced. Heretofore, encapsulation is the most distinguishing feature of object-orientation [26]. A data abstraction is an object whose state is accessible only through its operations [66]. Encapsulation evolved with the introduction of inheritance to include, along with the notion of data abstraction, the ability to hide references to subobjects from the clients of the inherited object. This evolution is in fact related to the area of alias management.

Reference aliasing is the property that allows multiple objects to refer to a single object which can be changed by any of the referrers. Aliasing is inevitable in object-orientation. Object-oriented languages without aliasing

are inefficient, if not powerless. Yet, aliasing is a major source of complications, and an important source of malfunctioning in object-oriented systems. The misuse of aliasing breaks the encapsulation needed for building reliable software components [31, 39]. Therefore, aliasing must be managed.

Object ownership [21] is one approach to alias management. This approach restricts the interplay between aliasing and the mutable state of an object, so that an incoming reference cannot penetrate from the outside of that object to directly access its internal state. This is done by identifying an *owner* for every individual object in a system, and by allowing modifications only through the object's owner. *Deep ownership* [17, 19] is one of the kinds of object ownership, which lays out the heap into nested constructs. *Generic ownership* [61] is the merging of deep ownership types with parametric polymorphism.

Potanin et al. [61] provide a formal model for combining deep ownership information with Java generics. Potanin et al. also provide an extended version of Java, named *Ownership Generic Java* (OGJ). This prototype addresses the basic features of Java, such as class declaration, method parameterisation and field access, to show how ownership information can be merged into generic declarations and how restrictions can be imposed statically.

In this thesis, we concentrate on Generic Ownership in the context of Java. The popularity of Java and the vivid alias management research induce the research problem of this thesis: What are the potentials and limits of combining deep ownership with Java generics in a real language implementation that addresses all features of Java? The implementation of Java generics does not touch the Java Virtual Machine (JVM) or the class file structure, what are the possibilities of keeping them untouched without compromising the generic ownership encapsulation? Deep ownership is too strict to support constructs like iterators, how could we support iterators without compromising encapsulation? Finally, what are the poten-

tials and limits of our language's usability in terms of programmability and performance?

1.1 Contributions

This thesis makes the following contribution:

- **OGJ$_+$** is an extension of Java 6 with support for Generic Ownership Types. OGJ$_+$ provides ownership support for all features of Java. To provide support for constructs such as iterators, OGJ$_+$ introduces *readonly* references using the wildcard feature of Java generics. OGJ$_+$ utilises wildcards to relax the restrictions imposed by the deep ownership model without compromising the nested structure of the heap.

- **OGJ$_+$ Compliant Collections:** In order to evaluate the applicability of OGJ$_+$'s encapsulation system to a real life code base, we refactored the complete set of general purpose implementations of the JDK 1.6 Collections Framework. In addition to the ten general purpose implementations, we refactored two legacy implementations and one special purpose implementation. All of these implementations are decedents of five abstract data types: `List`, `Queue`, `Deque`, `Map`, and `Set`. These interfaces were refactored along with the abstract classes that provide their skeleton implementations. In total, we have refactored 38 classes and interfaces.

- **A Micro-Benchmark Study:** In order to evaluate the refactored collections in terms of performance, we conducted a micro-benchmark study quantifying the overall impact of emplacing ownership information into the declarations of Java generics.

1.2 Outline

The next chapter provides a background on the problem of aliasing and representation exposure in object-oriented programming. Thereafter, we review some of the proposals that address this problem, showing different forms of alias management. Finally, we focus on object ownership, providing background on the attempts which inspired our work.

Chapters 3 and 4 introduce the OGJ$_+$ extension of Java. Both chapters give a detailed explanation of the various features of OGJ$_+$, with the necessary discussion on how each feature deals with aliasing.

Chapter 5 describes the implementation details of applying generic ownership to the Collections Framework, with the necessary discussion on the programmability of OGJ$_+$.

Chapter 6 quantifies the cost of our encapsulation system, measuring the runtime overhead imposed by the OGJ$_+$ collections on a set of micro-benchmarks implemented specially to target the Java Collections Framework. We discuss the pros and cons of the integration of object ownership into Java generics.

Finally, Chapter 7 concludes with a summary and directions for future work.

Chapter 2

Background

2.1 Object Sharing

Sharing objects is fundamental to object-oriented programming [39, 42]. According to Wegner [66], 'a language is object-based if it supports objects as a language feature'; 'an object-based language is object-oriented if its objects belong to classes and class hierarchies may be incrementally defined by an inheritance mechanism'; 'a class is a template (cookie cutter) from which objects may be created (instantiated) by `create` or `new` operations'; 'objects of the same class have common operations and therefore uniform behaviour'. Declaring a class creates a unique type that can be called class type, user-defined type or reference type. In a mainstream object-oriented language such as Java, variables of class types are references to objects. That is, a variable of a class type holds the memory address of where the actual class instance (object) is stored; and therefore, assignments to variables of class types copy the references only. These assignments are said to have *reference semantics*. Reference semantics make object sharing attainable and make efficient use of the heap. Consider Fig. 2.1, variables `x` and `y` are of the same class type that represents a list of integers. Variable `x` refers to a list whose members are the first four positive integers. The assignment statement `y=x` does not make a copy of `x`,

but makes `y` refer to the same list referred to by `x`. The change in `y` also changed `x`. References `x` and `y` are said to be aliases, of one another, for the same object. An object is aliased whenever it is referred to by more than one other object.

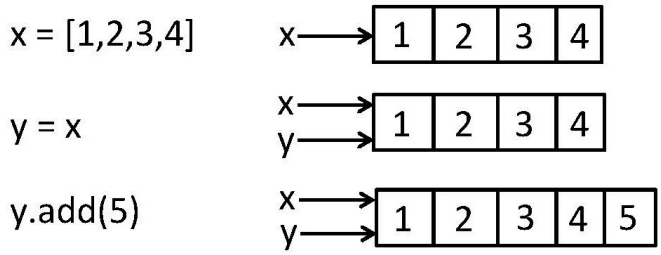

Figure 2.1: Shared List.

Reference semantics are the only possible approach to sharing objects in mainstream object-oriented languages. While sharing is an empowering feature in terms of heap utilization and in the design of data structures, misusing reference aliasing can lead to unexpected program behaviours. Such behaviours can be very difficult to understand and reason about as the system grows.

Consider the example in Fig. 2.2, in line 11 the field `wPos` forms the private state of class `Widget`; `wPos` is of type `Point`; and `Point` is what we call a reference type or class type. That is, the data referred to by `wPos` is always an instance of class `Point`, and represents the position of an individual `Widget` instance (say `w1`). Sharing this data with any other `Widget` instance (say `w2`) will result in dragging `w1` behind `w2` wherever `w2` moves, and vice versa. That is why `wPos` is declared `private`, meaning that the position data of any `Widget` instance is not sharable. Consider the instantiations in lines 22 and 24, if we try to have an assignment statement such as `w1.wPos=pos`, we will get a compile time error saying `wPos` has private access in `Widget`. This is not enough, however. Consider the benign-looking method invocations in line 26, method `getPosition()` returns `wPos`; which means that `w1.getPosition()` is the same as `w1.wPos`,

```
1  class Point{
2      public int x;
3      public int y;
4
5      public Point(int x, int y) {
6          this.x = x;
7          this.y = y;
8      }
9  }
10 class Widget{
11     private Point wPos;
12
13     public void setPosition(Point pos) {
14         this.wPos = pos;
15     }
16     public Point getPosition() {
17         return wPos;
18     }
19 }
20 public class AccessWidget{
21     public static void main(String[] args) {
22         Widget w1= new Widget(), w2 = new Widget();
23
24         Point pos = new Point(50,50);
25         w1.setPosition(pos);
26         w2.setPosition(w1.getPosition());
27         System.out.println(w2.getPosition().x+", "+w2.getPosition().y);
28
29         pos.x = pos.y = 99;
30         w1.setPosition(pos);
31         System.out.println(w2.getPosition().x+", "+w2.getPosition().y);
32     }
33 }
34 ================================================================================
35 run:
36 50, 50
37 99, 99
```

Figure 2.2: Representation Exposure Example.

but Java allows this method invocation. That is, `wPos` is not properly hidden or encapsulated inside `Widget`; and as a result, object `w2` managed to share the position data of `w1`. That is, the method invocations in line 26 are in fact a workaround the disallowed statement `w2.wPos=w1.wPos`, by

means of public getter/setter methods. Java's class-level `private` fields restrict access to the names of the local state, but does not restrict access to the objects to which the names refer. An object's private state or internal representation [52] can still be accessed and modified by its public methods that can directly return references to that private representation.

The principle of *information hiding* [54, 55] was introduced as a design criterion in modular programming. Information hiding means that independent information (`wPos`) should be hidden in independent modules (`Widget`). That is, as long as programmers have as little code dependencies as possible, development risks are diminished. In object-orientation, independent information or the internal representation (aka the private state) should be accessed only through operations (`getPosition()` & `setPosition()`) which are grouped together with the internal representation in a data abstraction (`Widget`). This grouping property in object-oriented languages is what we call *Encapsulation* [6, 26, 39, 40, 41, 42]. Our example followed these principles; the operations of the data abstraction were the only access mechanism to the internal representation. Nevertheless, the encapsulation was easily breached.

Much of the flexibility and efficiency of object-oriented coding is grounded by the notion of sharing mutable objects. Nevertheless, the unrestricted interplay between aliasing and the mutable state can lead to unfavourable outcomes. What happened in the example above is that a supposedly hidden object, `wPos`, was exposed. This phenomenon of *representation exposure* [20, 22, 39, 46] is the side-effect of reference semantics. Representation exposure is the inability of an object to prevent references to objects that make up its internal representation from leaking outside its boundary. That is, representation exposure occurs as long as an object lacks the capacity to constrain aliasing sufficiently in order to enforce encapsulation.

One of the most expressive comments on the situation described in this section came from John Hogg [30]:

> *"Object-oriented languages have a light side and a dark side. The light side is that the programming model makes rapid prototype implementation much easier, since components can be easily reused. The dark side is that as these prototypes mature, the components can manifest strange behaviors due to unforeseen interactions and interrelationships. The big lie of object-oriented programming is that objects provide encapsulation"*.

2.2 Alias Control

Protecting values from being changed due to aliasing has been a research concern since the very early 1990s. In this section, a brief review of literature on the alias management research is provided. Since this thesis work is set in the context of ownership types, we allocate the next section for a general overview of object ownership.

2.2.1 The Geneva Convention

The Geneva Convention on the Treatment of Object Aliasing [31] describes aliasing as a problem for both formal verification and practical programming. It sorts out four approaches to dealing with aliasing: *detection*, *advertisement*, *prevention* and *control*.

Alias detection is subsequent to program implementation. Alias detection is the process of detecting actual and potential aliasing using static and dynamic techniques. The resulting information will classify each object as *never*, *sometimes*, or *always* aliased by any two variables.

Alias advertisement keywords can be used to annotate methods on the basis of their resulting aliasing properties, so that modular analysis can be made more efficiently. Alias advertisement is desirable for the reason that comprehensive detection is impractical.

Alias prevention is the act of verifying statically that aliasing is prohibited within confirmed contexts. This requires conservatively defined static constructs. Eliminating aliasing within particular contexts can help reason about the validity of programs.

Alias control is the act of adopting techniques capable of preventing a system from reaching the level of causing unexpected aliasing. Alias control is the most appreciated approach to dealing with aliasing, since prevention hinders the flexibility of object structures. According to Hogg et al. [31], there are situations in which the dynamic state of the system needs to be taken into consideration in oreder to pin down the bad effects of aliasing. Therefore, alias control needs to be applied at the programming level.

2.2.2 Alias Transitivity and Aggregation

An aggregate is an abstraction for an object that represents a storage destination for a collection of other objects (e.g., a list, table or bag). Noble et al. [52] describe the problem of representation exposure in the context of aggregation. An aggregate object's state is likely to change via an alias to any of the component objects that make up the aggregate's representation, while the aggregate itself is unaware of any aliasing. An aggregate is considered unaware of the aliases of its representation whenever an external object manages to bypass the aggregate's interface operations and directly refers to a state object. Any reference from the outside to the inside of an aggregate can mutate the state of the object's implementation, either by modifying a field or via method calls.

Clarke [16] classifies references, in the context of aggregation, into four categories: *internal*, *external*, *outgoing* and *incoming* references; or into *benign* references (internal and external) and *problematic* references (outgoing and incoming).

Internal references are those created inside the aggregate object, referring to internal objects, and solely to interact internal to the aggregate's implementation. They are safe as long as they are not depending on outgoing references or responsive to incoming aliasing. See references `a`→`b` and `aggregate`→`a` in Fig. 2.3.

External references are those to the aggregate itself, but neither expose nor access the internal implementation of the aggregate. See reference `x`→`aggregate` in Fig. 2.3.

Outgoing references pass from an object inside the aggregate to alias an object external to it. The aggregate has no control over external objects, while the references depend on them and there is no guarantee that the dependency is on the immutable parts of the external object [52]. See reference `a`→`y` in Fig. 2.3.

Incoming references penetrate from the outside to the inside, and are considered very critical since they can directly access the internal state of the aggregate, and thus mutate it, without the aggregate being aware of the aliasing [31]. See reference `z`→`b` in Fig. 2.3.

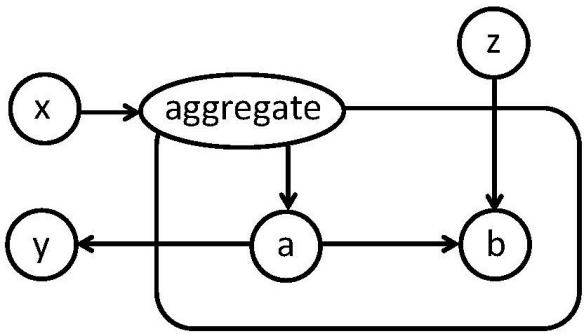

Figure 2.3: Types of references. Object `aggregate` is denoted by the box.

Various attempts have been proposed to prevent the problem of representation exposure, and thus eliminating the leak of references to the internal representation. In the rest of this chapter, we will highlight some of the key attempts.

2.2.3 Uniqueness

An object is *unique* as long as it is being referred to by only one reference. If a language can constrain an object to have only one external reference, then such an object can never be aliased. Several proposals deal with different notions of uniqueness [4, 10, 11, 27, 30, 43, 65].

Class-level annotations [43] are the very basic form of uniqueness, proposed for Eiffel. In this encapsulation system, any instance of the annotated class can be referred to by only one reference. This proposal does not make clear whether uniqueness or non-uniqueness should be preserved over subtyping or not. Wrigstad [67] argues that even if uniqueness is not preserved over subtyping, uniqueness might not become invalidated. Nevertheless, non-uniqueness must be preserved over subtyping, since a non-unique superclass might involve methods that create aliases to `this`; invoking such methods in a unique subclass should invalidate its uniqueness. Eiffel has expanded classes [42]. An expanded type object cannot be referenced by other objects, it can only be copied. The expansion nature of a class is not transmitted via inheritance unless the subclass is declared expanded. Also, the other way around holds; a subclass inherited from a reference (non-expanded) class can be declared expanded. That is, the argument about the preservation of non-uniqueness also applies to the preservation of non-expansion.

Reference-level access control proposals [10, 11, 27, 65] demonstrate more refined and restrictive forms of uniqueness. A variable or field annotated as unique is the only possible reference in the system to a particular object. Limiting the number of references to only one reference to a particular object is simple and powerful, but not practical since unique references cannot, for example, refer to the internally aliased objects of a node-linked model, such as a doubly-linked list or a tree map. In most cases, the components of aggregate objects are internally aliased.

Instance-level uniqueness proposals consider an object as free [30, 52] or virgin [37] if it can be initialised without being given a reference. An object's internal representation can then be set to such a free object, guaranteeing that it will not be captured in a variable. Hogg [30] provides the *free* aliasing mode among other aliasing modes required to characterise an object as part of an island or as a bridge capable of connecting two islands; this is discussed in subsection 2.2.4. Noble et al. [52] provide the *free* aliasing mode among other aliasing modes to maintain the flexibility required for alias-protected aggregates; this is discussed in subsection 2.2.6.

To transfer an instance's unique reference from one place to another, additional language features are required. Almeida [4] uses copy assignments. Hogg [30] proposes an atomic operation, called *destructive read*, to nullify unique variables after returning their values. Boyland [10] statically checks the sharing properties of the variables to nullify them after the transfer if they hold unique references. To avoid destructive reads, Boyland[10] accommodates a `borrowed` qualifier so that a unique reference can be passed to a method only for the span of its operation. A borrowed reference transiently loses its uniqueness, but cannot be returned by the method or captured in fields. The unique reference is still visible during the borrowing, however.

Other language features can be associated with uniqueness. Among other capabilities, including *unique capability*, Boyland et al. [11] provide a *null capability* so that a reference with this capability can only perform instance identity comparisons; such a reference can neither access state variables nor invoke operations. Wrigstad [67] argues that references with null capability to unique objects weaken uniqueness.

2.2.4 Full Alias Protection

Islands [30] and Balloons [4] control possible aliasing amongst the state objects statically. External objects are not permitted to have references to the internal representation. This subsection provides a brief overview of the aim of both proposals. This aim is considered common and termed 'full alias encapsulation' in Noble et al. [52].

The Islands proposal aims to group objects into sets of dependent objects; each set is called an island. The objects that make up an island can access each other without restrictions. Objects in one island cannot access objects in another island. Sharing between two islands can only be done through a *bridge* object. All parameters and results of a bridge object's operations must be *read*, *unique* or *free*. An interface can be annotated as: *read* if it is required to be only readable and not assignable to a field; *unique* if the object is required to have only one reference to it; or *free* if the object is not required to have any references.

In the Balloons proposal, data types are classified into balloon types and non-balloon types. Non-balloon types are for full freedom of sharing. Balloons cannot be accessed via state variables of any external object. Any object pointed to by a balloon is part of the balloon and hence has the same aliasing restrictions as the balloon. All objects inside a balloon have access to each other. Any instance of a balloon can be referred to by one and only one external reference; incoming references are not allowed. Copy assignments are then relied on to pass references from one balloon to another, which is expensive.

The Islands proposal relies mainly on annotations. The Balloons proposal uses a single class annotation to binary classify data types into balloon and non-balloon types; program analysis is then relied on to verify statically the sharing properties of the objects. In both proposals, fully encapsulated aggregate objects cannot share their contents between each other. To minimize restrictions, both proposals make a distinction between static aliasing (local state variables involved) and dynamic aliasing (stack

based variables involved). Islands allow dynamic aliases to the internal representation but they are restricted to be read only; this form of dynamic aliasing does not break encapsulation. Balloons can be Opaque or Transparent. Opaque balloons do not allow any form of dynamic aliasing. Transparent balloons do not enforce any restrictions on dynamic aliasing; that is, dynamic aliases are allowed to the internal representation; this form of dynamic aliasing breaks encapsulation and exposes the internal representation.

2.2.5 Confined Types

The *Confined Types* proposal [64] is another static technique to enforce the protection boundaries required to prevent undesirable aliasing. Confinement is a package-level encapsulation scheme proposed for Java. In this proposal, any instance of a class annotated with the keyword `confined` cannot be referenced outside the package in which this annotated class is declared. The same applies to instances of any subclasses of the confined class.

This proposal also introduces the notion of *anonymous methods* to relax confines to the extent that allows flexible code reuse. Any method annotated with the keyword `anon` will propagate the identity of the current instance only and solely to anonymous methods; no aliases to the current instance can originate from within such a method. The flexibility provided by introducing anonymous methods is that confined classes are allowed to inherit methods from unconfined superclasses. A confined class should only call anonymous methods and non-native methods defined in other confined classes. Anonymous methods are disallowed from fetching other objects unless through variable `this`; but cannot assign `this` to a variable or method argument; and cannot return `this`.

2.2.6 Flexible Alias Protection

Flexible alias protection [52] is another conceptual model for enforcing alias encapsulation and managing the effects of aliasing. Rather than recognizing aliasing as a problem in itself, the visibility of an object's state changes via aliases is the premise on which this model is founded. This model distinguishes between two sets of objects: private unexposed mutable representation objects, and public shareable immutable argument objects. That is, an aggregate's representation objects can be read and write, but should not be visible from the outside of the aggregate; and the aggregate's argument objects can be aliased without restrictions, but the aggregate should not depend on their mutable state. This model introduces aliasing mode declarations to annotate static types, and aliasing mode checking to statically verify the aliasing properties of an object's implementation. Aliasing modes are incorporated into a language's type expressions, resulting in moded type expressions. The proposed modes are:

rep This mode classifies an object as a private mutable representation object that is restricted from being sent out of the object to which it belongs.

arg R This mode classifies an object as a public immutable argument object. R is an optional role tag used to individualize respective roles.

free This mode classifies an object as an unaliased object.

val This mode classifies an object as an instance of a value type. It has the same semantics as arg, but does not require a role.

var R This mode classifies an object as an aliased mutable object. It has the same semantics as regular references in typical object oriented languages.

2.3 Object Ownership

In the light of the Flexible Alias Protection model [52], the pioneer model for ownership types was introduced by Clarke et al. [21]. Any object should be owned by only one owner object, and can only be accessed through that owner object. The set of objects owned by the same owner is named a *context*, since these objects have one sharing context [62]. Ownership information is incorporated into type expressions, akin to the Flexible Alias Protection. That is, ownership types are types annotated with *context declarations* [21, 52, 62]. The ownership relationships between objects can be thought of as a tree. All objects in a system must be brought together into a single *ownership tree* [62]. This way, owned objects can be protected from being accessed directly from the outside of the owner's context. Any object outside a context must go through the owner in order to have access to the other objects inside the context.

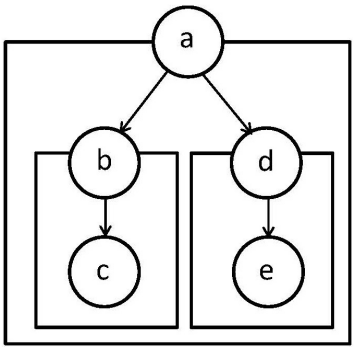

Figure 2.4: Ownership Tree with Sharing Contexts

Consider the ownership tree, depicted in Fig. 2.4, object `a` owns objects `b` and `d`, and hence the three of them form one sharing context. Objects `b` and `c` form another context, and hence `a` and `d` cannot access `c` unless through `b`. Consider the path from `c` to `e`, or from `e` to `c`, objects `c` and `e` can access directly any object in that path as long as they do not penetrate context boundaries. That is, `c` can access directly any object but not `e`, and `e` can access directly any object but not `c`.

The most researched two approaches to object ownership are: *owners-as-dominators* [62] and *owners-as-modifiers* [25].

The owners-as-dominators approach stresses that the dominator object (the owner) is the only entry point for accessing the objects it owns. No one single reference can bypass the dominator. An object *can only be referenced* by references that do pass through its owner. In other words, the path from the root to the private internal representation of any object must pass through the given object.

The owners-as-modifiers approach implies that an object *can only be modified* by references that do pass through the owner of that object, but it can be referenced by any arbitrary object with no restrictions. That is, read-only references are allowed to bypass the owner, but cannot update fields or invoke methods.

In the next two subsections, two object ownership models will be described. The first model, deep ownership [19, 21], adopts the owners-as-dominators approach. The second model, universes [45, 46], adopts the owners-as-modifiers approach.

2.3.1 Deep Ownership

Paradigms that adopt a fully nested scheme of the owners-as-dominators approach are called deep ownership types [19, 21]. The notion of object contexts accommodates the capability to organize the heap into nested constructs. Consider the ownership tree, depicted in Fig. 2.5, we say that the *root context* is `World`. That is, the root context is the set of objects owned by `world`. Objects which are owned by `World` are accessible to all objects. The solid arrows between objects represent the ownership relationships. The boxes represent the *context boundaries*. The dashed arrows represent the references that should stop by the boundaries and pass through the right entry point (i.e., the owner).

Still with Fig. 2.5, the ownership relationships can be seen as a tree rooted at `World`. An owner is another object or `World`. Every object should have an owner. An object can have only one owner. The owner cannot be changed during the lifespan of the object. Object `a` owns `b` but does not own `c`. An object can access the objects it owns. An object can access its ancestors and objects they own. That is, any object can access any object owned by `World`. Henceforth, the notion of *containment* [21] (inside-outside relationships) can be used as we describe the relationships between objects. That is, if we say that `b` and `d` are inside `a`, and `c` is inside `b`, then this means that `b` and `d` are owned by `a` or in `a`'s context, and `c` is owned by `b` or in `b`'s context. Furthermore, we say that `b` and `d` are siblings.

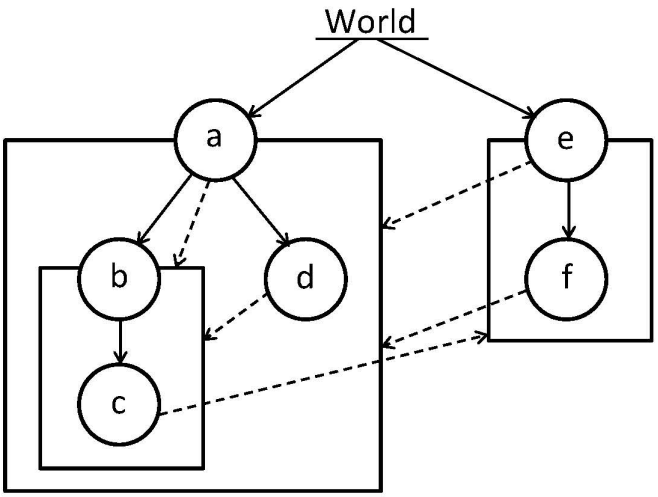

Figure 2.5: Deep Ownership Relationships Between Objects.

The earliest ownership type system [21] introduces ownership types as static types annotated with context declarations. The `rep` annotation denotes an object as owned by `this`; that is, a `rep` object is part of the owner's private representation. The `Owner` annotation denotes an object as having the same owner as `this`; that is, a sibling object to the enclosing `this`. The `World` annotation denotes the absence of an identified owner;

that is, objects owned by `World` can own but cannot be owned; visible to, and accessable by, all objects but not dominated.

This model does not permit a container's iterator to access the container's internal representation. This problem is tackled by permitting inner classes [9] or local variables [17] to have special privileges, or by way of using unique incoming references [19]. The idea is that deep ownership imposes an equal treatment of inner classes and their enclosing classes. That is, inner classes' instances cannot share the internal representation of their enclosing object. Boyapati et al. [9] considered this restriction too strict to support constructs like iterators; and thus, argue that the appropriate way to relax the owners-as-dominators property is to permit inner classes' instances and their enclosing object to have a common representation. A comprehensive study on the use of ownership types in design patterns [50] concludes that permitting an inner class to have privileged access to its enclosing class's internal representation has not shown very useful apart from the iterator pattern; and that even with highly collaborating classes, the use of inner classes is undue in most situations.

Since an iterator needs to be able to access the elements stored in its respective aggregate, the typical approach taken for implementing iterators is to declare them as inner classes. Consider the diagram in Fig.2.6, deep ownership requires an aggregate object to own its private representation. That is, the representation is directly inside the owner context. Since an iterator is usually instantiated independently of its enclosing aggregate, then it should not be able to access the aggregate's representation without the aggregate's knowledge. That is, an iterator should pass through its aggregate's operations in order to access the private representation. The proposed ad hoc relaxation, by Boyapati et al. [9], critically violates the representation containment invariance [21] necessary for deep ownership types, and breaks encapsulation, since incoming aliases are allowed to instances of iterators that share the internal representation of their enclosing aggregates.

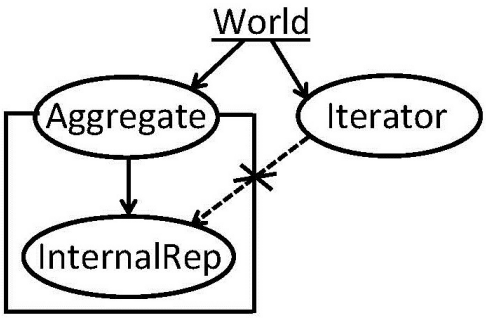

Figure 2.6: References must not cross a context boundary from the outside to the inside

2.3.2 Universes

Müller and Poetzsch-Heffter [45, 46] introduced *Universes* with the following aspirations: Universes needs to (1) have simple semantics, (2) be easy to apply, (3) be statically checkable (4) guarantee an invariant that is strong enough for modular reasoning, and (5) be flexible enough for many useful programming patterns. The Universe type system is an ownership type system to enforce the owner-as-modifier approach that has been inspired by Flexible Alias Protection. The original syntax of Universes, which is different from the current one, used a *type combinator*, higher-order function, to construct types. Later in 2001, Müller and Poetzsch-Heffter [47] used three ownership modifiers `peer`, `rep`, and `readonly` (changed later to be `any`), to extend all type declarations with one of these modifiers. The modifier `peer` denotes an object as having the same owner as `this`; `rep` denotes an object as owned by `this`; and `readonly` denotes an object as capable of having any owner, but `readonly` references cannot update fields or invoke methods. The type combinator function takes two of these modifiers (both arguments can be the same modifier) and returns the resulting ownership modifier to determine the type of field accesses and method call parameters and results. That is, the ownership type of `x.f` is determined, according to the combination to be made inside the type

combinator, by passing the ownership modifiers of both `x` and `f` as reference parameters. The table in Fig. 2.7 is used to determine the resulting modifier. The table uses modifier `this`, which cannot be supplied by a user, as the modifier of receiver `this`.

	readonly	peer	rep
this	readonly	peer	rep
readonly	readonly	readonly	readonly
peer	readonly	peer	readonly
rep	readonly	rep	readonly

Figure 2.7: Universe Types Modifier Combinator.

The Universe type system has achieved gradual development in the past years. The most notable ones: are the integration into the Java Modeling Language (JML) [25], and another type system called Generic Universe Types (GUT) [23] to integrate the owner-as-modifier property into generic types in mainstream object-oriented languages; GUT was also implemented in JML.

The use of read-only references, that can have any owner, makes some programming idioms more expressible with this model than with the deep ownership model. Nevertheless, deep ownership and universes have been evaluated [50] for compatibility with object-oriented design patterns, and have not proved sufficient. The evaluation suggests that many design patterns require moving particular objects, after they have been initialised, from their originating contexts to some other contexts.

2.3.3 External Uniqueness

In conjunction with deep ownership types, a relaxed form of uniqueness called *External Uniqueness* [19] validates the use of incoming references only if they are constrained to be unique.

As explained in subsection 2.2.3, uniqueness is not practical for aggregate objects, since a unique reference cannot refer to the internally aliased components. Clarke and Wrigstad [19] propose that a component object can have only one alias from the outside of its aggregate, while still being aliased freely from the inside of the aggregate. That is, internal aliases to an externally unique object cannot be observed by external clients.

In conjunction with deep ownership, this proposal suggests that an externally unique object can be safely transferred from one context to another sibling context, as depicted in Fig. 2.8. Clarke and Wrigstad argue that this form of ownership transfer is sufficient for concurrent object-oriented programming.

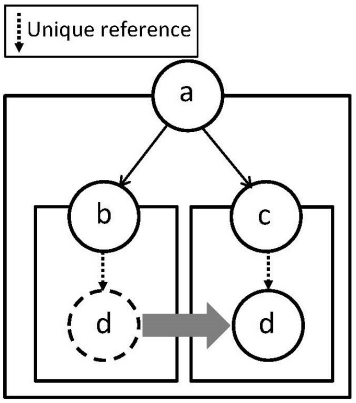

Figure 2.8: Ownership Transfer of an Externally Unique Object

2.3.4 Ownership Generic Java (OGJ)

As a combined mechanism to facilitate deep ownership types in conjunction with parametric polymorphism, Potanin et.al. [59, 60, 61] introduced *Generic Ownership* along with an extended version of Java, named *Ownership Generic Java* (OGJ). This combination provides the ability to constrain aliasing and detect errors statically, while neither demands an added syntactic extension nor imposes overheads at runtime. This is because OGJ

treats ownership information as supplementary generic types. That is, Java Generics are extended by the ability to hold, and interpret, ownership information.

In OGJ, any class declaration needs to allocate at least one type parameter to receive context declarations as type arguments. Usually, this type parameter is bounded by `World`, in which case it is dubbed *owner parameter*. A possible declaration would read as:

```
class Foo<Owner extends World> {...}
```

If an instantiation such as `new Bar<Owner>()` is created within `Foo`'s class body, then `Foo` and `Bar` are siblings owned by the same owner. OGJ uses the last type parameter to denote an object's owner. For example, instantiating a `List` container can read as:

```
List<String,Owner> l = new ArrayList<String,Owner>();
```

OGJ introduced three ownership domains [3], namely `World`, `This`, and `Package`. Objects owned by `World` are global objects that can be freely aliased by any arbitrary object. Objects owned by `This` can only be aliased within the class in which they have been created; they can only be accessed through `this` as a receiver. That is, owner argument `This` denotes an object as owned by an instance of the enclosing class. With regard to the ownership tree which is depicted in Fig. 2.5, there seems no need for more context declarations in order to maintain deep ownership types. We have global objects, private hidden objects, and sibling objects which are owner parameterised by the same owner variable of the enclosing class. In order to make some programming idioms more expressible, the ownership domain `Package` is introduced in OGJ so that objects owned by `Package` can only be aliased local to the package in which they have been created, with no restrictions on how they can be aliased within the package. This follows from the *Confined Types* proposal [64].

2.3.5 Wildcards and Generic Ownership

WOGJ [13] and Jo$\exists$ [12] applies the theory on existential types [56] to generic ownership. Both systems provide theoretical foundations for employing Java wildcards in favour of ownership types. The fundamental goal of both systems is to eliminate the special treatment of the `This` context in parametric ownership type systems, so that the heap structure becomes independent of the encapsulation system, and thus there is no enforcement of the owners-as-dominators property. Jo$\exists_{deep}$ makes this enforcement optional. The above mentioned systems are proved to be theoretically sound, with no practical language design or implementation.

2.3.6 Generic Universe Types (GUT)

As is the case with OGJ, Dietl et.al. [23] combined Universe Types with type genericity. GUT is integrated into JML. GUT enforces the owner-as-modifier approach. As mentioned above, this approach provides for an object to only be modified through its owner, and does not limit the freedom to alias. What applies to the original universe types applies to the generic universe types: the ownership information is object-specific, not class-specific; it provides static control; and annotation overheads.

As is the case with the universe type system, the type combinator function is relied on. The idea is that the three ownership modifiers `peer`, `rep`, and `any` were remodelled to couple type arguments. A possible declaration would be as:

```
rep Iter<any Node<rep ID, any Data>> i;
```

That is, the ownership information is not type parameters, but instead is associated with type arguments as well as with type instances (parameterized and non-parameterized).

Chapter 3

OGJ$_+$ Language

This chapter explains the main concepts of OGJ$_+$, our extension of Java 6 with support for deep ownership types. OGJ$_+$ provides support for generic ownership types, and addresses all of the remaining open issues that were not addressed in OGJ [57, 59, 60, 61]. That is, OGJ$_+$ treats arrays, inner classes, static contexts, wildcard types, clone, equals, enum types and exception handling. We allocate the next chapter to these open issues.

The previous version of OGJ is an extension of Java 5 and provides ownership support for the basic features such as classes and subtyping, field access and assignment, and methods. OGJ$_+$ treats these basic features in a broader perspective, so that interfaces, constructors, method parameters, return types, method invocations and casts can conform to our treatment of the other features. Ideas will be clarified gradually as we progress with this chapter and the next chapter. Altogether, OGJ$_+$ is ownership for all of Java.

OGJ$_+$ provides a very restrictive form of encapsulation. The basis of this is a strict adherence to the rules of the deep ownership encapsulation model which is too strict to support some programming idioms and design patterns. The resulting language is rather verbose. Existing Java programs might require significant restructuring in order to comply with OGJ$_+$.

3.1 An OGJ$_+$ Example

Although the very basic concepts of generic ownership are provided in the previous chapter, we will re-explain them using the example in Fig. 3.1, which is an OGJ$_+$ compliant version of the example in Fig. 2.2.

An OGJ$_+$ program may involve only two ownership domains: `World` and `This`. The first type parameter in a generic parameter list is always reserved to denote an object's owner.

Class instances owned by `World` (see line 24, Fig. 3.1) are public objects, to which access is unrestricted and they are visible to all objects in the ownership tree. That is, `World`-owned objects belong to the root context.

Objects owned by `This` (see line 11) are private objects that need to remain hidden from the other objects in the tree, except from their enclosing objects. That is, `This`-owned objects belong to the current object, and can only be referred to by `this`. The use of owner `This` is exemplified by class `Widget`. `Widget` objects own their `Point` objects as they form the internal representation. This dominance is represented by owner `This` of `Widget`'s field `wPos`, which refers to the widget's position.

Finally, there are objects that need to have the same owner as the `this` object. A referenced object can share the same owner as the current object in order to share the same context. In line 17, the local variable `pos` is owner parameterised by `Owner`, which is the same owner variable used to specify the owner of the enclosing class `Widget` (see line 10). That is, any `pos` object will always be owned by the same owner object as the enclosing `Widget` instance.

The class headers at lines 1, 10 and 21 illustrate how class owners can be defined. Since `Owner` is bounded by `World`, a class can be instantiated as (1) a public object owned by `World` (see line 24); (2) a private object owned by `This` (see line 11), since `This` is a subtype of `World`; or (3) a sibling object owned by `Owner` (see lines 13 & 17).

```
1  class Point<Owner extends World> extends OwnedObject<Owner>{
2      public int x;
3      public int y;
4
5      public Point(int x, int y) {
6          this.x = x;
7          this.y = y;
8      }
9  }
10 class Widget<Owner extends World> extends OwnedObject<Owner>{
11     Point<This> wPos;
12
13     public void setPosition(Point<Owner> pos) {
14         this.wPos = new Point<This>(pos.x, pos.y);
15     }
16     public Point<Owner> getPosition() {
17         Point<Owner> pos = new Point<Owner>(this.wPos.x, this.wPos.y);
18         return pos;
19     }
20 }
21 public class AccessWidget<Owner extends World> extends OwnedObject<Owner>{
22     static { begin(); }
23     public static void main(OwnedArray<World, OwnedString<World>> args){
24         Widget<World> w1= new Widget<World>(), w2= new Widget<World>();
25
26         Point<World> pos = new Point<World>(50,50);
27         w1.setPosition(pos);
28         w2.setPosition(w1.getPosition());
29         System.out.println(w2.getPosition().x+", "+w2.getPosition().y);
30
31         pos.x = pos.y = 99;
32         w1.setPosition(pos);
33         System.out.println(w2.getPosition().x+", "+w2.getPosition().y);
34     }
35 }
36 ================================================================================
37 run:
38 50, 50
39 50, 50
```

Figure 3.1: A generic ownership compliant version of Fig. 2.2.

Methods, whose formal parameter types and/or return types are owned by `This`, can also only be invoked within the current object (i.e., as accessible via `this`) because owner `This` is instance-specific and is not compatible with owner `This` in another class. As a result, we could not have `Point<This>` as a parameter type and return type in `setPosition()` and `getPosition()`, respectively, since invoking these methods in another class, the public class (line 21), is no more possible than through owners other than `This`. Since `Widget`'s field `wPos` is owned by `This`, it is not possible within `setPosition()` to assign the `Owner`-owned parameter `Pos` to `this.wPos`; owners are incompatible. The same applies to `getPosition()`, it is not possible to return `this.wPos`. In this example, the chosen solution is to use value semantics instead of reference semantics; that is, `getPosition()` clones the return value, and `setPosition()` clones the actual parameter's value. Value semantics are not inevitable in OGJ$_+$, as will be discussed later in this chapter.

3.2 Reference Types

There are two kinds of reference types: type variables and nonvariable types. A type variable in OGJ$_+$ needs to have an explicitly owned bound, as illustrated by type variable `T` in Fig. 3.2; the example shows a possible formation of a class header in OGJ$_+$. As illustrated by the example, `T` is bounded by `OwnedObject`, which is the root of the class hierarchy in OGJ$_+$, and is owner parameterised by the owner variable `TOwner`.

```
1 class Foo<Owner extends World, T extends OwnedObject<TOwner>, TOwner extends
      World> extends OwnedObject<Owner> {
2     Foo<This, OwnedString<TOwner>, TOwner> f;
3     ...
4 }
```

Figure 3.2: A possible OGJ$_+$ class header.

A nonvariable type in OGJ$_+$ involves a class name, owner parameter, and optional type arguments. Type `Foo` in line 2, Fig. 3.2, is an example of a nonvariable type; `Foo` involves the type argument `OwnedString` plus two owners, one as the first parameter and the other as the last parameter in the generic parameter list. OGJ$_+$ uses the first type parameter to denote an object's owner. Such an owner is conventionally known as the *distinguished owner*, and is mandatory since every object should be owned. The distinguished owner must be preserved over subtyping, as will be described in subsection 3.3.1. Consider the header of class `Foo`, there are two different owner variables defined, `Owner` and `TOwner`. `Owner` is `Foo`'s distinguished owner, and any owner argument in place of `Owner` will denote the current `Foo` object's owner. As for `TOwner`, this is what we call a *normal owner* parameter. Normal owners can occur, more than once, anywhere in the generic parameter list. In this example, we use `TOwner` to constrain the owner of the generic type argument of `T`.

There are times when owner parameters are not just required to define distinguished owners or to constrain the owners of type parameters, but for example to infer the objects' owners. Consider the example in Fig. 3.3, `Foo1` owns `f2` as represented by owner `This` in line 2. Java will not allow the method invocation, in line 4, if any of the owners `World`, `This`, `Owner2` occurs in place of `MOwner` (line 6). Java does not allow this because the type parameter (or owner parameter in OGJ$_+$'s parlance) should match the owner of `this`, which is `Owner1`. For methods in Java to facil-

```
1 class Foo1<Owner1 extends World> extends OwnedObject<Owner1>{
2     Foo2<This> f2;
3     void meth1(){
4         f2.meth2(this);     }   }
5 class Foo2<Owner2 extends World> extends OwnedObject<Owner2>{
6     <MOwner extends World> void meth2(Foo1<MOwner> f1) {  ...   }
7 }
```

Figure 3.3: Using a naked owner to infer ownership information.

itate type argument inference [28], they had to be generified. That is, the generic parametricity of method `meth2()`, in line 6, allows the method to get the correct owner of `this` as a result of the invocation in line 4. In comparison to normal owners, the owner parameter in this method's signature does not constrain a type parameter's owner. Such an owner is what we call a *naked owner* parameter.

Finally, OGJ$_+$ provides the possibility to define a *placeholder* corresponding to a yet to be defined representation context, through the use of wildcards. The example in Fig. 3.4 shows the header of class `OwnedArray`, which we use in OGJ$_+$ to wrap array objects, as will be explained in subsection 4.1. The formal type parameter `T` is bounded by `OwnedObject`, but the owner (representation context) is not explicitly provided as we use a wildcard in place of the actual owner parameter. As will be explained in subsection 3.6, a wildcard owner parameter must be bounded by `World`. Line 23 in Fig. 3.1 shows how type `OwnedArray` involves the type argument `OwnedString` as being located in the root context.

```
1 public final class OwnedArray<Owner extends World, T extends OwnedObject<?
      extends World>> extends OwnedObject<Owner> implements  java.io.Serializable {
2     ...
3 }
```

Figure 3.4: `OwnedArray` class header.

3.3 Class Hierarchy

In deep ownership, the root of the class hierarchy can be envisioned as a context-specific root. That is, the root of one context is not the same as the root of another, since the heap is transformed into separate nested parts, where `Object` cannot jump arbitrarily from one to another. Strictly speaking, each context should be able to have its own `Object`, which means that `Object` should also be type-parameterised by an owner parameter,

as is the case with every other class. To facilitate this, OGJ$_+$ provides a new class called `OwnedObject<Owner>` to serve as the root class in its respective context; and thus, subtyping `Object` is no longer permitted. Accordingly, a typical OGJ$_+$ class declaration would be as in line 1 of Fig. 3.5.

3.3.1 Subtyping

Context information must be preserved over subtyping [18]. That is, we need to map the owner of the subclass to the owner of the superclass, as appears in lines 1 and 7, Fig. 3.5. This is enforced by the type checker; otherwise, objects can overstep context boundaries using subtyping. Consider the class declaration in line 3, Fig. 3.5, where the owner of `Bar` is not mapped to the owner of `Foo`. As a consequence of this class declaration, Java allows the assignment statement in line 10, where a public object from the root context is able to refer to a private object that is supposed to be hidden in a nested context.

```
1  class Foo<Owner extends World> extends OwnedObject<Owner> { ... }
2
3  class Bar<Owner extends World> extends Foo<World>{ // Owner is not preserved
4                                                     // over subtyping
5      ...
6  }
7  class Break<Owner extends World> extends OwnedObject<Owner>{
8
9      Bar<This> b;
10     Foo<World> f = b;  // Allowed by Java
11 }
```

Figure 3.5: Context information must be preserved.

By preserving ownership over subtyping, the nesting among contexts is preserved and objects cannot overstep the boundary limitations, imposed by the dominators property, using subtyping. Assignment operations should always involve compatible owners. An object's value is

assignable to a reference variable of a supertype of the given object, only if they share the same ownership context. Since each ownership context must have its own class hierarchy, the context information that is propagated amongst the objects contained in one context is not compatible in another context. That is, two instances of the same type, or supertype, are considered incompatible (not assignable to each other) if they have different owners.

3.3.2 Interfaces

Now that extending `Object` is prohibited, and every single object is enforced to be owned and to only subtype owned objects; we still have interfaces: Java expects every interface to only extend another interface. To overcome this, an interface called `IOwnedObject<Owner>` is implemented so that an interface can have its root originated from within its respective context. Similar to class declarations, an interface must preserve its owner over subtyping.

Any class in OGJ$_+$ must be a descendant of `OwnedObject`; and any interface must be a descendant of `IOwnedObject`. By design, `OwnedObject` **`implements`** `IOwnedObject`, and preserves its own owner over this subtyping. Since the owner of a class must be preserved over subtyping, interfaces are safe from being inherited (using **`implements`**) with a different owner than that of the given class.

3.4 Nesting Scheme

As explained in chapter 2, deep ownership is a fully nested scheme of the owners-as-dominators approach to encapsulation. The owner nesting type checking can be concisely explained as going over every individual owner parameter involved in any generic parameter list to verify if it is the same or outside the distinguished owner of the defining object. For domain parameters, this means that `This` is inside everything, and `World` is outside everything. That is, if the distinguished owner is `This`, then there will be no restrictions placed on the other owner parameters involved; if the distinguished owner is `World`, then all of the other owner parameters should be `World`. For owner variables, the relationships between them will be clarified as we describe type checking against variable definitions in subsection 3.4.2. Owner nesting needs to be verified for every class, method and field/variable declaration. The rest of this subsection describes how the inside-outside relationships can be maintained, and what happens if these relationships are broken.

3.4.1 Class Declaration

The determination of nesting relationships amongst the owner parameters in a class header can be seen as if it is amongst the bounds of these owner parameters; and if the owner bound is itself an owner variable, then the compiler will go find its bound, and so on. As in most cases, if the distinguished owner bound is `World`, then every other bound should be `World`.

3.4.2 Field/Variable Declaration

All kinds of variables (namely fields, local variables and parameters) are treated the same, when it comes to owner nesting verification within the types of the variables. OGJ$_+$ assumes that `This` is inside everything, `World` is outside everything, and the distinguished owner variable of the

enclosing class is inside every other owner variable involved in the header of that enclosing class. This means that the field declaration in line 4 of Fig. 3.6 is illegal, since `Owner1` is inside `Owner2`. If this is not illegal, then the instantiation in line 13 will result in `InnerFoo<World, This> f;` which breaks the nesting rule, and exposes the field to the outside world, while the defining object (i.e., the current `Foo`) is supposed to be hidden, and be able to hide its interior as need be.

```
1  class Foo<Owner1 extends World, Owner2 extends World>
2      extends OwnedObject<Owner1>{
3
4      InnerFoo<Owner2,Owner1> f; //Illegal
5
6      public <OM extends World> void method(OwnedObject<OM> om) {
7          ...
8      }
9
10     class InnerFoo<O extends World, O1 extends World>
11         extends OwnedObject<O>{
12
13         Foo<This,World> t = new Foo<This,World>();
14         Foo<O,O1> t1 = new Foo<O,O1>();
15         Foo<World,World> t2 = new Foo<World,World>();
16
17         public void m() {
18             t1.<This>method(t); // ERROR t1's owner(O) is
19                                 // outside t's owner (This)
20             t2.<O>method(t1);   // ERROR t2's owner(World) is
21                                 // outside t1's owner (O)
22         }
23     }
24 }
```

Figure 3.6: Illegal Owner Nesting.

3.4.3 Method Declaration

In certain cases, such as the one described in subsection 3.2, a method declaration might need to be owner parameterised with a supplementary

naked owner parameter to infer the correct owner when the method is being invoked in different ownership contexts. In such a case, this owner parameter will need to be checked for owner nesting against the distinguished owner of the enclosing class. Correspondingly, when it comes to method invocation, the owner of the method receiver must be inside every actual owner parameter used as a distinguished owner for any of the method's arguments. Consider the erroneous method invocations in lines 18 and 20 of Fig. 3.6. The relationship between the objects involved in the first method invocation is depicted in Fig. 3.7. The owner parameter `O` is declared, in line 10, as the owner of `InnerFoo`. The method receiver `t1` is declared, in line 14, as a sibling to `InnerFoo`; which means that `InnerFoo` and `t1` are located in `O`'s ownership context. The method argument `t` is declared, in line 13, as owned by `InnerFoo`; which means that `t` is located in `InnerFoo`'s nesting context. Since `t1` is located outside `InnerFoo`'s context, `t1` in not allowed to cross `InnerFoo`'s context boundary and access `t`. That is why we have to ensure that the owner of the receiver is inside the owner of the argument. The same idea applies to the method invocation in line 20; `t2` is owned by `World` which is outside everything; that is, the owner of the argument cannot be anything other than `World`.

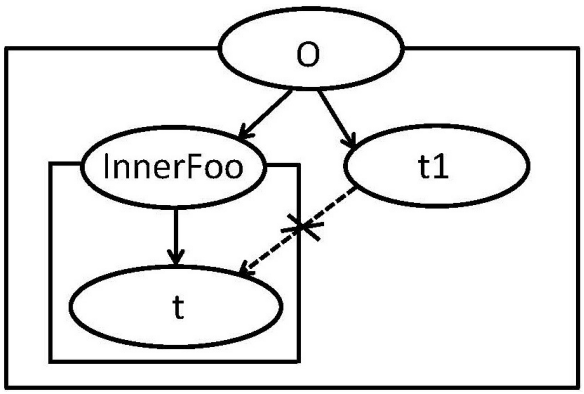

Figure 3.7: Context boundaries should not be broken through method invocations.

3.5 Instantiation and Casting

Ownership information should not be lost while instantiating a new object or due to casting to a particular type. Type erasure is the process by which the Java compiler removes all type parametricity information, parameters as well as arguments, by replacing all type variables with only the class names of their bounds, or with `Object` if a type variable is not bounded. On that account, programmers are allowed to drop the actual type parameters from parameterised types as need be. A parameterised type used without an accompanying type argument is conventionally known a *raw type*. For OGJ$_+$ to ensure that ownership information cannot be lost, raw types are forbidden in all respects. Consider the different kinds of erroneous statements in Fig. 3.8. In addition to the prevention of casting to raw types, and `Object`, casts such as the ones shown in lines 9 and 10 should not be allowed as they create a sharable reference to a private object, in the first case; and a public reference to a within-context sharable object, in the second case. That is, cast ownership information should be the same as the instance ownership information.

```
1  class Foo<Owner extends World> extends OwnedObject<Owner> {
2
3      Foo<Owner> f = new Foo<Owner>();
4      Foo<This> ff = new Foo<This>();
5      Foo<Owner> f0 = new Foo();          //ERROR: f0 initialized ownerless
6      Foo f1 = f;                         //ERROR: Raw Type
7      Object o1 = f;                      //ERROR: Ownership information erased
8      Foo<This> f2 = (Foo)f;              //ERROR: Cast to raw type
9      Foo<Owner> f3 = (Foo<Owner>)ff;     //ERROR: ff loses privacy
10     Foo<World> f4 = (Foo<World>)f;      //ERROR: f loses context information
11 }
```

Figure 3.8: Preserving Ownership Information

Upcasting happens implicitly whenever a reference to a subtype is assigned to a supertype reference, and the supertype might not involve the type parameters that its subtypes might involve, as illustrated in lines 3

and 9, Fig . 3.9. Downcasting the supertype reference as in line 10 is allowed by Java, but will result in `f2` referring to the same object as `f1` with different ownership information, since the actual type parameter `Foo1` of `f2` has different ownership information than that of `f1`'s `Foo1`. OGJ$_+$ prohibits the kind of downcast in line 10, but unfortunately prohibits also the downcast in line 11, since it does not provide a mechanism for preserving the ownership information of runtime downcasts [8]. Nevertheless, OGJ$_+$ mitigates this restriction through the use of wildcard owner parameters, as illustrated in lines 6 and 12. The safety of utilising the wildcard feature of Java generics will be described in detail in the next section.

```
1  public class Foo1<Owner extends World> extends OwnedObject<Owner> {
2
3     Foo2<This, Foo1<Owner>> f1 = new Foo2<This, Foo1<Owner>>();
4     Foo2<This, Foo1<World>> f2 = new Foo2<This, Foo1<World>>();
5
6     Foo2<This, ? extends Foo1<? extends World>> f3 = new Foo2<This, Foo1<This>>();
7
8     public void meth() {
9         OwnedObject<This> o1 = f1;
10        f2 = (Foo2<This, Foo1<World>>) o1;              // ERROR
11        f1 = (Foo2<This, Foo1<Owner>>) o1;              // ERROR
12        f3 = (Foo2<This, Foo1<? extends World>>) o1;    // OK
13    }
14 }
15
16 class Foo2<Owner extends World, T extends OwnedObject<? extends World>>
17    extends OwnedObject<Owner> { ... }
```

Figure 3.9: Preserving Ownership Information against Up/DownCasting

The idea of using wildcards to synthesise owner parameters [15], when downcasting from one type to another, while the latter has ownership information that the former has not, takes its root from "Existential Downcasting" proposed by Wrigstad and Clarke [68]. The authors simplify the idea as *"if a Java-style downcast (disregarding ownership) of an object to some class c succeeds, then we could infer the owner parameters necessary to form the new type from c's class header. We call the inferred owners existential owners, and*

types that use them existential types". Wrigstad and Clarke built their case based on the Java equals idiom, and argue that it is not sufficient for the overriding `equals()` operations to keep track of the owners at run-time by extending each class by a field for each owner the class uses [8]. The equals idiom will be described in section 4.5.

3.6 Wildcard Types

Generic types should be invariant in sound type systems; therefore, Java does not accommodate a built-in conversion from, for example, `List<Manager>` to `List<Employee>` based on the datum that `Manager` is a subtype of `Employee`. That is, there should be no dependency, or subtype covariance, between two objects generated from a single generic class; otherwise, employees who are not managers might be added to `List<Manager>`. As a consequence, it is not possible to write a reusable subroutine that can list the different categories of employees by parameterising the subroutine with `List<Employee>`, or even `List<Object>`, because Java generics are not associated with a built-in subtype covariance. Nevertheless, to attain the flexibility required for writing reusable software, Java facilitates a programmer-defined covariant subtyping through *wildcard types* [28].

A wildcard type is a generic type that uses `?` as a type argument, for example `List<?>`. Wildcard types are carefully integrated with Java generics to provide as safe as possible form of covariant subtyping. For example, Java does not allow any add operations to a `list` object of type `List<?>`, since the element type of the `list` object is unknown, and thus the creation of new element objects is disallowed. In contrast, retrieve operations can face less restriction in making use of the result, since a result type is, in the end, of type `Object`. Nevertheless, a wildcard type can be further constrained as `List<? extends Employee>`, which means that the unknown element type must be a subtype of

Employee; and we say that Employee is the *upper bound* of the wildcard. An unbounded wildcard type, such as List<?>, has the same interpretation as List<? extends Object>. It is also possible to define a *lower bound* for the wildcard using the keyword super; for example, List<? super Trainee>, which means that the unknown element type must be a supertype of Trainee. Furthermore, the bounding types can be parameterised types; and here is exactly how OGJ$_+$ can support wildcard types, since all reference types must be owner parameterised in OGJ$_+$.

OGJ$_+$ provides the same programmer-defined covariant subtyping mechanism as Java. While Collection<?> is the supertype of all kinds of Java collections, Collection<O,? extends OwnedObject<EO>,EO> is a possible supertype of all kinds of collections located in the same ownership context, where O is the Collection's distinguished owner, and EO is the owner of the unknown element type. The use of unbounded wildcard types in OGJ$_+$ is not allowed, since Object is no longer the root class; a wildcard should be bounded by 'OwnedObject', 'IOwnedObject', or a class that is subtyping any of them.

3.6.1 Ownership Context Covariance

OGJ$_+$ also supports ownership context covariance. That is, an object can belong to an unknown ownership context for more flexible code reuse. A wildcard can occur in place of an actual owner parameter, but can only be bounded by World. Accordingly, the supertype of all kinds of collections in a yet to be defined context is as follows:

```
Collection<? extends World,
           ? extends OwnedObject<? extends World>>
```

If a distinguished owner is a wildcard, then any other owner argument must be either a wildcard or World. This is because if we have, for example, a field, local variable, or method parameter of type

`List<? extends World, ...>`, we do not know which owner would replace the wildcard. The owner might be `World`, which means that every other owner must be `World` in order to preserve the nesting relationships between owners—the distinguished owner must be inside every other owner, and `World` is outside everything. So, having a wildcard as the distinguished owner necessitates that all of the other actual owner parameters be made nesting `World` via a wildcard, or `World` as infrequently need be. Declaring an object, whose type is a wildcard owner parameterised, is subject to owner nesting verification by treating every wildcard as `World`. This way, it is possible to have a non-wildcard distinguished owner (e.g., `World`, `This`, or an owner variable), while wildcards can be used in place of other owner arguments. Since every declared object (with wildcards or without) is subject to owner nesting verification, the nesting scheme is not to be violated. The example in Fig. 3.10 illustrates the use of wildcard types in OGJ$_+$. Method `printList()` is able to access `list` objects of unknown contexts. `list01` and `list02` (declared in lines 16 and 21, respectively) are objects located in different contexts. Passing `list01` and `list02` to `printList()` (lines 26 and 27) is safe since the type checking required for owner nesting verification is applied beforehand at the declaration sites.

```
1  public class TestCovariance<Owner extends World, O extends World>
2      extends OwnedObject<Owner> {
3
4      void printList( List<? extends World,
5                           ? extends OwnedObject<? extends World>> list ) {
6
7          Iterator<? extends World,
8                   ? extends OwnedObject<? extends World>> i = list.iterator();
9
10         for (int j = 0; j < list.size(); j++) {
11             System.out.print(i.next()+" ");
12         }
13     }
14
15     void meth(){
16         List<Owner, OwnedString<O>> list01 =
17                 new ArrayList<Owner, OwnedString<O>>();
18         list01.add(new OwnedString<O>("Abc"));
19         list01.add(new OwnedString<O>("Bcd"));
20
21         List<This, OwnedInteger<Owner>> list02 =
22                 new ArrayList<This, OwnedInteger<Owner>>();
23         list02.add(new OwnedInteger<Owner>(123));
24         list02.add(new OwnedInteger<Owner>(234));
25
26         printList(list01);
27         printList(list02);
28     }
29
30     static { begin(); }
31     public static void main(OwnedArray<World, OwnedString<World>> args) {
32
33         TestCovariance<World, World> t = new TestCovariance<World, World>();
34         t.meth();
35     }
36 }
37 ==========================================================================
38 run:
39 Abc Bcd 123 234
```

Figure 3.10: Generic Ownership Wildcard Types

As is clarified in previous subsections, the meaning of owner `This` relies on the presence of receiver `this`. Accordingly, invoking methods, which involve owner `This` within their formal parameter types and/or return types, must be via receiver `this`. Nevertheless, a method might involve owner `This` at the time of invocation, not at the time of declaration, if it is an owner parameterised method (line 2, Fig. 3.11). In such a case, OGJ$_+$ will not check if owner `This` is involved within the declaration signature or not, or if receiver `this` is involved in the invocation or not, but owner nesting verification will be applied to ensure that the distinguished owner of the receiver is inside every actual owner parameter used as a distinguished owner for any of the method's arguments. So, what if the receiver's owner is a wildcard (lines 12, 13 and 14, Fig. 3.11), or if the method argument's owner is a wildcard (lines 16, 17 and 18)? The only legal method invocation, if the receiver's owner is a wildcard, is that if the method argument's owner is `World`; and the only legal method invocation, if the method argument's owner is a wildcard, is that if the receiver's owner is `This`. Otherwise, there is no guarantee that the unknown owner will preserve the nesting relationships.

3.6.2 Readonly References

A wildcard owner makes a reference *readonly*, and hence cannot be manipulated by arbitrary external objects. Consider the example in Fig. 3.12, class `OwnsFoo` owns `f1` and has a getter method `getFoo()` that returns `f1`. The return type of `getFoo()` is owner parameterised by a wildcard, hence the method can be invoked as in line 14. The other option for this method is to have a return type that is owned by `This`, in which case the method invocation in line 14 will not be possible in OGJ$_+$. Class `TryExpose` has the wildcard-owned reference `f2` that can be set to any `Foo` object of any ownership context. The only way, in Java, to write to `f2.bar` (line 17) is if `OwnedString` is raw, which cannot be the case

```
1  class Foo<Owner extends World> extends OwnedObject<Owner>{
2      public <OwnerM extends World> void meth(OwnedObject<OwnerM> o) { ... }
3  }
4  class Test<Owner extends World> extends OwnedObject<Owner> {
5      Foo<? extends World> fWild = new Foo<This>();
6
7      Foo<This>  fThis  = new Foo<This>();
8      Foo<Owner> fOwner = new Foo<Owner>();
9      Foo<World> fWorld = new Foo<World>();
10
11     public void test03() {
12         fWild.meth(fThis);    // ERROR an unknown might not be 'This'
13         fWild.meth(fOwner);   // ERROR an unknown might not be inside 'Owner'
14         fWild.meth(fWorld);   // OK an unknown is always inside World
15
16         fThis.meth(fWild);    // OK 'This' inside everything
17         fOwner.meth(fWild);   // ERROR 'Owner' cannot be inside an unknown
18         fWorld.meth(fWild);   // ERROR 'World' cannot be inside an unknown
19     }   }
```

Figure 3.11: Restrictions on method invocations via wildcard owner parameterised receivers

```
1  class Foo<Owner extends World> extends OwnedObject<Owner> {
2      OwnedString<Owner> bar;
3  }
4  class OwnsFoo<Owner extends World> extends OwnedObject<Owner> {
5      Foo<This> f1 = new Foo<This>();
6
7      Foo<? extends World> getFoo() {
8          return f1;
9      }
10 }
11 class TryExpose<Owner extends World> extends OwnedObject<Owner>
12 {
13     OwnsFoo<Owner> of = new OwnsFoo<Owner>();
14     Foo<? extends World> f2 = of.getFoo();
15
16     public void meth(){
17         f2.bar = new OwnedString("No Raw Types .. No Writing");
18     }
19 }
```

Figure 3.12: Wildcard owned references are readonly

with OGJ$_+$; and that is why we worry about the ownership of a final immutable object such as `String`, since all objects in OGJ$_+$ must belong to an ownership context. Classes in the Java standard library (e.g., `Integer`, `Number`, `String`, etc.) have their corresponding classes in OGJ$_+$ (e.g., `OwnedInteger`, `OwnedNumber`, `OwnedString`, etc.)

Chapter 4

From Java To OGJ$_+$

4.1 Arrays

One of the fundamental issues that any attempt to extend Java with ownership types encounters is arrays. In relation to the use of owners-as-dominators, prototype implementations of ownership type systems in the context of Java are usually bound to remain JVM-compatible; very much as is the case with Java generics, whose information is erased by the compiler after the type checking. SafeJava [7] could not support safe runtime downcasts to array types, since there is no way, while working on the language level, to provide an array object with an added owner field that can be accepted by the JVM at runtime. This is also the case with Cameron & Noble [15] in treating array objects just like primitive types that are not required to be owned, while providing only the array elements with generic owners, whose presence does not violate the JVM.

In relation to the use of owners-as-modifiers, Universe type system (UTS) [25] handles array objects by providing two ownership modifiers, one for the array itself and the other for the array elements. UTS integrates ownership information to Java Modeling Language (JML) [34, 35], which facilitates runtime assertion checking or verification. Assertion checking ensures that an object meets an assertion during program execution. That

is, an object can be assured to hold certain ownership information at runtime.

The fact is that Java arrays are objects and therefore need to be owned. Generic ownership is a mechanism for treating ownership information as supplementary generic types, but there is no generic array creation in Java. Java generics are a language-level mechanism, while Java arrays are checked at the JVM-level. Accordingly, OGJ$_+$ forbids the use of traditional Java arrays. We created the `final` wrapper object `OwnedArray`, whose state is represented by a `private` one-dimensional array object. `OwnedArray` bundles operations that can: return an element at a specified position, replace an element at a specified position, return the number of elements, resize the capacity, and copy a segment from the current `OwnedArray` object to another destination object. `OwnedArray` can be used within an OGJ$_+$ program in the way illustrated by the example in Fig. 4.1

4.1.1 The `main()` Method

Since the commonly practiced Java array types are no longer available in OGJ$_+$, the `main()` method is therefore adapted to capture the command line arguments in an `OwnedArray` object. The code in Fig. 4.1, as well as the code in Fig. 3.1, shows a runnable OGJ$_+$ program. The static initializer in line 18 is mandatory as it invokes the operation `begin()`, which is responsible for translating the new `main()` method signature in line 19. As illustrated, the `OwnedArray` object used takes an `OwnedString` object as a type argument, and the actual owner parameters are all `World`, since the main method is a static context; this is enforced by OGJ$_+$ as will be explained in subsection 4.3. Nevertheless, as the logic goes, the `main()` method is an entry point for running a program whose ownership hierarchy should be rooted at `World`.

```
1  public class ArrEx<Owner extends World> extends OwnedObject<Owner> {
2
3      OwnedArray<This, OwnedInteger<Owner>> arr;
4
5      void meth() {
6
7          arr = new OwnedArray<This, OwnedInteger<Owner>>(10);
8          OwnedInteger<Owner> i;
9
10         for (int j = 0; j < arr.length(); j++) {
11             i = new OwnedInteger<Owner>(j);
12             arr.set(j, i);
13         }
14         for (int j = 0; j < arr.length(); j++)
15             System.out.print( arr.get(j).value + " " );
16     }
17
18     static { begin(); }
19     public static void main(OwnedArray<World, OwnedString<World>> args) {
20
21         ArrEx<World> arr = new ArrEx<World>();
22         arr.meth();
23     }
24 }
25 ================================================================================
26 run:
27 0 1 2 3 4 5 6 7 8 9
```

Figure 4.1: The Use of `OwnedArray`

4.2 Inner Classes

Inner classes are non-static classes which are declared within other classes. Nevertheless, this nesting is only a relationship between classes, not objects. Java facilitates such nesting so that an inner class's instance can maintain an implicit pointer to the object of its enclosing class. This way, the state of the enclosing class is always available to its inner classes; although, as instances, each has its own independent identity.

As explained in subsection 2.3.1, the owners-as-dominators approach to encapsulation accommodates an equal treatment of inner classes and their enclosing classes, since inner classes are usually instantiated independently and therefore references from them to the outer class's representation are incoming references. That is, there should be no common representation between nested classes. For OGJ$_+$ to maintain deep *object* ownership, the interpretation of owner `This` is class-specific, since the meaning of owner `This` depends on the existence of receiver `this`. That is, an inner class cannot access fields declared in an enclosing class as owned by `This`. An inner class cannot also invoke methods defined in an enclosing class with formal parameter types and/or return types owned by `This`. See the erroneous statements in Fig. 4.2, where owner `This` in the outer class is not considered compatible with owner `This` in the inner class. Rationally, OGJ$_+$ allows a readonly wildcard owner parameterised reference to refer to objects in the outer class.

```
1  public class TestInner<Owner extends World> extends OwnedObject<Owner> {
2
3      OwnedObject<This> fieldInTestInner;
4
5      OwnedObject<This> meth(OwnedObject<This> o) {
6              return o;
7      }
8
9      class Inner<IOwner extends World> extends OwnedObject<IOwner> {
10
11         OwnedObject<This> field01InInner = fieldInTestInner;  //Error
12         OwnedObject<This> field02InInner = meth(field01InInner);  //Error
13
14         void methInInner() {
15             field01InInner = fieldInTestInner;  //Error
16             field01InInner = TestInner.this.fieldInTestInner;  //Error
17             TestInner.this.fieldInTestInner = field01InInner;  //Error
18
19             meth(field01InInner);  //Error
20             TestInner.this.meth(field01InInner);  //Error
21
22             field02InInner = meth(fieldInTestInner);  //Error
23             field02InInner = TestInner.this.meth(fieldInTestInner);  //Error
24         }
25     }
26 }
```

Figure 4.2: No common representation between inner classes and outer classes

Since the instances of inner classes are independent of their enclosing objects (i.e., the nesting is between classes, not objects), Java normally does not allow type variables declared in an inner class to be used in an outer class; however, Java allows vice versa. As a result, owner parameters declared in an inner class are safe from being used in an outer class, but not vice versa. For OGJ$_+$ to maintain proper independency between object, the use of an enclosing class's distinguished owner in any of the enclosing class's inner classes is prohibited. This is because there should be no nesting relationship between the distinguished owner of an enclosing class and the distinguished owners of the inner classes. If we are to allow the

use of the outer class's owner in an inner class, then the inner class will be able to define siblings to the outer class without the outer class's knowledge.

Although Java prohibits the explicit use of inner class's type variables in the enclosing class, generic methods facilitate a workaround, which OGJ+ can catch. See the erroneous statement in Fig. 4.3. `Owner` must be inside `MOwner`; and by extension, `Owner` must be inside the owner that `meth()` would infer. Since there is no nesting relationship between `Owner` and `IOwner`, the call to `meth()` violates the requirements by `Owner`.

```
1  class TestInner<Owner extends World> extends OwnedObject<Owner> {
2
3      <MOwner extends World> OwnedObject<MOwner> meth() {
4          OwnedObject<MOwner> o1 = new OwnedObject<MOwner>();
5          return o1;
6      }
7
8      class Inner<IOwner extends World> extends OwnedObject<IOwner> {
9          OwnedObject<IOwner> o2 = meth(); // Error: Owner is not inside IOwner
10     }
11 }
```

Figure 4.3: An outer class's owner is not inside the inner class's owner.

One limitation regarding the treatment of inner classes in OGJ+ is that implementing event-listeners as anonymous inner classes is not possible in the way illustrated in Fig. 4.4, since OGJ+ will not allow the method invocation, in line 27, as long as the actual parameter is owned by `This`. However, event-listeners can be implemented safely as anonymous inner classes in OGJ+, in the way illustrated in Fig. 4.5, as long as the return type, in line 20, is owned by `This`. Consider the changes made in lines 2, 10, 11 and 19 to provide the context covariance necessary to avoid restructuring the original code, if at all possible. As explained earlier in subsection 3.6.1 wildcard owners are mainly utilised in OGJ+ to facilitate context covariance, not only readonly references, and that is why a wildcard owner can only be bounded by World.

```
1  public interface EventListener<Owner extends World>  extends IOwnedObject<Owner>
       {
2      void fireEvent(Event<Owner> e);
3  }
4  public class Event<Owner extends World>  extends OwnedObject<Owner> {
5      public Event(OwnedString<Owner> e) {
6          System.out.println(e);
7      }
8  }
9  public class SomeObject<Owner extends World>  extends OwnedObject<Owner> {
10     private EventListener<This> lst;
11     public void setListener(EventListener<This> lst) {
12         this.lst = lst;
13     }
14     public void somethingHappened() {
15         lst.fireEvent(new Event<This>(new OwnedString<This>("Something Happened")
               ));
16     }
17 }
18 public class MyCode<Owner extends World>  extends OwnedObject<Owner> {
19     public EventListener<This> eventListener() {
20         return new EventListener<This>() {
21             @Override
22             public void fireEvent(Event<This> e) { ... }
23         };
24     }
25     public void meth() {
26         SomeObject<Owner> s = new SomeObject<Owner>();
27         s.setListener(eventListener()); //ERROR
28         s.somethingHappened();
29     }
30 }
```

Figure 4.4: Event-Listener Pattern implemented in OGJ_+ (uncompilable)

```
1  public interface EventListener<Owner extends World>  extends IOwnedObject<Owner>
       {
2      void fireEvent(Event<? extends World> e);   //CHANGED
3  }
4  public class Event<Owner extends World>  extends OwnedObject<Owner> {
5      public Event(OwnedString<Owner> e) {
6          System.out.println(e);
7      }
8  }
9  public class SomeObject<Owner extends World>  extends OwnedObject<Owner> {
10     private EventListener<? extends World> lst; //CHANGED
11     public void setListener(EventListener<? extends World> lst) {   //CHANGED
12         this.lst = lst;
13     }
14     public void somethingHappened() {
15         lst.fireEvent(new Event<Owner>(new OwnedString<Owner>("Something Happened
               ")));  //CHANGED
16     }
17 }
18 public class MyCode<Owner extends World>  extends OwnedObject<Owner> {
19     public EventListener<? extends World> eventListener() { //CHANGED
20         return new EventListener<This>() {
21             @Override
22             public void fireEvent(Event<? extends World> e) { ... } //CHANGED
23         };
24     }
25     public void meth() {
26         SomeObject<This> s = new SomeObject<This>();    //CHANGED
27         s.setListener(eventListener());
28         s.somethingHappened();
29     }
30 }
```

Figure 4.5: Event-Listener Pattern implemented in OGJ+ (compilable)

4.3 Statics

Java does not allow static declared member classes, methods and fields to use a type variable, and hence an owner variable, defined by the enclosing class. Moreover, if the enclosing class is a static member class, the same applies to its static members. This is because all type variables are non-static. Since a static member is shared along with its enclosing class, and since there should exist only a unique instance of that static member for all instances of the generic enclosing class, a static member needs to be accessible by objects located in any arbitrary ownership contexts.

The treatment of statics has not been specified explicitly in many proposals. Treating statics within the notion of owners-as-dominators [32] is not significantly different than within the notion of owners-as-modifiers [36, 44], as long as the object ownership heap graph is of only one tree structure. Huang and Milanova [32] force all static fields to belong to the root context, and assume that static methods have a virtual receiver `this`. Universe type system (UTS) [36, 44] does the same for static fields; but for static methods, the meaning of the `peer` modifier is slightly adapted. Normally, the `peer` modifier annotates an object as a sibling object; that is, a `peer` object has the same owner as the `this` object in order for both objects to share the same context. In the case of static methods, if a method is called as `peer Class.method()` at a static call site, then the callee's context will be the same as the caller's. If the same static method is called at a non-static call site, then the callee's context will be the same as the `this` object.

Summers et.al.[63] proposes extending the universe types heap structure with multiple trees, where each tree is rooted in a class. Classes can, therefore, own objects as static fields. This can be achievable since the root context of a UTS is the set of objects with no owner, and the root context is excluded from the verification of the nesting relationships between owners. In deep ownership, every single object should have an owner.

If we are to have an individual tree for each object owning static fields, while forcing such an object to be owned by `World`, then we will end up with a single tree that does not preserve the nesting relationships. If we are to treat an object, that owns static fields, as non-owned or as virtually owned by `World`, then we will end up breaking the rule of having all objects owned, as well as breaking the nesting relationships. OGJ$_+$ ensures that a static field is either in the root context or wildcard-owned in order to guarantee correct object encapsulation, while sustaining the possibility of accessing static fields from any arbitrary context.

4.3.1 Static Fields

A static field is also known as a class variable; so, as the name implies, a class variable is a per-class incarnation rather than a per-instance incarnation. As a rule, Java does not allow the use of a type variable as an actual type parameter within a static field; accordingly, there is no way to have an owner variable as an actual owner parameter within a static field.

As for the use of owner `This`, since the significance of `This` relies on the presence of receiver `this`, owner `This` cannot be used within a static field declaration. Consider the example of Fig. 4.6, Java only provides a warning about "accessing static filed" in response to the assignment statement in line 7; and if `meth()` is static, Java will provide an error because Java treats `this` as a non-static variable. The principle that should hold here is that each private static field—as a unique instance—must be accessible by any of the enclosing class's instances, no matter the location of these instances in the ownership tree. Although owner `World` can stand alone as a statically safe owner for static field declarations, we also have wildcard owners through which static fields can still be available for objects located in different contexts. A static field can then be declared `private` for proper information hiding, as need be.

```
1 public class Foo<Owner extends World> extends OwnedObject<Owner> {
2
3     static Foo<This> f1;     // Forbidden by OGJ+
4     static Foo<This> f2;     // Forbidden by OGJ+
5
6     public void meth(){
7         this.f1 = this.f2;
8     }
9 }
```

Figure 4.6: Illegal Static Field Declaration

4.3.2 Static Methods

Similar to static field declarations, owner variables declared by the enclosing class cannot be used within a static method, neither within the signature, nor within the body. Nevertheless, since a method can be owner parameterised, a static method's local variable or parameter can share contexts, as need be. An owner parameterised static method can be called, at a static call site, as `Class.<World>method()`, or as `Class.<Owner>method()`, so that the method can be evaluated either in the root context, or in the context of the caller, respectively. If the same static method is called at a non-static call site, then it can be called as above, or as `Class.<This>method()`, so that it can be evaluated in the context of the `this` object. If a static method is not owner parameterised, then it can only be evaluated within the root context, unless it involves wildcard owners. For the same reason as static fields, a static method cannot use owner `This` in the signature or in the body, as it should be available in arbitrary contexts.

4.3.3 Static Blocks and Nested Static Classes

Owner `This` is disallowed within static blocks and nested static classes. Methods declared inside these static contexts can be owner parameterised as need be. OGJ_+ treats anything declared within a nested static class

as a static context. Inner classes, methods, blocks, and fields which are declared within a static class are not allowed to use owner `This`.

4.4 Clone

In languages that have reference semantics, the assignment operator is not appropriate for duplicating an object. Creating a duplicate copy of an object is sometimes essential for equality, or inequality, comparisons [29]. Copying an object is commonly conducted through a special default operation that is usually defined in the language's root class, as is the case with Java's `java.lang.Object.clone()`. The `clone()` method creates a new instance of the object that is being copied, then returns this new instance. All of the new instance's fields are aliases to the fields of the original object. Such an operation is conventionally known as *shallow copy*. The `clone()` method's return type is `Object`; therefore, casting back the result to the original object is required. This default implementation can be overridden by some other custom behaviour in order, for example, to protect mutable fields from being affected by the behaviour of the original object. Normally, the override `clone()` gets a copy of the original object by invoking `super.clone()` until it reaches `Object.clone()`, which throws a `CloneNotSupportedException` if the class of the original object does not implement the interface `Cloneable`.

Another alternative approach to object copy is *deep copy*, where the referents of all of the original object's fields are copied, not aliased. Deep copy causes no dependency between the source and target objects. So, the structure of the target object is isomorphic to the structure of the source object. On the contrary, shallow copy makes the target object completely dependent on the source, with no isomorphic structures.

Grogono and Sakkinen [29] argue that copying operations should respect the semantic properties of objects rather than merely their syntactic properties, because ”shallow” is too shallow and ”deep” is too deep. In other words, the original and the cloned objects do not necessarily need to have isomorphic structures, and should not be completely dependent on one another. More specifically, objects which are not part of the private internal representation of the source object can be freely aliased within the cloned object; otherwise, references should be traced in order to copy their referents.

Noble et al. [51] discuss the idea of an ownership based mechanism called *sheep cloning*. Since an object owns its internal representation, the sheep cloning operation will copy all objects which are transitively owned by the source object, and will alias objects which are not part of the representation. It is not clear how the sheep cloning mechanism treats cycles within the object graph. There is an ongoing research aiming to formalize a type system that supports sheep cloning [38].

The example in Fig. 4.7 illustrates the implementation of the override `clone()` method used in the `ArrayList` collection class. Line 2 declares `v` as an object of type `ArrayList` whose initial value is a shallow copy of the current `ArrayList`. The `ArrayList` class is implemented as to have an array buffer to contain the elements of the `ArrayList`; which means that the backing array is constituting the internal representation of the `ArrayList`. Having the current `ArrayList` shallow copied, as in line 2, means that its backing array object becomes aliased, which is not safe. Therefore, the assignment statement in line 3 copies the backing array, `elementData`, into a new array object assigned to `v.elementData`. Since `elementData` forms the internal representation of the `ArrayList`, it should be owned by `This` in OGJ$_+$. As a result, the field access on the left hand side of the assignment statement, line 3, would be erroneous in OGJ$_+$, since `elementData` can only be accessed via receiver `this`.

```
1 public Object clone() {
2     ArrayList<E> v = (ArrayList<E>) super.clone();
3     v.elementData = Arrays.copyOf(elementData, size);
4     v.modCount = 0;
5     return v;
6 }
```

Figure 4.7: The `ArrayList`'s override `clone()` method

The use of wildcard owners can mitigate the above mentioned restriction in the way illustrated in Fig. 4.8. A `private` getter method (line 1) can be defined to return a wildcard owner parameterised reference to the backing array. This reference can then be used inside `clone()` (lines 8 and 9) to copy the original array into the array of the clone. We used the method `OwnedArray.copySegment()` in line 9 instead of `Arrays.copyOf()` (line 3, Fig. 4.7) which deals only with Java arrays. The receiver of `copySegment()` is the source array; the arguments are, respectively: the starting position in the source array, the destination array,

the starting position in the destination array, the number of array elements to be copied. Although implementing custom behaviours that deal with the `This`-owned objects is possible, shallow copying breaks deep ownership encapsulation, and hence should be forbidden in OGJ$_+$ because the clone will still be referring to the private representation of the original object; the statement in line 3, Fig. 4.7, and the statement in line 9, Fig. 4.8, do not copy the actual elements but only references to the elements. Since immutable arrays exist neither in Java nor in OGJ$_+$, facilitating deep copy is the appropriate solution for OGJ$_+$.

```
1  private OwnedArray<? extends World, E> getTable() {
2      return elementData;
3  }
4
5  @Override
6  public IOwnedObject<Owner> clone() {
7      ArrayList<Owner, E> v = (ArrayList<Owner, E>) super.clone();
8      OwnedArray<? extends World, E> tab = v.getTable();
9      elementData.copySegment(0, tab, 0, elementData.length());
10     v.modCount = 0;
11     return v;
12 }
```

Figure 4.8: An unreal OGJ$_+$ version of `ArrayList`'s `clone()`

We choose to facilitate an automatic version of deep copying. The `deepCopy()` operation in OGJ$_+$ can be overridden in a way similar to that of `java.lang.Object.clone()`, so that custom behaviours can be implemented, as need be, to alias the objects outside the representation of the copied object. OGJ$_+$ requires the object being copied to be serialisable. This can be done by making all of the classes that make up the class of the source object implement the `Serializable` interface. Having the source object serialisable means that all of its values can be serialised and written into a `ByteArray`; then, the `ByteArray` can be deserialised into a new object, during which we can assign the required owner; that is, copies can

be public, private or siblings; see lines 17, 20 and 23, Fig. 4.10. This approach takes care of the superclass fields, and traces the object graph to handle repeated references to the same object within the graph. There is no need to cast the result back to the source object, since `deepCopy()` is a generic method, whose type variable's bound, `IOwnedObject`, is wildcard owner parameterised. The example in Fig. 4.10 illustrates the use of the deep copy operations provided by OGJ$_+$.

```
1 public IOwnedObject<Owner> deepCopy() {
2     ArrayList<Owner, E> v = super.deepCopy();
3     v.modCount = 0;
4     return v;   }
```

Figure 4.9: The OGJ$_+$ `ArrayList`'s deep cloning operation

Consider Fig. 4.9 which is the cloning operation `deepCopy()` of the OGJ$_+$ `ArrayList`. In comparison with the `clone()` operation in Fig. 4.7, the backing array `elementData` does not need to be cloned again, because `elementData` was already deep copied by `super.deepCopy()` (line 2, Fig. 4.9) . Any other non `This`-owned object outside the private internal representation of `ArrayList` (e.g., `modCount`, line 3, Fig. 4.9) can still be accessed and aliased, as need be, in order to provide the required *sheep* (or sheep-like) cloning. The `Arrays.copyOf()` operation in Fig. 4.7 shallow copies `elementData`; that is, creates a new array object but only references to the elements are copied, not the actual elements. Using OGJ$_+$'s `deepCopy()`, everything is deep copied; and deep ownership encapsulation is guaranteed to be preserved, since the elements of an aggregate are part of its private internal representation.

The limitation of relying only on `deepCopy()` is that programming idioms which rely on shallow copying and on pointers to the internal representation of both the clone and the original object (e.g., to observe the changes since the clone was created) can no longer be implemented in OGJ$_+$ without being restructured to perform value comparisons rather than reference comparisons, if at all possible.

```
1  public class TestDeepCopy<Owner extends World> extends OwnedObject<Owner>{
2
3      static class Foo01<Owner extends World> extends OwnedObject<Owner>
4          implements java.io.Serializable {
5
6          public int value;
7      }
8
9      static class Foo<Owner extends World> extends OwnedObject<Owner>
10         implements java.io.Serializable {
11
12         public Foo01<Owner> test = new Foo01<Owner>();
13
14         public void meth() throws IOException, ClassNotFoundException {
15
16             this.test.value = 1;
17             Foo<Owner> x = super.deepCopy();
18
19             this.test.value = 2;
20             Foo<World> y = super.deepCopy();
21
22             this.test.value = 3;
23             Foo<This> z = super.deepCopy();
24
25             this.test.value = 4;
26
27             System.out.print("Foo: " + this.test.value + "; ");
28             System.out.print("z: " + z.test.value + "; ");
29             System.out.print("y: " + y.test.value + "; ");
30             System.out.print("x: " + x.test.value + ".");
31         }
32     }
33     static { begin(); }
34     public static void main(OwnedArray<World, OwnedString<World>> args)
35             throws IOException, ClassNotFoundException {
36
37         Foo<World> f = new Foo<World>();
38         f.meth();
39     }
40 }
41 ================================================================================
42 run:
43 Foo: 4; z: 3; y: 2; x: 1.
```

Figure 4.10: Deep copy operations in OGJ_+

4.5 Equals

Java's `equals()` method is another default operation defined in the root class `Object`. As the method's name implies, it performs equality comparisons. Nevertheless, `equals()` operates in exactly the same way as the equality testing operator '==', which performs identity equality comparisons rather than object equality comparisons. Similar to Java's `clone()`, the default implementation can be overridden by hand-coded versions. Java's `equals()` method is mostly overridden by, recursively, comparing referenced objects with `equals()`; for instance, see Fig. 4.11, line 12.

```
1  public boolean equals(Object o) {
2      if (o == this)
3          return true;
4      if (!(o instanceof List))
5          return false;
6
7      ListIterator<E> e1 = listIterator();
8      ListIterator e2 = ((List) o).listIterator();
9      while(e1.hasNext() && e2.hasNext()) {
10         E o1 = e1.next();
11         Object o2 = e2.next();
12         if (!(o1==null ? o2==null : o1.equals(o2)))
13             return false;
14     }
15     return !(e1.hasNext() || e2.hasNext());
16 }
```

Figure 4.11: `java.util.AbstractList.equals()`

The `equals()` signature is: `boolean equals(Object obj)`. This means that the overriding implementation should check if the method's actual parameter is of the required type (line 4, Fig. 4.11); this is normally the type of the enclosing class, a superclass, or the abstract type it implements. If the check holds true, then it is inevitable to downcast the method's parameter to the verified type (line 8) in order to perform the required equality comparison between the receiver and the argument of the overriding implementation. This standard code reuse practice became

entangled by the use of ownership types, since not only the object graph should be taken into account but also ownership information should be evaluated. Moreover, with deep ownership types, the nesting information should also be evaluated.

```
1  public <O extends World> boolean equals(IOwnedObject<O> o) {
2      if (o == this)
3          return true;
4      if (!(o instanceof List))
5          return false;
6
7      ListIterator<Owner,E> e1 = listIterator();
8      ListIterator<O, E> e2 = ((List<O, E>) o).listIterator();
9      while(e1.hasNext() && e2.hasNext()) {
10         E o1 = e1.next();
11         IOwnedObject<? extends World> o2 = e2.next();
12         if (!(o1==null ? o2==null : o1.equals(o2)))
13             return false;
14     }
15     return !(e1.hasNext() || e2.hasNext());
16 }
```

Figure 4.12: `OGJ.java.util.AbstractList.equals()`

The example in Fig. 4.11 shows the overriding `equals()` implementation for the collection class `java.util.AbstractList` of Java 1.6. In line 8, consider the downcast of the method's parameter to the abstract type `List`. First, OGJ$_+$ does not allow raw cast since all objects should be owner parameterised. Second, passing a `List` object to the method means upcasting the argument to `Object`, then downcasting it back to `List`, which is something that cannot be done in OGJ$_+$ as explained in section 3.5 by the example in Fig. 3.9. To overcome this restriction, OGJ$_+$ only permits this kind of downcasting if all of the additional outside owners are hidden via wildcards. The example in Fig. 4.12 shows the overriding implementation of `equals()` for `OGJ.java.util.AbstractList`. The latter class has the type variable `E` declared in its header with a placeholder owner as follows: `E extends IOwnedObject<? extends World>`.

That is, the downcast in line 8 is allowed. In conclusion, the restrictive form of casting in OGJ$_+$ might seem to impose an impact on the way an `equals()` is written in OGJ$_+$, but the simple decision of using a placeholder owner within the class header mitigated this impact. As illustrated by the example in Fig. 4.12, we did not need to restructure the example in Fig. 4.11, but did supply only owner parameters. Note also that the OGJ$_+$'s `equals()` is owner polymorphic so that equality testing between objects with different ownership information can be conducted.

```
1  class Foo<Owner extends World> extends OwnedObject<Owner> {
2      Bar<This> bar;
3
4      private Bar<? extends World> getBar() {
5          return this.bar;
6      }
7
8      public <O extends World> boolean equals(IOwnedObject<O> o) {
9
10         Foo<O> f = (Foo<O>) o;
11
12         // return bar.equals(f.bar);        //ERROR
13         // return f.getBar().equals(bar);   //ERROR
14
15         return bar.equals(f.getBar());
16     }
17 }
```

Figure 4.13: `equals()` accessing two unrelated private objects

Consider the example in Fig. 4.13, the field access `f.bar` in line 12 is erroneous since `bar` is owned by `This`, and hence can only be accessed via receiver `this`. To overcome this restriction, we can define a getter method, such as `getBar()` in line 4, and then invoke it as in line 15. Nevertheless, `getBar()` cannot be used the way illustrated in line 13, since there is no guarantee that the returned wildcard-owned type is inside `bar`'s owner, as explained in section 3.6. Moreover, OGJ$_+$ prohibits method invocations if the receiver and the actual parameter are both wildcard-owned. That, in fact, imposes a limitation on one's ability to use OGJ$_+$'s `equals()` and

method invocation in general (see Fig. 4.14, line 28); and would obstruct both the implementation and utilisation of some programming idioms and design patterns (e.g., caching pattern).

```
1  public class DefaultCacheImpl<Owner extends World>
2         extends OwnedObject<Owner>
3         implements Cache<Owner> {
4
5      private HashMap<This,
6                      OwnedObject<? extends World>,
7                      OwnedObject<? extends World>> map;
8              ...
9      public Collection<? extends World,
10                        Map.Entry<? extends World,
11                                  OwnedObject<? extends World>,
12                                  OwnedObject<? extends World>>> getAll() {
13
14          return new ArrayList<This,
15                               Map.Entry<? extends World,
16                                         OwnedObject<? extends World>,
17                                         OwnedObject<? extends World>>>(map.
                                               entrySet());
18     }
19             ...
20 }
21
22 public class MyImpl<Owner extends World> extends OwnedObject<Owner>{
23
24   DefaultCacheImpl<This> c = new DefaultCacheImpl<This>();
25
26   public boolean meth(OwnedObject<Owner> obj) {
27             ...
28       return c.getAll().equals(((DefaultCacheImpl<Owner>) obj).getAll()); //ERORR
29   }
30 }
```

Figure 4.14: Illegal `equals()` invocation (receiver and actual parameter are wildcard-owned)

4.6 Exception Handling

Java utilises exceptions to handle errors and exceptional events that could happen during the operation of a method at runtime. Whenever such an event occurs, the relevant method creates an exception object and hands it to the runtime system; this procedure is called *throwing an exception*. Whenever an exception is thrown, the runtime system goes over the *call stack* (the chain of methods called to reach the method that has thrown the exception) in search of a method that contains an exception handler.

Exception handling in type systems that enforce ownership encapsulation properties raises the issue that an object may create an exception in one ownership context, while the object that has the exception handler might be in another context. Consider the example in Fig. 4.15, the object of class `FoosFather` owns `Foo`; and `Foo` owns `FoosChild`. `FoosChild` creates an exception object, at line 17, while `FoosFather` handles the exception. `Foo` is in the call stack, and is neither throwing nor handling an exception, as depicted in Fig. 4.16. Consider the object ownership structure in Fig. 4.17, `FoosChild` is in a different ownership context (`Foo`'s context) than that of `FoosFather`. If method `FoosChild.init()` creates an exception object local to `Foo`'s context, then the exception will not be able to propagate farther `Foo`'s context boundary.

Dietl and Müller [24] explain and evaluate four approaches to exception handling in ownership type systems: 1) cloning exception objects from the context where they are created to the context where they are handled; 2) transferring exception objects from the context where they are created to the context where they are handled; 3) treating exceptions as global data that can be accessed from all contexts; and 4) propagating exceptions via read-only references, which may cross context boundaries. The authors think that both global exceptions and read-only exceptions are more applicable alternatives, since both approaches do not lead to specification overhead, and do not increase the complexity of the type system signifi-

```
1  class FoosFather<Owner extends World> extends OwnedObject<Owner>{
2      Foo<This> myChild;
3      void init() {
4          try {
5              myChild.init();
6          } catch( FooException e) {
7              System.err.println ( e.origin.getClass() );
8          }   }   }
9  public class Foo<Owner extends World> extends OwnedObject<Owner> {
10     FoosChild<This> myChild;
11     void init() throws FooException {
12         myChild.init();
13     }   }
14 class FoosChild<Owner extends World> extends OwnedObject<Owner> {
15     void init() throws FooException{
16         FoosChild<World> f = deepCopy();
17         throw new FooException(f);
18     }   }
19 class FooException extends Throwable{
20     OwnedObject<World> origin;
21     public FooException(OwnedObject<World> origin) {
22         this.origin = origin;   }   }
```

Figure 4.15: Exception Handling Example

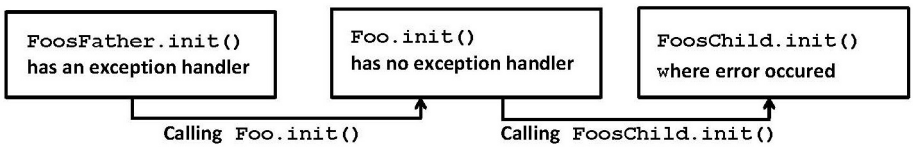

Figure 4.16: Depiction of the call stack exemplified in Fig. 4.15

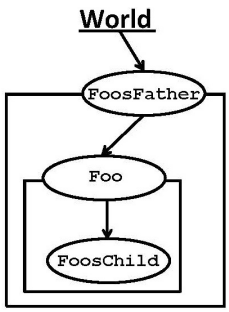

Figure 4.17: Depiction of the object ownership structure exemplified in Fig. 4.15

cantly. The authors explain that read-only exceptions have two limitations. First, read-only references lead to a weaker ownership invariant. The second limitation is that an exception object is modified while being propagated. Handling this would require cloning the exception, modify the clone, then throw the clone. In relation to global exceptions, the declaration and propagation of exceptions are straightforward. Global exceptions are viable for ownership type systems that permit references to objects in ancestor contexts, as is the case with deep ownership.

OGJ$_+$ treats exception classes the same way as static declarations, where owner `World` and wildcards are the only allowed owner parameters within an exception class. If the example in Fig. 4.15 is implemented with wildcard owners used in lines 20 and 21, then we could have normally replaced `f`, in line 17, by `this` and removed line 16; this is OGJ$_+$'s best option. The example uses owner `World` within `FooException`; that is, we cannot use `this` in place of `f` in line 17, since casting `this` to `OwnedObject<World>` is not possible; owners do not match. Using the default `deepCopy()` operation provided by OGJ$_+$, as in line 16, allows us to get the required public deep copy of the current `this`.

4.7 Enum Types

An enumeration type is mostly declared to contain a set of named constant values. Nevertheless, in Java, it is more appropriate to say that an enumeration type contains a set of closely related elements; for example, `Colour{RED, BLUE, YELLOW}`. This is because an enumeration type in Java is a distinct compiler-generated class, not an arithmetic type, whose constant values cannot be directly set to numeric values.

Java *enum types*, or *enums*, were introduced in the type safe JDK 1.5, as a type safe facility, along with genericity. Nevertheless, enums cannot be generic types, although enums preserve type safety by implicitly extending the abstract generic class `java.lang.Enum`, whose header is as follows:

```
public abstract class Enum<E extends Enum<E>>
    implements Comparable<E>, Serializable
```

The class `Enum` is self-referencing in its generics, and implements `Comparable` whose type argument is `Enum`'s type variable. This ensures that an enum of one type can only be compared with another enum of the same type.

Since enums extend `java.lang.Enum`, enums cannot extend any other class or enum, as there is no multiple inheritance in Java. Enums are implicitly `final`; thus cannot be subclassed. Also, an enum is implicitly `static` when being a member of a class. Enum constants are implicitly `static final`. There is no way to assign something other than the predefined constants to an enum variable. Since enums are classes, it is possible to define constructor, methods, variables, and member classes inside Java enum. Nevertheless, a constructor must be declared `private`, to preserve the property that an enum is neither allowed to be instantiated nor extended.

Since an enum class, as well as its constants, cannot be compromised via the getter/setter idiom, the safety measures that OGJ$_+$ applies are directed to any variable, method, or member class declared inside an enum class. What applies to anything declared outside an enum is applied to everything declared inside an enum. If an enum is declared as a member of a class, which is mostly the case, then OGJ$_+$ will treat the enum, and anything declared inside it, as being in a static context. An enum cannot extend classes but can implement interfaces; and therefore, an enum can only implement `IOwnedObject`, or its descendants.

4.8 Implementation Methodology

This section describes the implementation approach adopted to extend Java with the encapsulation system described above. The implementation is done on the `javac` open source compiler.

4.8.1 Ownership Domains and Types

The ownership domains `OGJ.Owners.World` and `OGJ.Owners.This` are implemented as blank interfaces. `This` inherits `World`, which inherits `Object`. The root class `OGJ.OwnedObject` is an owner parameterised class, which implements the owner parameterised root interface `OGJ.IOwnedObject`. The default equality test operation `equals()` is declared in `IOwnedObject`, and thus is overridden in `OwnedObject`. `OwnedObject` also contains the default copy operation `deepCopy()`, which is declared `protected`.

The default array type in OGJ$_+$, `OGJ.OwnedArray`, is declared `final`, with a `public` constructor that takes an integer parameter as the initial array size, then initialises a new array object through a `private` constructor. `OwnedArray` takes an owned type argument, as the type of the elements, in the way explained in section 4.1, then creates a normal Java array type and stores its value as a `private` field, which can be initialized by the `private` constructor and accessed directly. `OwnedArray` is declared serializable in order to conform with `deepCopy()` that uses Java serialization, as explained in section 4.4. `OwnedArray` bundles the following operations: `get()`, `set()`, `length()`, `resize()`, and `copySegment()`.

Classes in the Java standard library (e.g., `Integer`, `Number`, `String`, etc.) have their corresponding classes in OGJ$_+$ (e.g., `OwnedInteger`, `OwnedNumber`, `OwnedString`, etc.); they are all owner parameterised, serializable, and have `public` constructors.

4.8.2 Type Checking

The OGJ$_+$ code is type-checked through an extension to the Java Compiler (`javac`). `javac` is a program to put Java language source code files (`.java`) into an intermediate representation (IR), which is an abstract language independent of the specifics of a particular machine (platform-neutral). The IR of a Java program is known as bytecode. The compiler processes source files to generate output machine-readable executable bytecode class files (`classname.class`). For each class defined in the source code file(s), the compiler creates a class file. So, Java class files, or bytecode classes, are the accepted form that can be converted, by the Java Virtual Machine, into native platform-specific executables. In that sense, `javac` is typically a front end of a traditional compiler, which parses source code in order to generate IRs.

For the compilation process to get started, the `JavaCompiler` class, which is defined in `com.sun.tools.javac.main`, will be invoked to read the source files specified on the command line. The compiler, first, enters a parsing phase, in which it processes source files with the help of classes defined in `com.sun.tools.javac.parser.*`. The parser will then call the lexer in order to generate a stream of tokens from an input stream of characters, then will map them into an abstract syntax tree (AST); this can be thought of as an advance towards the utilization of the package `com.sun.source.tree`.

As far as OGJ$_+$ is concerned, `javac` employs the AST to symbolize and process programs at compile time; and eventually, the AST is transformed to Java bytecode and written to a class file(s). The root class for AST nodes is `com.sun.tools.javac.tree.JCTree`, which implements `com.sun.source.tree.Tree`. To work with a tree, it is essential to traverse it somehow; `javac` utilizes the design pattern *Visitor*. The visitor class for trees is `com.sun.tools.javac.tree.JCTree.Visitor`, which encloses a number of visitor methods to carry out attribution on the AST. OGJ$_+$ injects only a couple of visiting messages, before and after

attribution, into the `JavaCompiler` class. The visitor methods which are overridden by OGJ$_+$ are as follows:

- `visitVarDef(JCVariableDecl tree)`
  Deals with variable declarations (fields, variabls, parameters).
- `visitMethodDef(JCMethodDecl tree)`
  Deals with method declarations.
- `visitClassDef(JCClassDecl tree)`
  Deals with class declarations.
- `visitApply(JCMethodInvocation tree)`
  Deals with method invocations.
- `visitSelect(JCFieldAccess tree)`
  Deals with field accesses and assignments.
- `visitTypeCast(JCTypeCast tree)`
  Deals with type casts.
- `visitTypeApply(JCTypeApply tree)`
  Deals with parameterised types.
- `visitTypeParameter(JCTypeParameter tree)`
  Deals with formal class parameters.
- `visitAssign(JCAssign tree)`
  Deals with assignment statements.
- `visitNewClass(JCNewClass tree)`
  Deals with `new(...)` operations.
- `visitReturn(JCReturn tree)`
  Deals with `return` statements.
- `visitBlock(JCBlock tree)`
  Deals with statement blocks.

Each of the above methods is overridden to behave, and perform checks, as explained in a relevant section or subsection within the thesis. For example, in relation to nesting verification, subsection 3.4.2 describes

part of the behaviour of `visitVarDef()`; in relation to instantiation and casting, section 3.5 describes the dehaviour of `visitNewClass()` and `visitTypeCast()`, and so on.

4.8.3 Testing

OGJ$_+$ implementation is, by all means, a Test-Driven Development (TDD). The process is mainly depending on reiterating brief development cycles. At the beginning, a test case, that contains a faulty OGJ$_+$ construct, is written to define the required new function, or modification; then, the code is written to satisfy that test. This process requires automated unit tests to verify that all added functionalities are still working after code changes. We used the JUnit testing framework as a test case execution tool for OGJ$_+$.

Test Cases

Most of the test cases are moderately small; they range in length from 3 to 100 lines of code. The length varies mainly depending on the integrity of the number of errors that `javac` error log can report for each OGJ$_+$ example program (test case). Sometimes, one test case cannot comprise all of the potential deviations for the same construct; it is therefore at times essential to have more than one test case for the same construct. We have more than 70 Java class files to test error reporting; some of these files can report up to 50 errors; some others contain correct OGJ$_+$ code to detect problems with the type checker. That is, there are two kinds of tests: failing tests and passing tests. A failing test checks that a given file does indeed fail, as should be; a passing test verifies that a given file does compile, as should be. Furthermore, applying OGJ$_+$ to the Collections Framework has turned out very useful in detecting subtle bugs that would otherwise have gone unnoticed.

4.8.4 Usage

A minimal runnable OGJ$_+$ program should have the lines shown in Fig. 4.18. The Compiler is available for the Windows platform from the following link:

`http://homepages.ecs.vuw.ac.nz/~ahmkhal/`

To run OGJ$_+$, Java should be run from the command prompt as follows:

`X:\>java -jar "OGJ_JDK6_v20130630.jar" "Class.java"`

See subsection 4.1.1 for discussion on OGJ$_+$'s `main()` method (and thus the `begin()`).

```
1  import static OGJ.ArraylessMain.*;
2  import OGJ.Owners.*;
3  import OGJ.*;
4
5  public class AnyClassName<OwnerVariable extends World>
6      extends OwnedObject<OwnerVariable> {
7
8      static { begin(); }
9      public static void main(OwnedArray<World, OwnedString<World>> args) {
10
11     }
12 }
```

Figure 4.18: Mandatory declarations required for a minimal runnable OGJ$_+$ program

Chapter 5

Generic Ownership Compliant Collections

To evaluate the applicability of OGJ$_+$'s encapsulation system to a real life code base, we decided to refactor the JDK 1.6 Collections Framework. We found that Java Collections Framework is large, complex and important enough to be representative of the encapsulation issues we are targeting (e.g., the iterator pattern). Not to mention the fact that ownership was first introduced to target aggregation and container objects; and in the end, the refactored framework is in fact integrated to the new language. That is, OGJ$_+$ has its own collection library.

The Java Collections Framework [1, 5, 49] was introduced in JDK 1.1, and then redesigned more thoroughly in JDK 1.2. With generics introduced in JDK 1.5, a collection object became restricted to hold a particular data type, since the only thing a collection knows it holds was a handle to an `Object`. Finding out type mismatches at compile time is known as *compile-time type safety*, which relieves programmers of the burden of casting when reading elements from collections. So, what we are about is supporting owner detection, as companion to type detection, through generic declarations.

In terms of encapsulation, a collection object can have its state compro-

mised via an incoming reference to its internal representation. How? One of the very short, but most substantial, answers is: Iterators. The idea is that for a collection object to be able to share its element objects, an iterator needs to access the elements stored in that collection. As explained in subsection 2.3.1, an iterator can bypass the collection's interface and refer directly to any arbitrary element by maintaining an incoming reference to the collection's private representation that can in turn be mishandled via public getter/setter methods.

With ownership types, it is possible to have the internal representation explicitly declared as owned by its respective collection class. That is, a collection's representation is hidden: constrained to interact only with its owner collection object, with no incoming aliases.

This chapter explains how we refactored the JDK 1.6 collections' code base, in order to use OGJ$_+$'s ownership support. The classes under study are the ones which are circled in Fig. 5.1 and Fig. 5.2. These classes include all of the general-purpose implementations, namely `ArrayList`, `LinkedList`, `ArrayDeque`, `PriorityQueue`, `HashMap`, `LinkedHashMap`, `TreeMap`, `HashSet`, `LinkedHashSet`, and `TreeSet`. Also, the legacy implementations `Vector` and `Hashtable` will be investigated, as well as the special purpose implementation `IdentityHashMap`. The non-circled components in Fig. 5.1 and Fig. 5.2 are (1) the abstract data types, or interfaces, of which the collection classes provide implementations, and they appear in red colour; and (2) the extended abstract classes that provide the skeleton implementations of the relevant abstract data types. All of these classes and interfaces are subject to refactoring.

Typically, the refactoring of a class (or interface) starts off by declaring a distinguished owner for the class itself, then the appropriate owner parameters and bounds are supplied to the type parameters, as explained earlier in the thesis, and as will be explained later in this chapter. If the class is not extending another class in the hierarchy depicted in Fig. 5.1 (or Fig. 5.2), then the class should extend `OwnedObject` and the owner

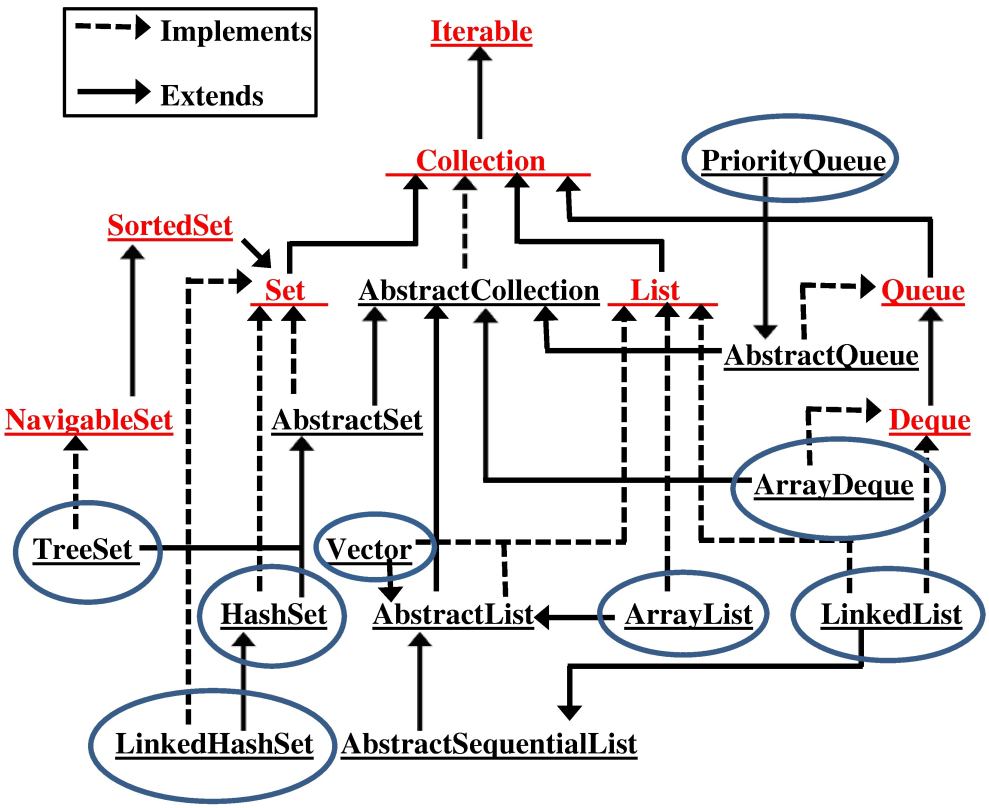

Figure 5.1: Iterable Collections

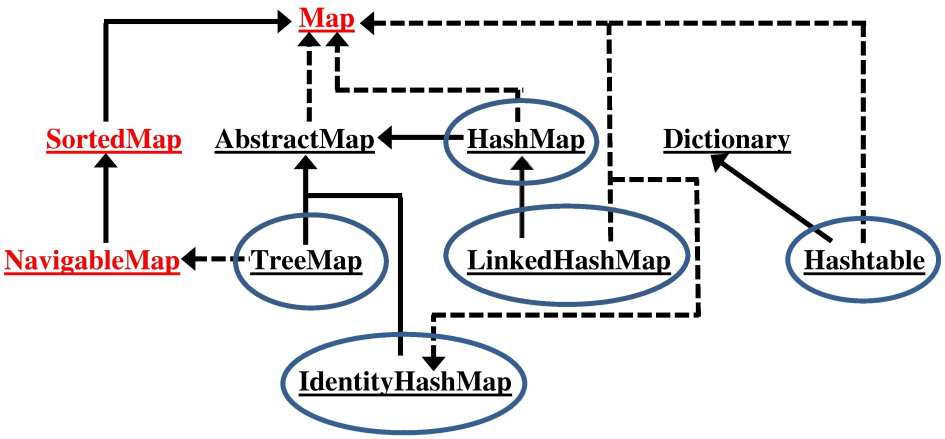

Figure 5.2: Maps

of the class should be preserved as explained earlier in chapter 3. In a similar manner, an interface should extend `IOwnedObject`, if it is not extending another interface in its respective hierarchy. Refactoring the body of a class starts off by identifying its representation part and making it owned by `This` (i.e., to be private and hidden). Owner `This` will not allow any access to that representation part unless through variable `this`. The other non-representation objects will be initially siblings owned by the same owner as the defining collection class, which means that these objects can only be shared with objects owned by the same owner as the current instance of the defining collection class. Exceptions occur when, for example, a representation object is originally being assigned to any of the other objects. In this case, such objects will need to be made owned by `This`, and thus they will need to be only accessed via variable `this`, which at times is not the originally intended receiver. Here we come to what we call cross incompatibility between owners. The remainder of this chapter explains the modifications found necessary, and why a particular object had to become private, sibling, public or subject to context covariance.

Identifying the representation part for this particular refactoring was straightforward, as we considered every private field, in the original implementations, a representation object. Otherwise, the determination of the representation part of an object relies on human judgement, if not design principles.

Since most collection classes have compile-time dependencies on other classes in the framework, the priority was always to make sure that the whole set of classes under refactoring can altogether be successfully compiled with OGJ_+, after every modification cycle. For rapid error reporting for that whole set of classes, we utilised the JUnit testing framework which provides an automated infrastructure for running repeatable tests. That is, the collection classes were dealt with as passing test cases that should altogether produce no errors. Modifications to one class can cause another

to fail to compile, while the former can still pass the compilation. Thus, the integrity, and quality of having unconflicted set of class relationships, was a priority during every implementation cycle.

5.1 Interfaces

The set of classes mentioned above are descended from the two main collection interfaces: `Collection` (Figure 5.1) and `Map` (Figure 5.2). The set of interfaces and classes involved in both hierarchies are in the package `java.util`. This study is mainly concerned with five abstract data types: `List`, `Queue`, `Deque`, `Map`, and `Set`. These general purpose collections can be classified into iterable collections and maps. Iterable collections are those interfaces that extend `Collection`; and by extension, their implementor classes. Maps are all interfaces and classes which are rooted by the interface `Map`.

Although the interface `Iterable` (in package `java.lang`) is outside the Framework, `Collection` extends `Iterable` so that lists, queues, and sets can be used with the foreach statement. It is not possible to iterate over a `Map` object directly; instead, `Map` provides `Collection` views so that a `Map` object can be iterated over through these views. That is, maps are not iterable collections; and thus, there is no need for `Map` to extend `Iterable`.

5.1.1 The `Collection` Interface

To apply Generic Ownership to the interface `Collection`, the appropriate owner parameters and bounds are supplied to the interface header as shown in line 1, Fig. 5.4. Fig. 5.3 and Fig. 5.4 are the original JDK 1.6 and the OGJ$_+$ compliant versions of the interface `Collection`, respectively.

```
1  public interface Collection<E> extends Iterable<E> {
2
3      // Adding Elements
4      boolean add(E e);
5      boolean addAll(Collection<? extends E> c);
6
7      // Querying the Contents of a Collection
8      int size();
9      boolean isEmpty();
10     boolean contains(Object o);
11     boolean containsAll(Collection<?> c);
12
13     // Removing Elements
14     boolean remove(Object o);
15     boolean removeAll(Collection<?> c);
16     boolean retainAll(Collection<?> c);
17     void clear();
18
19     // Making a Collection's Contents Available for Further Processing
20     Iterator<E> iterator();
21     Object[] toArray();
22     <T> T[] toArray(T[] a);
23 }
```

Figure 5.3: JDK 1.6 `Collection` Interface.

```
1  public interface Collection<Owner extends World, E extends IOwnedObject<? extends
       World>> extends Iterable<Owner,E> {
2
3      // Adding Elements
4      boolean add(E e);
5      boolean addAll(Collection<Owner, ? extends E> c);
6
7      // Querying the Contents of a Collection
8      int size();
9      boolean isEmpty();
10     boolean contains(E o);
11     <O extends World> boolean containsAll(Collection<O, ? extends E> c);
12
13     // Removing Elements
14     boolean remove(E o);
15     <T extends E> boolean removeAll(Collection<Owner, T> c);
16     <T extends E> boolean retainAll(Collection<Owner, T> c);
17     void clear();
18
19     // Making a Collection's Contents Available for Further Processing
20     <O extends World> Iterator<O, E> iterator();
21     <O extends World> OwnedArray<O, E> toOwnedArray();
22     <T extends E> OwnedArray<Owner, T> toOwnedArray(OwnedArray<Owner, T> a);
23 }
```

Figure 5.4: OGJ$_+$ `Collection` Interface.

The OGJ$_+$ `Collection` header has `Owner` as the owner of `Collection`, and `E` as the type of the elements. `E` is bounded by `IOwnedObject` which is the root interface in OGJ$_+$, and which `OwnedObject` implements, so that classes and interfaces can be used as bound objects. The owner of the elements is subject to context covariance (recall subsection 3.6.1). `Collection` extendes the interface `Iterable`, which must be within the same ownership context as `Collection`, and thus is owned by `Owner`. As for the definition of `Iterable`, it extends `IOwnedObject` and it reads as follows:

```
public interface Iterable<Owner extends World,
        T extends IOwnedObject<? extends World>>
         extends IOwnedObject<Owner> {
    <O extends World> Iterator<O, T> iterator(); }
```

According to Naftalin and Wadler [49], the methods which are defined by `Collection` can be classified into four groups in accordance with their functionalities: adding elements, querying the contents of a collection, removing elements, and making a collection's contents available for further processing. In the rest of this subsection, I will clarify the required adjustments in the signatures of these methods.

Adding Elements

These are the methods in lines 4 and 5, Fig. 5.4. The `add()` method did not require any modification in the signature. For `addAll()`, other than supplying the parameter type with the relevant owner parameter, there are no significant adjustments to be made.

Querying the Contents of a Collection

These are the methods in lines 8-11, Fig. 5.4. According to Naftalin and Wadler [49], the signatures of `contains()` and `containsAll()` in the original generified `Collection` (lines 10 and 11, Fig. 5.3) are a more liberal alternative than the following:

```
boolean contains(E o);
boolean containsAll(Collection<?  extends E> c);
```

The latter signatures catch more errors at compile time (while also ruling out some sensible tests). We could not adopt a liberal OGJ_+ version similar to that in line 7, Fig. 5.5, because OGJ_+ will not allow the method invocation in line 8 unless `Owner` (the owner of `c`) is inside the owner of `o`. Since the owner of `o` is unknown, determining the nesting structure at compile time will not be possible. OGJ_+ can determine the nesting structure through the ownership information provided by the class header. OGJ_+ knows that the owner of the class must be inside all of the other owners involved in the class header. That is, using `E` as the type of `o` means that `Owner` is inside the owner of `o`.

```
1  public class MyCollection<Owner extends World, E extends OwnedObject<? extends
       World>>
2      extends OwnedObject<Owner>
3      implements Collection<Owner,E> {
4
5          Collection<Owner,E> c;  // Backing Collection
6                          ...
7          public boolean contains(OwnedObject<? extends World> o) {
8              return c.contains(o);  // Error
9          }
10 }
```

Figure 5.5: A liberal signature of `contains()`

So, what applies to `add()` and `addAll()` applies to `contains()` and `containsAll()`. The only difference is that `containsAll()` is owner-polymorphic. This is because the `equals()` method is owner-polymorphic; see subsection 4.5. The idea is that the implementation of the overriding `equals()`, in the abstract class `AbstractSet`, compares the specified `Set` (method parameter) for equality with the `this Set` by checking if the `this Set containsAll()` the elements in the specified `Set`, after confirming that they both have the same size. So, to send a `containsAll()` message by passing the specified `Set` as argument, `containsAll()` needs to be able to infer the owner of the specified `Set`, which is already inferred by `equals()`.

Removing Elements

These are the methods in lines 14-17, Fig. 5.4. The signature of `remove()` is treated in a similar fashion as the signature of `contains()`. Also, the signatures of `removeAll()` and `retainAll()` similarly correspond to the signature of `addAll()`, although they utilise method genericity instead of using wildcards. Apart from the owner parameter, in the resulting byte code there is no difference between the signature of the generic `removeAll()` and its corresponding signature that uses a wildcard. We

found the generic signature mandatory for our refactoring, however. The example in Fig. 5.6 shows the overriding `removeAll()` implementation for the collection class `AbstractCollection` of OGJ$_+$. The method invocation in line 5 would not compile if the signature of `removeAll()` required its argument to be a `Collection<Owner, ? extends E>`, since `contains()` actual argument `E` cannot be converted to the capture of `? extends E` by method invocation conversion. What applies to `removeAll()` applies straightforwardly to `retainAll()`.

```
1  public <T extends E> boolean removeAll(Collection<Owner, T> c)  {
2      boolean modified = false;
3      Iterator<Owner, ? extends E> e = iterator();
4      while (e.hasNext()) {
5          if (c.contains((T)e.next())) {
6              e.remove();
7              modified = true;
8          }
9      }
10     return modified;
11 }
```

Figure 5.6: OGJ$_+$ `AbstractCollection.removeAll()`

Making a Collection's Contents Available for Further Processing

These are the methods in lines 20-22, Fig. 5.4. The `iterator()` method returns an `Iterator` over the receiver collection. The two overloaded `toOwnedArray()` methods transfer the contents of the receiver collection into a new `OwnedArray` object. Both methods are the OGJ$_+$ versions of the two overloaded `toArray()` methods in the JDK 1.6 `Collection` interface. The advantage of the second `toOwnedArray()` method, line 22, is that it infers the type of the returned array from the method's argument type in order to accurately realize control over the runtime type of the returned array.

There are times when we need to transfer the contents of a sibling collection object into a private `This`-owned array object. For example, the `ArrayList` class has a constructor that takes a `Collection` argument to construct a list containing the elements of that `Collection`. Consider the example in Fig. 5.7, the formal parameter `c`, line 6, is owned by `Owner`, while the backing array object `elementData` is owned by `This`. That is, we cannot use an assignment statement such as `elementData = c.toOwnedArray();` but we can have the returned `OwnedArray` owned by `This`, if `toOwnedArray()` is owner-polymorphic so that the assignment statement would appear as in line 7.

```
1  public class ArrayList<Owner extends World, E extends IOwnedObject<? extends
       World>> extends ... implements  ... {
2
3      private transient OwnedArray<This, E> elementData;
4      private int size;
5                          ...
6      public ArrayList(Collection<Owner, ? extends E> c) {
7          elementData = (OwnedArray<This, E>) c.<This>toOwnedArray();
8          size = elementData.length();
9      }
10 }
```

Figure 5.7: `ArrayList` constructor takes a `Collection` argument

The implementation of `toOwnedArray()` is one of the implementations that required more flexibility in handling the `iterator()` method's owner parametricity. That is, `iterator()` needed to be owner-polymorphic in order to infer owner arguments that do conform to the owner arguments of other iterator-manipulating owner-polymorphic methods.

5.1.2 The `Map` Interface

`Map` is the other dominant interface, in that it sets up the second interface hierarchy in the Collections Framework. Since any descendant of this interface represents a mapping object that maps keys to values, the header of this interface involves two type parameters: `K` for the type of the keys, and `V` for the type of the values. Fig. 5.8 and Fig. 5.9 are the original JDK 1.6 and the OGJ$_+$ compliant versions of the interface `Map`, respectively. Unlike `Collection`, `Map` directly extends `IOwnedObject`, as it is not iterable; and as a matter of course, `IOwnedObject` preserves the same owner as `Map`. In close parallel to the categories of methods defined by `Collection`, the methods in `Map` can be grouped into: adding mappings, removing mappings, querying the contents of a map, and providing collection views.

```
1  public interface Map<K,V> {
2
3      // Adding Mappings
4      V put(K key, V value);
5      void putAll(Map<? extends K, ? extends V> m);
6
7      // Removing Mappings
8      V remove(Object key);
9      void clear();
10
11     // Querying the Contents of a Map
12     int size();
13     boolean isEmpty();
14     boolean containsKey(Object key);
15     boolean containsValue(Object value);
16     V get(Object key);
17
18     // Providing Collection Views
19     Set<K> keySet();
20     Collection<V> values();
21     Set<Map.Entry<K, V>> entrySet();
22 }
```

Figure 5.8: JDK 1.6 `Map` Interface.

```
1  public interface Map<Owner extends World, K extends IOwnedObject<? extends World
       >, V extends IOwnedObject<? extends World>> extends IOwnedObject<Owner> {
2
3      // Adding Mappings
4      V put(K key, V value);
5      void putAll(Map<Owner, ? extends K, ? extends V> m);
6
7      // Removing Mappings
8      V remove(K key);
9      void clear();
10
11     // Querying the Contents of a Map
12     int size();
13     boolean isEmpty();
14     boolean containsKey(K key);
15     boolean containsValue(V value);
16     V get(K key);
17
18     // Providing Collection Views
19     Set<Owner,K> keySet();
20     Collection<Owner,V> values();
21     Set<Owner, Map.Entry<? extends World,K, V>> entrySet();
22 }
```

Figure 5.9: OGJ$_+$ `Map` Interface.

As appears in Fig. 5.9, there is no significant changes in the first category; only `putAll()` needed to have an owner for the parameter type. Similar to `contains()`, in the previous subsection, the method signatures of the second and third categories are treated in a more preservative fashion than the original liberal version. As for the fourth group, since `Map` cannot explicitly be iterated over, this category provides three methods that return collection views, each with its own iterator. The first method `keySet()` returns a `Set` view of the keys. The second method `values()` returns a `Collection` rather than a `Set`, due to the possibility of having duplicate values associated with different keys. The third method `entrySet()` returns a `Set` view of the mapping `key`⇒`value`. The elements of the returned set implement the interface `Map.Entry` that embodies a mapping from a key to a value. Since the owner of the elements of any collection class, in our design, is subject to context covariance, the

owner of the elements of the `Set` to be returned by `entrySet()` is a wildcard.

5.1.3 The `Iterator` Interface

```
public interface Iterator<Owner extends World, E
        extends IOwnedObject<?  extends World>
        extends IOwnedObject<Owner> {
    boolean hasNext();
    E next();
    void remove(); }
```

The `Iterator` interface is included in the package `java.util` together with the general-purpose collections. Any class that implements `Collection` maintains an `iterator()` method to return an `Iterator` object applicable to the instances of that class. The `iterator()` method is defined either inside the collection class itself or inside an abstract class (e.g., `AbstractList`) that provides a skeleton implementation of the relevant abstract data type (e.g., `List`).

Typically, iterators are implemented in the Collections Framework as inner classes implementing the `Iterator` interface; or as is the case with lists, iterators are inner classes that implement the interface `ListIterator`, which in turn extends `Iterator`. As clarified in the inner classes' subsection 4.2, iterators are implemented as inner classes in order to maintain direct access to the `private` backing fields of the outer collection classes (and thereof to the element objects). The nesting between classes is not a nesting between objects, however; hence `private` fields are exposed to incoming aliases to inner classes' instances. Implementing iterators as inner classes is basically the main issue in applying ownership to collections. Backing fields, as the internal private representation of their containers, must be owned by `This`; and therefore, cannot be directly accessed even through a `This`-owned reference defined in an inner class.

Consider the example of Fig. 5.10, `header` is a representation object of `LinkedList`, which means that `header` should be owned by `This`. `ListItr` is the iteration inner class of `LinkedList`. In the original implementation of `LinkedList`, `header` is assigned to `lastReturned` inside `ListItr`, as in line 7. If we are to make `lastReturned` owned by `This`, this does not mean that `header` and `lastReturned` have the same owner. `header` belongs only to the outer class, and can only be accessed through the operations of `LinkedList`. Within the course of the rest of this chapter, we will demonstrate how each collection class treats its iterators.

```
1  public class LinkedList<Owner extends World, E extends IOwnedObject<? extends
       World>> extends AbstractSequentialList<Owner,E> implements List<Owner,E>, ...
        {
2
3      private transient Entry<This, E> header = new Entry<This, E>(null, null, null
           );
4                                  ...
5      private class ListItr<Owner extends World> extends OwnedObject<Owner>
           implements ListIterator<Owner,E>{
6
7          private Entry<This, E> lastReturned = header;    //ERROR!!!
8                                  ...
9      }
10 }
```

Figure 5.10: Linked List's Iterator as an Inner Class

5.2 Lists

Lists are implemented through the interface `List`. `List` extends `Collection` and has a pretty similar header to `Collection`, see Fig. 5.11. `Owner` owns `List`, and `E` is the type of the elements.

```
1  public interface List<Owner extends World, E extends IOwnedObject<? extends World
       >> extends Collection<Owner,E> {
2
3      int indexOf(E o);
4      int lastIndexOf(E o);
5      List<Owner,E> subList(int fromIndex, int toIndex);
6
7      void add(int index, E element);
8      boolean addAll(int index, Collection<Owner, ? extends E> c);
9      E get(int index);
10     E set(int index, E element);
11     E remove(int index);
12
13     ListIterator<Owner, E> listIterator();
14     ListIterator<Owner, E> listIterator(int index);
15 }
```

Figure 5.11: OGJ$_+$ `List` Interface.

In order to provide search operations and range-views, `List` supports positional indexing so that the numerical position of a specified object (or a view of a range of the list) can be returned. The methods in lines 3-5, Fig. 5.11, support these operations. Since the nature of lists is to allow duplicates, `List` defines two methods for searching a specific element: the first one, in line 3, is to return the position of the first occurrence of the specified element; the second, in line 4, returns the last occurrence. Method `subList()`, in line 5, returns a range-view of the receiver list preserving owners. The original signatures of methods `indexOf()` and `lastIndexOf()` involve `Object` as a parameter type, so they were adapted to be more preservative.

Together with the methods inherited from `Collection`, the `List` interface defines additional position-oriented methods for inserting (collec-

tions as well as elements), retrieving, modifying and removing elements. These methods are in lines 7-11, Fig. 5.11. Except for `addAll()`, the other signatures did not require any modifications. What applies to `addAll()` in `Collection` applies to the position-oriented version of `addAll()` in `List`.

Finally, there are two iteration methods (lines 13 and 14, Fig. 5.11); each of them is returning a `ListIterator` over the receiver list. The `java.util.ListIterator` interface extends the `Iterator` interface. Due to the sequential nature of lists, it is possible to have bi-directional access to a list; that is, traversing a list backwards and forwards. So, in conjunction with the three main operations of `Iterator` (namely `hasNext()`, `next()` and `remove()`), `ListIterator` defines `hasPrevious()`, `previous()`, along with position-oriented methods such as `nextIndex()` and `previousIndex()`. Moreover, `ListIterator` defines `add()` and `set()` in order to facilitate insertions and replacements. All of these methods do not need any modifications in their signatures.

Back to the methods that return `ListIterator`, the first one is similar to `iterator()`, in `Collection`, in that it returns an iterator that is positioned at index 0. The second method returns an iterator that takes its initial position at the specified index. During the implementation, we found no need to make these methods owner-polymorphic as `Collection.iterator()`. We made them owner-polymorphic, however, so that their owner arguments can conform to the owner arguments of owner-polymorphic methods that might manipulate them.

5.2.1 Array List

`ArrayList` is a resizable version of an ordinary array. The `ArrayList` class is implemented as an array buffer in which the contents of the `ArrayList` are contained, and an integer field to keep track of the size of the list. These fields are referenced as `elementData` and `size`, respec-

tively. That is, the representation objects of an `ArrayList` are those referenced as `elementData` and `size`. The field `size` is of value type `int`, and we have no aliasing problems with value types. As for `elementData`, the array field to which it refers is of type `Object`, and it had to be substituted with an `OwnedArray` type. Since `elementData` is referring to a representation object, `OwnedArray` needs to be owned by `This` (i.e., to be private and hidden); which means that `elementData` can only be accessed via receiver `this`. Also, the type of the elements inside `OwnedArray` is no longer required to be other than `E` (originally `elementData` was of type `Object[]`), as we are no longer worried about Java's *no generic array creation* property, see subsection 4.1. The definition of `elementData` reads as follows:

```
private transient OwnedArray<This, E> elementData;
```

The `ArrayList` class provides no implementation for `iterator()` (defined in `Collection`) or for `listIterator()` (defined in `List`); instead, `ArrayList` inherits these implementations from the superclass `AbstractList`. `AbstractList` has two iteration inner classes; one implements `Iterator`, and the other implements `ListIterator`. Neither of these inner classes requires access to a representation field; `AbstractList` does not have fields. `AbstractList.iterator()` returns an implementation of the `Iterator` interface that depends entirely on three methods of the extending list (in this case `ArrayList`). These methods are `size()`, `get()`, and `remove()`. Likewise, `AbstractList.listIterator()` returns an implementation of the `ListIterator` interface that depends on the `get()`, `set()`, `add()`, and `remove()` methods of `ArrayList`. `ArrayList` encapsulates its representation, and thus the adaptation of `ArrayList` and `AbstractList` is just to make them compliant with OGJ_+.

5.2.2 Linked List

`LinkedList` is the other concrete implementation of `List` in this study. In contrast to `ArrayList`, which preserves encapsulation by depending entirely on the `List` interface, `LinkedList` preserves efficiency by having an iterator that bypasses the interface through an incoming reference to the private representation of `LinkedList`. That is, efficiency is preserved by breaching encapsulation. As depicted in Fig. 5.12, the entry nodes of the list are a number of objects placed into the heap with each of them having its own data plus pointers to the next and previous nodes; then, a `ListIterator`, as an outsider object, bypasses the `List` object's interface operations, and directly accesses the list's entries by pointing at any object in the chain.

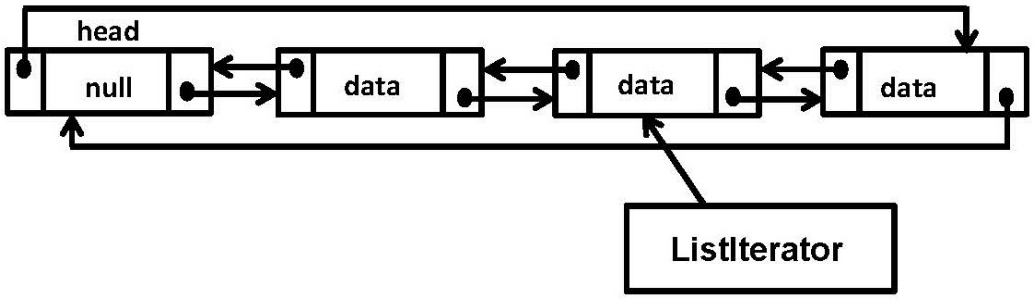

Figure 5.12: Linked List with Iterator

An entry node is implemented in `LinkedList` as a static member class called `Entry`, and contains three fields: `element`, `next`, and `previous`. `LinkedList` uses a list header that has the same formation as any other entry node. That is, to construct a new empty linked list, a new header object should be instantiated with a `null` element and pointers to itself, then new entry nodes can be linked to that header or inserted between each other. The `LinkedList` class is implemented as to have two fields: an `Entry` object referenced as `header`, and an integer `size` to keep track of the size of the list. These are our representation objects; and thus, `header` should be owned by `This`. The example of Fig. 5.13 shows the declara-

tions of the fields and constructors.

```
1  private transient Entry<This,E> header = new Entry<This,E>(null, null, null);
2  private transient int size = 0;
3  public LinkedList() {   // Constructs an empty list
4          header.next = header.previous = header;
5  }
6  // Constructor that takes a Collection argument
7  public LinkedList(Collection<Owner,? extends E> c) {
8      this();
9      addAll(c);
10 }
```

Figure 5.13: `LinkedList` Fields and Constructors

Similar to `ArrayList`, `LinkedList` inherits the implementation of `Collection.iterator()` from `AbstractList` to return an iterator for traversing the list from head to tail and safely removing entries. `LinkedList` also provides an implementation for `Deque.descending-Iterator()` to return an iterator for traversing the list from tail to head and remove entries. That is, `LinkedList.descendingIterator()` returns an implementation of the `Iterator` interface; this implementation is an inner class of `LinkedList` and does not maintain any incoming aliases as it provides iterators via `previous()` and `hasPrevious()` methods of the inner class implementation of `ListIterator`.

As explained earlier in subsection 5.1.3, `ListItr` is the inner class implementation of `ListIterator` in `LinkedList`. As the example in Fig. 5.10 shows, the representation object `header` is assigned to `lastReturned` when the iterator is initially constructed. What happens is that `lastReturned` gets information from the other incoming reference `next` that points to the current `Entry` node. Methods `next()` and `previous()` return `lastReturned` after setting it to `next` that moves along the linked nodes forward and backward by these methods. This will still be the case with `LinkedList` as an OGJ_+ compliant implementation, since the incoming references from `ListItr` to the internal representation

of `LinkedList` are wildcard-owned, and hence readonly.

`ListIterator` supports adding, removing and changing elements in the underlying list. The implementations of `add()`, `remove()`, and `set()` in `LinkedList.ListItr` is made through the operations of the `LinkedList`, and thus there were no significant changes required to preserve encapsulation in accordance with the rules of OGJ_+.

5.2.3 Vector (legacy)

`Vector` is the legacy version of `ArrayList`. The main difference between a `Vector` and an `ArrayList` is that the `Vector` class is synchronized. `Vector` has exactly the same internal representation as `ArrayList`. Also, `Vector` inherits the implementations of `Collection.iterator()` and `List.listIterator()` from `AbstractList`, where the returned iterators are implemented as inner classes. So, in relation to these implementations, `Vector` maintains the same encapsulation guarantees as `ArrayList`.

Additionally, `Vector` provides the method `elements()` that returns an implementation of the legacy interface `Enumeration`. `Enumeration` iterates over the implementor collection for retrieval of stored elements. `Enumeration` is implemented in `Vector` as an anonymous class. An anonymous class can be declared in OGJ$_+$ as a sibling of the enclosing class. That is, the instances of `Vector` and the sibling implementation of `Enumeration` are located in the same ownership context, but `Vector` and its internal representation form another nesting context, as depicted in Fig. 5.14. Accordingly, `Enumeration` has no right to directly access the internal representation of `Vector`. The example in Fig. 5.15 shows how the `Enumeration` interface can be implemented as an anonymous class owned by the same owner of as `Vector`

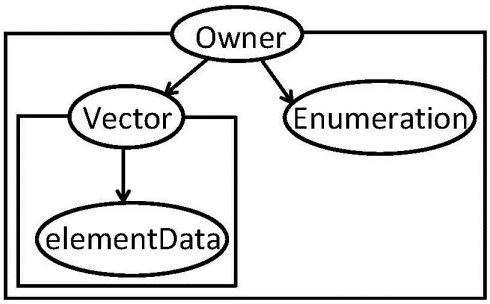

Figure 5.14: An object of an anonymous class as a sibling of the enclosing object

```
1  public class Vector<Owner extends World, E extends OwnedObject<? extends World>>
       extends AbstractList<Owner, E> implements List<Owner, E>, ... {
2
3      OwnedArray<This, E> elementData;
4                                  ...
5      public Enumeration<Owner, E> elements() {
6          return new Enumeration<Owner,E>() {
7                                  ...
8          }
9      }
10 }
```

Figure 5.15: `Enumeration` implemented as an anonymous class in the same context as `Vector`

`Enumeration` provides two methods (`hasMoreElements()` and `nextElement()`) for traversing a `Vector` object. Method `nextElement()` returns the element at the relevant position by directly accessing the backing array, which means that the elements can be retrieved without the `Vector`'s knowledge. To correct this breach, that OGJ$_+$ does not allow, `nextElement()` was modified to depend on the `get()` methed of the current `Victor` object.

5.3 Queues

Applying generic ownership to the interfaces `Queue` and `Deque` (double-ended queue) is no different to the interfaces explained above. All modifications can be justified on the premises mentioned in subsection 5.1.1. This section describes the issues addressed in two implementations of these interfaces. `PriorityQueue` implements `Queue`; and `ArrayDeque` implements `Deque`.

5.3.1 Priority Queue

`PriorityQueue` is backed by an array. The representation objects of this

implementation are quite similar to those of `ArrayList` and `Vector`: an array buffer to hold the contents of the queue; and an integer for keeping track of the size of the queue. This implementation maintains its iterator as an inner class, as mostly the case with collections; and as commonly happens, a `next()` method directly accesses the backing array. In `Vector`, we modified `nextElement()` to depend on the `get()` method, which is defined within the body of the `Victor` class. Due to the `FIFO` nature of queues, the `Queue` interface does not provide any position-oriented methods for retrieving elements form the backing array. To overcome this, we defined a `private` method within the body of `PriorityQueue` to get the element at the specified index, and made `next()` rely on this `private` method.

5.3.2 Array Deque

As its name implies, the `ArrayDeque` implementation is also backed by an array. Since a deque supports insertion, retrieval, and removal of elements at either ends of the queue, `ArrayDeque` has two representation integer fields, namely `head` and `tail`, to keep track of the indices of the elements at the head and tail of the deque. `ArrayDeque` is no different to `PriorityQueue`, however, in that it does not provide element retrieval via position-oriented methods.

This implementation maintains two iterators, namely `DeqIterator` and `DescendingIterator`, also as inner classes. Each of these iterators has a `next()` method to return the element at the cursor position. Both methods directly access the backing array on two occasions: first, to get the relevant element; and second, to get the length of the array in order to calculate the next cursor position. Moreover, the `remove()` method in both iterators accesses the array to get its length in order to recalculate the cursor position, so that the increment made by `next()` can be undone. In a similar fashion to `PriorityQueue`, both iterators were refactored to

relay on two `private` methods defined within the body of `ArrayDeque`; one to return the element at the specified index, and the other to return the array length.

5.4 Maps

Although a map can be thought of as a special kind of set of values associated with unique keys (one-directional mapping from a key to a value) the `Map` interface does not extend `Set` or `Collection`. On the other hand, the `Set` interface general-purpose implementations are backed by `Map` implementations in order to make use of the properties of these `Map` implementations. For example, the `HashSet` implementation does not provide a specific `Iterator`; instead, the method `HashSet.iterator()` returns an iterator over the key set view of the backing `HashMap`. Since it is not possible to iterate over a `Map` object directly, `Map` provides set views. That is, the relationship between the notions of sets and maps forms a key element in their design and implementation. So, why does `Map` not extend `Collection`? Further, why does `Set` not extend `Map`? The answer to these questions are best described in Java Collections API Design FAQ [2]:

> *"This was by design. We feel that mappings are not collections and collections are not mappings. Thus, it makes little sense for `Map` to extend the `Collection` interface (or vice versa). If a `Map` is a `Collection`, what are the elements? The only reasonable answer is "Key-value pairs", but this provides a very limited (and not particularly useful) `Map` abstraction. You can't ask what value a given key maps to, nor can you delete the entry for a given key without knowing what value it maps to. `Collection` could be made to extend `Map`, but this raises the question: what are the keys? There's no really satisfactory answer, and forcing one leads to an unnatural interface. `Map`s can be viewed as `Collection`s (of keys, values, or pairs), and this fact is reflected in the three "`Collection` view operations" on `Map`s (`keySet`, `entrySet`, and `values`). While it is, in principle, possible to*

view a `List` *as a* `Map` *mapping indices to elements, this has the nasty property that deleting an element from the* `List` *changes the* `Key` *associated with every element before the deleted element. That's why we don't have a map view operation on* `Lists`*.*".

So, in terms of design, conformance, and coding, treating maps as `Collection` implementations (i.e., `Map` **`extends`** `Collection`) would definitely hinder the flexibility of implementing maps, as well as the performance, resulting in an inefficient `Map` abstraction. Treating collections as `Map` implementations is not reasonable in terms of handling the contents, properties, and solution design options. Maps are set containers, however; mathematics says so, and even the code says so.

The Collections Framework provides three general-purpose `Map` implementations: `HashMap`, `LinkedHashMap`, and `TreeMap`. This study is concerned with these implementations of `Map`, the legacy implementation `HashTable`, and the special-purpose implementation `IdentityHashMap`.

5.4.1 Hash Map

`HashMap` is a `Map` implementation backed by an array. The contents of this array are the mapping entries of the table that makes up the internal representation of `HashMap`. Each mapping entry is an instance of the static inner class `HashMap.Entry`. In the `List` and `Queue` implementations, the routine was to hide the backing `OwnedArray` objects by making them owned by `This`, since they hold the elements. In `LinkedList`, the `Entry` objects, which each hold an element, needed to be hidden, since they are consistently available as entry points. With `HashMap` we need to hide both the backing array as well as the `Entry` objects that hold the key-value mappings. So, the backing field `table` would be declared as follow:

```
OwnedArray<This, Entry<This, K, V>> table;
```

As explained in subsection 5.1.2, `Map` provides methods that return iterable collection views. These methods are `keySet()`, `values()` and

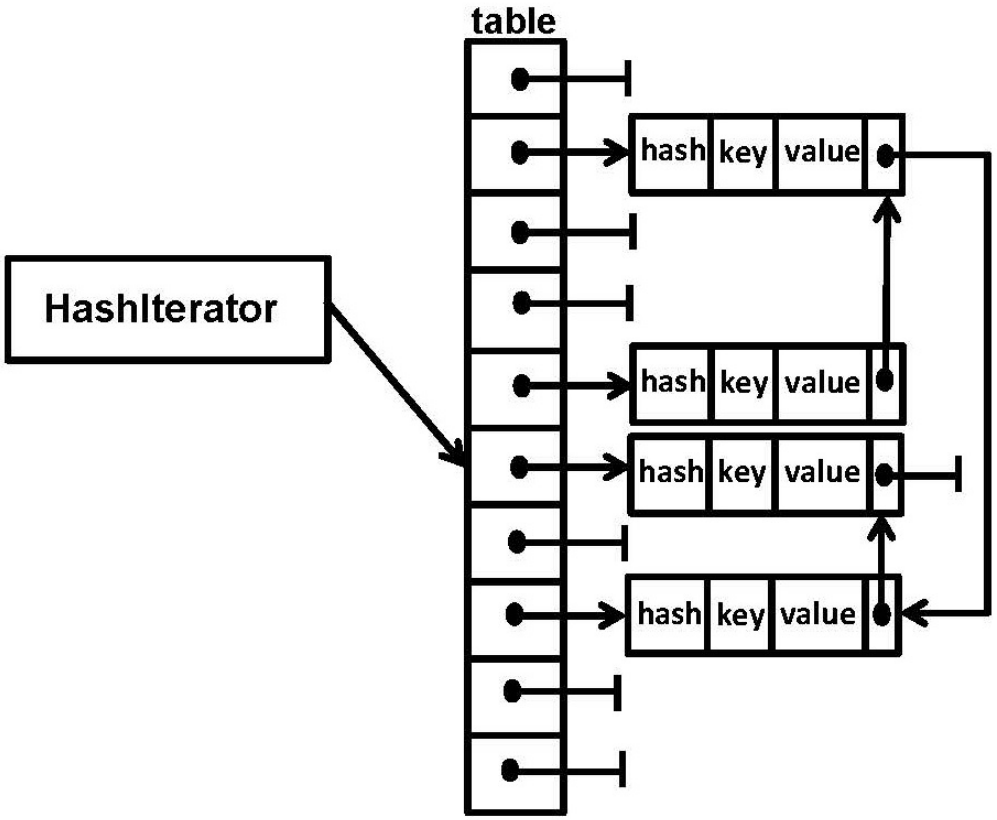

Figure 5.16: Hash Map with Iterator

`entrySet()`; their signatures appear in Fig. 5.9. The views are backed by the current map; that is, changes in the map are reflected in the views, and vice-versa. To maintain this mirroring property while allowing each view's iterator to support entry removal, each iterator needs to run over the indexed mapping entries of the backing array. In doing so, `HashMap` provides three iteration inner classes, each for a view, as adapters that provide iterators via `HashIterator.nextEntry()`. `HashIterator` is a `private abstract` inner class implementation of the `Iterator` interface, and does not override the abstract method `Iterator.next()`. Each of the three adapters extends `HashIterator` in order to override `Iterator.next()` in behalf of `HashIterator`. Each adapter's

overriding `next()` relies on the indexed mapping entry returned by the `nextEntry()` method. The `nextEntry()` method and the constructor of `HashIterator` maintain incoming references to `table`. Through these reference, `nextEntry()` accesses `table` to get its length, and runs another pointer, `next`, over the table entries to get, and return, the `Entry` object at the specified index.

To refactor this breach, we removed the incoming reference to `table`, and kept `next` as a wildcard owner parameterised reference, so that `next` can point to the `Entry` objects which are owned by `This` (i.e., the `this HashMap`). Since accessing the table needs to be done through the `HashMap` operations, we need to find methods that return indexed mapping entries and the table length. `HashMap` has a `capacity()` method which returns the length of the backing array. `HashMap` treats `table` as indexed by keys, and hence has no methods that allow dealing with `table` as indexed array. We defined a `private` method inside `HashMap` to return the `Entry` object at the specified index, and made `next` point to the returned entry.

5.4.2 Linked Hash Map

The `LinkedHashMap` implementation inherits from `HashMap`. `LinkedHashMap` maintains a doubly-linked list over the mapping entries to be stored in `HashMap.table`. `LinkedHashMap.Entry` inherits from `HashMap.Entry`, and provides additional pointers to the next and previous `Entry` objects to form the required nodes for a linked list. Similar to `LinkedList`, `LinkedHashMap` uses a list header, which is an entry node with a `null` mapping. That is, an `Entry` object referenced as `header` is our representation object that should not be accessed via incoming aliases.

The collection views' iterators are implemented in `LinkedHashMap` in the same fashion as in `HashMap`. The views' iterators inherit from the `private abstract` inner class `LinkedHashIterator`, which correspond to `HashIterator` in `HashMap`. `LinkedHashIterator` maintains an incoming reference to the representation field `header`. This reference must be owner parameterised with a wildcard, so that it can safely be set to `header`.

Since lookups are based on unique keys, the implementation of `HashMap` uses a hash to store the key in the map for fast lookups. Therefore, a `HashMap` does not guarantee the order of the mapping entries. By maintaining a doubly linked list, `LinkedHashMap` maintains the order of the entries according to the order in which they are inserted. Moreover, `LinkedHashMap` can maintain the ordering according to the order in which the entries are accessed. In doing so, `LinkedHashMap` provides a constructor that takes a `boolean` parameter to switch between the ordering modes. This `boolean` parameter is then assigned to the representation field `accessOrder`. Apart from this constructor, for every constructor declared in `HashMap`, `LinkedHashMap` declares a corresponding constructor. All of the constructors of `HashMap` are invoked by `super()` in their corresponding constructors of `LinkedHashMap`, and `accessOrder` is set to `false`. In order for `LinkedHashMap` to initialize the header node, the *Template Method design pattern* is adopted. This pattern requires two kinds of methods, one is called a *template method*, and the other is called a *hook* (or placeholder) method. The template method (in a superclass) contains the invariant behaviour and invokes the hook (in a subclass) for the variant behaviour. All of the constructors of `HashMap` are template methods that call a hook called `init()`. Method `HashMap.init()` does not hold any behaviour on its own, while the overriding `LinkedHashMap.init()` initializes the header node of the linked list. When the constructors of `LinkedHashMap` call `super()`, the header node is initialised via `LinkedHashMap.init()`.

In a similar fashion, methods `put()` and `putForNullKey()` are implemented in `HashMap` as template methods that invoke the hook `recordAccess()` which is declared in `HashMap.Entry`. This hook contains no behaviour. The overriding hook `recordAccess()` of `LinkedHashMap.Entry` moves the receiver mapping entry to the end of the list, if `LinkedHashMap` is access ordered (i.e., `accessOrder` flags `true`). In doing so, `recordAccess()` inserts the entry before the header. That is, `recordAccess()` needs to access both `accessOrder` and `header`. Class `Entry` is a static inner class from which non-static variables cannot be referenced. That is why `recordAccess()` takes a `HashMap` parameter. This parameter is then assigned to a `LinkedHashMap` local variable, through which the representation fields can be accessed; see the example in Fig. 5.17.

```
1 void recordAccess(HashMap<K,V> m) {
2     LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;
3     if (lm.accessOrder) {
4         lm.modCount++;
5         remove();
6         addBefore(lm.header);
7     }
8 }
```

Figure 5.17: Java 1.6 `LinkedHashMap.Entry.recordAccess()`

OGJ$_+$ does not permit the field access in line 6, Fig. 5.17, since `header` is owned by `This`. To overcome this restriction, we defined a `private` getter method inside `LinkedHashMap` to return a wildcard owner parameterised reference to the first entry, so that the statement in line 6 reads as `addBefore(lm.getFirstEntry().before)`. This required `addBefore()` to take a parameter of an unknown context, and to make the entry nodes' pointers (`after` and `before`) also wildcard owner parameterised. This solution worked for the above mentioned conjunction between the static context and the template method pattern.

The OGJ$_+$ approach to wildcards did not work, and should not, for a hook such as `transfer()`, whose behaviour is to move the contents of the backing array to another resized array. The resizing template method declared in `HashMap` assigns the new array object to `table` after invoking `transfer()`. This means that the new resized array and its entry objects should be owned by `This`. The overriding `transfer()` of `LinkedHashMap` makes use of the linked list for faster iteration. If `after` and `before` are of unknown context, then we cannot assign them to entry objects owned by `This`. We decided to keep the wildcard solution for `recordAccess()` and rely on the `transfer()` method for `HashMap`.

5.4.3 Hash Table (legacy)

`Hashtable` is the legacy version of `HashMap`. The main difference between `Hashtable` and `HashMap` is that `Hashtable` is synchronized. `Hashtable` maintains similar representation objects as `HashMap`. That is, `Hashtable` is a `Map` implementation backed by an array. The mapping entries of the backing array, `table`, are instances of the private static class `Entry`. The definition of `table` reads exactly as that of `HashMap`; `OwnedArray` is owned by `This`, and the `Entry` objects should also be owned by `This`.

`Hashtable` extends the abstract parent `Dictionary`, which provides the methods `keys()` and `elements()` that return enumerations of the keys and values, respectively. The returned enumeration objects are implementaions of the legacy interface `Enumeration`. This way, `Dictionary` provides methods for traversing the keys and values, but not the entry mappings; and `Enumeration` traverses a `Hashtable` object without modifications. `Hashtable` provides the private inner class `Enumerator` which implements both the `Enumeration` and `Iterator` interfaces. This way, `Enumerator` can traverse a `Hashtable` object as a dictionary (associative array) and as collection views. `Hashtable` pro-

vides key set, value collection, and entry set views, but none of them has its own implementation of `Iterator`. `Enumerator` has a constructor that takes two parameters; an integer parameter and a boolean parameter. `Enumerator` serves as an iterator if the boolean argument is `true`. The integer argument can be set to 0, 1, or 2, for traversing keys, values, or entries, respectively. Apparently, there is no use for an `Enumerator` such that: `new Enumerator(2, false)`.

The `Enumerator.next()` method (overrides `Iterator.next()`) relies entirely on the `Enumerator.nextElement()` (overrides `Enumeration.nextElement()`). The overriding `next()` method calls `nextElement()` and returns the value that `nextElement()` returns. The return type of `nextElement()` is of type `<T>`, which is declared in the header of `Enumerator` as the type of elements, and extends `IOwnedObject<? extends World>`. Method `nextElement()` returns an element such that:

```
type == 0 ? (T)e.key : (type == 1 ? (T)e.value : (T)e)
```

Variable `type` denotes the value of the constructor integer argument, while `e` is a wildcard owner parameterised reference to the current `Entry` object which is owned by `This`. OGJ$_+$ accepts the first two casts, since the owner of the type variable (`K` for keys, or `V` for values) is the same owner of the elements that the `Enumerator` instance is to traverse over. In fact, `Enumerator` is to be instantiated using the very same type variable (`K` or `V`). Per contra, there is no way to allow the third cast in OGJ$_+$. Although it is possible to assign an object owned by `This` to a readonly wildcard owner parameterised reference, it is not possible to allow a cast using incompatible owners. Therefore, we had to provide a special implementation of `Iterator` for the entry set views.

5.4.4 Identity Hash Map

`Map`'s general contract mandates the use of the `equals()` method to determine equality using the state of the objects. In violation to this contract, `IdentityHashMap` performs identity-equality comparisons to determine uniqueness. A key is considered equal to another key only if they are references to the same object. Since the object state is disregarded, an `IdentityHashMap` object can contain key values and mappings that would be dealt with as duplicates in other `Map` implementations. In addition to this distinct property, `IdentityHashMap` is also different in that it uses a technique called *hashing with linear probing* [53]. This technique is used to deal with hash collisions of values returned by the `hash()` method. `IdentityHashMap` uses this technique to sequentially search the hash table for a free pair of adjacent locations, since `IdentityHashMap` stores the key and value references directly in adjacent locations in the table, not in entry cells; the key is placed at `table[hash]`, while the value is placed at `table[hash+1]`. That is, `IdentityHashMap` does not use `Entry` objects, and hence the backing field `table` would be declared as follows:

```
OwnedArray<This,IOwnedObject<? extends World>> table;
```

In addition to the key set and value collection views, `IdentityHashMap` provides entry set views, however. Since `IdentityHashMap` does not maintain entry cells, the iterator of the entry set view is itself used as an entry that extends `IdentityHashMapIterator` and implements `Map.Entry`. `IdentityHashMapIterator` is similar to `HashIterator` and `LinkedHashIterator` in that it is an abstract inner class implementation of the `Iterator` interface, and does not override the abstract method `Iterator.next()`. The methods which override the abstract methods of `Map.Entry` rely on an incoming reference, defined in `IdentityHashMapIterator`, to `table`. This reference had to be owner parameterised by a wildcard in order to safely be set to the `This`-owned `table`.

Each view iterator overrides `Iterator.next()` relying on the indexed key returned by `IdentityHashMapIterator.nextIndex()`. Determining the initial index (0 if `table` is not empty, or table length if `table` is empty) requires access to `table` to get its length. That is, an incoming reference to `table` is maintained. To maintain access to `table` through the `IdentityHashMap` operations, we defined a private method within the body of `IdentityHashMap` to return the table length, and made the iterator depend on this method.

5.4.5 Tree Map

The Collections Framework of Java 1.6 has the new interface `NavigableMap`. The `NavigableMap` interface is a subtype of the `SortedMap` interface. `TreeMap` is the only general purpose collection that implements `NavigableMap`. As an implementation of `NavigableMap`, `TreeMap` provides navigation methods such as `descendingKeySet()`, `descendingMap()`, `headMap()`, `tailMap()` and `subMap()`. That is, `TreeMap` provides more views than the ordinary views (i.e., `KeySet`, `Values` and `EnterySet`); for example, `AscendingEntrySetView`, `DescendingEntrySetView`, `AscendingSubMap`, `DescendingSubMap`; each has its iterator. A lot of operations are involved; submaps and entries are defined more precisely; as a result, many different inner classes (static and non-static) had to be involved.

Except for `LinkedList`, all of the `Collection` and `Map` implementations that have been studied up to this subsection are backed by arrays, and we used to make the backing arrays owned by `This`. Except for `IdentityHashMap`, all of the other hash-based `Map` implementations use entry cells to hold the mappings, and these cells had to be owned by `This`. With `LinkedList` and `LinkedHashMap`, the list header is fixed and has a `null` element or `null` mapping. With `TreeMap`, the root is neither fixed nor having a `null` mapping. The root is part of the tree and is the mid-

dlemost element. The mapping entry that represents the root of the tree could be swapped through right and left rotations as we perform add and remove operations. That is, the backing field of `TreeMap`, namely `root`, should always be available to be set to a mapping entry. If we are to have `root` owned by `This` while changing the reference structure through rotations, then all of the references to the entry nodes which are involved in a rotation operation must be owned by `This` and declared within the same class as `root`. Because of the many different inner classes involved in the implementation of `TreeMap`, and because the entry nodes must be accessed through the operations of `TreeMap`, inner classes need to send messages to `TreeMap` by passing references to entry nodes as arguments. These references had to be owner parameterized by wildcards, and most of the operations of `TreeMap` had to be able to accept argument types that have wildcards as owner parameters. That is, `root` needs to be owner parameterized by a wildcard in order for the rotation operations to set `root` to mapping entries. Since all entry nodes must be owned by `This`, we made sure that every new entry node is initialized within an insertion method as follows:

```
Entry<This,K,V> e =
   new Entry<This,K,V>(key, value, parent);
```

Since all of the entries that make up the internal representation of `TreeMap` is to be initialized owned by `This`, we can safely declare the backing field `root` as follows:

```
private Entry<?  extends World, K, V> root;
```

5.5 Sets

Sets are implemented through the interface `Set`. `Set` extends `Collection`, overrides all of the methods of `Collection`, and does not provide more methods than `Collection` does. This study is concerned with three implementations of `Set`, namely `HashSet`, `LinkedHashSet` and

`TreeSet`. These sets are backed by `Map` implementations, and can be iterated over through the key iterators of their respective backing maps. The implementation of these sets is quite simple, and hence the refactoring is quite straightforward.

5.5.1 Hash Set

`HashSet` is backed by a `HashMap` instance referenced as `map`. `HashSet` relies only on the key store maintained by the backing `HashMap` to prevent duplicates. All mapping values are set to dummy values of type `Object`.

Since we made sure that the backing table of `HashMap` and the contained entry cells are hidden, then the elements of `HashSet` (i.e., the keys of the backing map) are also hidden whatever the owner of the backing field `map` is. We made `map` owned by `This`, however. Since the signature of `Set.iterator()` is owner polymorphic, the overriding method `HashSet.iterator()` can properly return the type returned by the call `map.keySet().iterator()`. The backing field `map` is declared as follows:

```
HashMap<This, E, IOwnedObject<? extends World>> map;
```

The type argument of `map`'s keys is `E`, which is the type of `HashSet`'s elements; and the type argument for the dummy values is the OGJ$_+$'s equivalence of `Object`.

5.5.2 Linked Hash Set

The `LinkedHashSet` implementation inherits from `HashSet`. The `LinkedHashSet` class has no backing field and has only four constructors; no other operations are defined in this class. All of the operations that a `LinkedHashSet` instance can conduct are inherited from `HashSet`. The four constructors of `LinkedHashSet` invoke only one constructor from `HashSet`. The invoked constructor form `HashSet` is made especially for `LinkedHashSet` and instantiates `LinkedHashMap`, so that the opera-

tions of HashSet can be conducted on a LinkedHashMap instance rather than a HashMap instance. Therefore, there are no security issues with this implementation.

5.5.3 Tree Set

In the same manner as described for TreeMap, TreeSet is the only general purpose collection that implements the NavigableSet interface. NavigableSet is a subtype of the SortedSet interface. That is, in addition to the sorting mechanisms, TreeSet provides the same navigation mechanisms as that of TreeMap, but only on the key store of the backing TreeMap. The backing field of TreeSet is of type NavigableMap, however. This is mainly because some of the navigation methods of TreeMap return NavigableMap views, and the corresponding navigation methods in TreeSet are made simple as to return new instances of TreeSet backed by these NavigableMap views.

As described for HashSet, there is no need to make the backing field owned by This since the elements are properly encapsulated in the key store of the backing map. We made the backing map of HashSet owned by This, and this did not cause any owner compatibility problems. For TreeSet, we decided to avoid making the backing map owned by This in order to avoid any owner compatibility issues between TreeSet and TreeMap. The backing field is declared as follows:

```
NavigableMap<Owner,E,IOwnedObject<? extends World>> m;
```

The backing map is owned by Owner which denotes the owner of TreeSet. The type argument of the keys is E, which is the type of TreeSet's elements; and the third type argument is for the dummy values.

5.6 Usability

Using the refactored collections requires a significant programming overhead, as appears in Fig. 5.19 which is the OGJ$_+$'s version of Fig. 5.18. Although OGJ$_+$ provides stronger encapsulation guarantees, it is hard to believe that productivity will not decline using our version of the Collections Framework. Using OGJ$_+$, and in particular the refactored collections, requires increased effort since, in addition to the annotation overheads, programmers will have to deal with the fact that standard code reuse practices became entangled by the use of deep ownership types.

```
1  public class Glossary{
2      public static void main(String[] args) {
3          Map<String, List<String>> map = new HashMap<String, List<String>>();
4
5          List<String> a = new ArrayList<String>();
6          a.add("Academy");
7          a.add("Anatomy");
8
9          List<String> b = new ArrayList<String>();
10         b.add("Balloon");
11         b.add("Burger");
12
13         List<String> c = new ArrayList<String>();
14         c.add("Cadbury");
15         c.add("Common");
16
17         map.put("A", a);
18         map.put("B", b);
19         map.put("C", c);
20
21         System.out.println("Get keys and their corresponding values:");
22         for (Map.Entry<String, List<String>> entry : map.entrySet()) {
23             String key = entry.getKey();
24             List<String> values = entry.getValue();
25             System.out.println("Key = " + key);
26             System.out.println("Values = " + values);
27         }
28     }
29 }
```

Figure 5.18: A `HashMap` storing multiple values for the same key using `ArrayList`.

```
1  public class GlossaryOGJ<Owner extends World> extends OwnedObject<Owner>{
2      static { begin(); }
3      public static void main(OwnedArray<World, OwnedString<World>> args) {
4          Map<World, OwnedString<World>, List<World, OwnedString<World>>> map =
5                  new HashMap<World, OwnedString<World>,
6                              List<World, OwnedString<World>>>();
7
8          List<World, OwnedString<World>> a =
9                  new ArrayList<World, OwnedString<World>>();
10         a.add(new OwnedString<World>("Academy"));
11         a.add(new OwnedString<World>("Anatomy"));
12
13         List<World, OwnedString<World>> b =
14                 new ArrayList<World, OwnedString<World>>();
15         b.add(new OwnedString<World>("Balloon"));
16         b.add(new OwnedString<World>("Burger"));
17
18         List<World, OwnedString<World>> c =
19                 new ArrayList<World, OwnedString<World>>();
20         c.add(new OwnedString<World>("Cadbury"));
21         c.add(new OwnedString<World>("Common"));
22
23         map.put(new OwnedString<World>("A"), a);
24         map.put(new OwnedString<World>("B"), b);
25         map.put(new OwnedString<World>("C"), c);
26
27         System.out.println("Get keys and their corresponding values:");
28         for (Iterator<World,
29                       Map.Entry<? extends World, OwnedString<World>,
30                       List<World, OwnedString<World>>>> entry =
31                 map.entrySet().iterator(); entry.hasNext(); ) {
32             Map.Entry<? extends World, OwnedString<World>,
33                       List<World, OwnedString<World>>> e = entry.next();
34             OwnedString<World> key = e.getKey();
35             List<World, OwnedString<World>> values = e.getValue();
36             System.out.println("Key = " + key);
37             System.out.println("Values = " + values.toOwnedString());
38         }
39     }
40 }
```

Figure 5.19: OGJ$_+$'s version of Fig. 5.18.

Chapter 6

Evaluation

To evaluate the refactored collections in terms of performance, we conducted some micro-benchmark tests comparing the performance time of the original and refactored collections.

As explained in the previous chapter, all class headers of the refactored collections are no longer compatible with those of the JDK 1.6 collections; all classes must involve owner parameters; all type variables are bounded by `IOwnedObject`; `Object` is no longer permitted as a type in OGJ$_+$; a significant number of method signatures have been changed; Java's class types, such as `String` and `Integer`, have been wrapped in owned classes; Java arrays are no longer permitted. As a result, any benchmark suite that is intended as a tool for Java benchmarking needs to be refactored, in order to benchmark our refactored collections.

We found that refactoring a macro-benchmark, such as DaCapo or SPECjvm2008, would be another big project; and in the end, we do not expect that collections form a considerable share of their execution. Accordingly, we decided to modify four of Doug Lea's JRS-166 [33] micro-benchmarks that were specially implemented to target the Collections Framework. These micro-benchmarks are `IteratorLoops`, `CollectionLoops`, `MapLoops` and `MapMicroBenchmark`. The latter is in fact implemented to represent realistic uses of the `Map` classes. Since

Doug Lea is a co-developer of the Collections Framework, this set of micro-benchmarks is considered the best for experimenting the collections' performance.

All benchmarks were modified to be compatible with the class headers and method signatures of the OGJ$_+$'s collection classes. Class types (e.g., `Integer`) were replaced with their corresponding types in OGJ$_+$. The modified micro-benchmarks were used along with the original ones to compare the performance of our refactored collections against the JDK 1.6 collections. Some changes have been applied to the behaviour of both the OGJ$_+$ compliant benchmarks and the original benchmarks. These changes are explained in detail, where applicable, in the following sections.

All benchmarks were run using the Java Platform JDK 1.6.0_30 (Java HotSpot 64-Bit Server VM 20.5-b03) on the NetBeans IDE 7.1 (Build 201112071828). The computer used has a 64-bit Intel chip (Core i3 at 2.13 GHz) and 4GB of RAM, running a 64-bit operating system (Microsoft Windows version 8). In the rest of this chapter, we present some experimental results.

6.1 `IteratorLoops`

`IteratorLoops` measures the performance time for each step of the iteration process. `IteratorLoops` can test `Collection` classes as well as `Map` classes. The estimates are provided in nanoseconds per each iteration step; that is, the time needed to execute a single `next()` operation call. `IteratorLoops` runs 268,435,456 iteration steps over an array of 16 `Collection` objects of size 16384 (the square root of the iteration steps). `IteratorLoops` intermittently adds about 1/8 of the preloaded elements at run-time.

Fig. 6.1 shows the estimates for all of our refactored collections and maps against the original ones. The plots in Fig. 6.1 are the average of four trials per run per collection class. This is the default number of trials `IteratorLoops` produce. We executed `IteratorLoops` several times

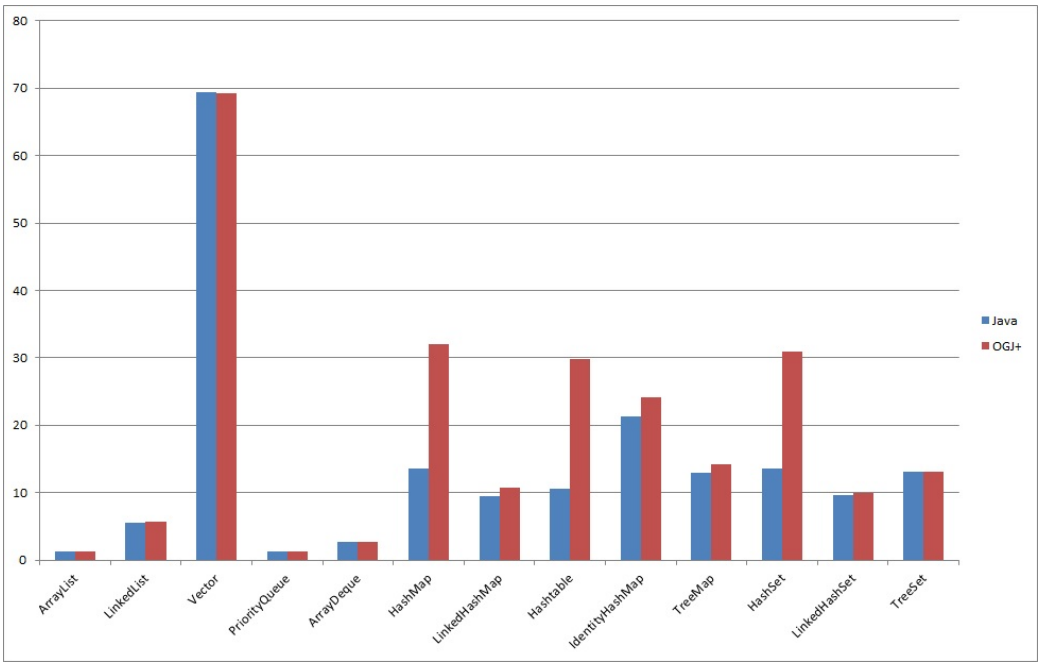

Figure 6.1: `IteratorLoops` – Nanoseconds per Iteration Step

for each collection class and found that the produced numbers are relatively stable.

Except for `HashSet`, all of the other refactored `Collection` classes show no variance of performance degradation. As for the `Map` classes, `LinkedHashMap`, `IdentityHashMap` and `TreeMap` show no significant variance (1-3 nanoseconds) of performance degradation. The performance of `Hashtable` and `HashMap` (and hence `HashSet`) is around 2.5 times slower. This is mainly because of the way the `next()` method works. In `HashMap`, the backing array of entries is not a contiguous store; there could be `null` values in between. The `next()` method performs a sequential walk over the backing array until a non-`null` value is found. The walking loop used by the `next()` method of the original `HashMap` is as follows:

```
while (index < table.length &&
           (next = table[index++]) == null);
```

Java keeps type information at runtime for array types. Java arrays are eliminated in OGJ$_+$ that uses the generic type `OwnedArray`. With generic types, type information is not kept at runtime; the compiler inserts `checkcast` instructions (The bytecode instructions for performing casting at runtime). That is, every walking step performed by the `next()` method of the refactored `HashMap` requires a `checkcast` to check if the object extracted from `OwnedArray` could be cast to the type of `next`. This should increase the time needed to execute the `next()` operation. The same applies to `Hashtable`.

6.2 CollectionLoops

`CollectionLoops` can test only `Collection` classes. `CollectionLoops` is originally implemented to exercise multi-threaded collections; but for an unknown reason, some `Collection` classes (original and refactored) cannot complete the test. We modified `CollectionLoops` to exercise single-threaded collections. A thread does a random walk over an array of 10,000 elements. On each iteration, `CollectionLoops` checks if the collection contains the given element. If the element is absent, `CollectionLoops` adds it. If the element is present, `CollectionLoops` removes it. `CollectionLoops` performs 100,000 operations per trial. A trial measures the performance time for each operation in nanoseconds.

By default, `CollectionLoops` runs four trials for single-threaded collections. With `Set` classes (refactored and original), we noticed that the numbers of the first two trials are significantly higher than those of the last two trials; and the numbers of the first two trials significantly vary from run to run. We decided to loop the four trials 25 times, and the output was 52 trials. We modified `CollectionLoops` to calculate the average of the 52 results. With the `HashSet` and `LinkedHashSet`, we found after several runs that the averages vary from 65 to 85 nanseconds per operation, and more than 90% of the 52 results are under or between

these two averages. Proportionally, the same happens with `TreeSet`. Although the other `Collection` classes produce stable results, we kept `CollectionLoops` with these modifications, and tested all of the collection classes. For the `Set` classes, the average of the best five repetitions was calculated to represent the performance time per operation.

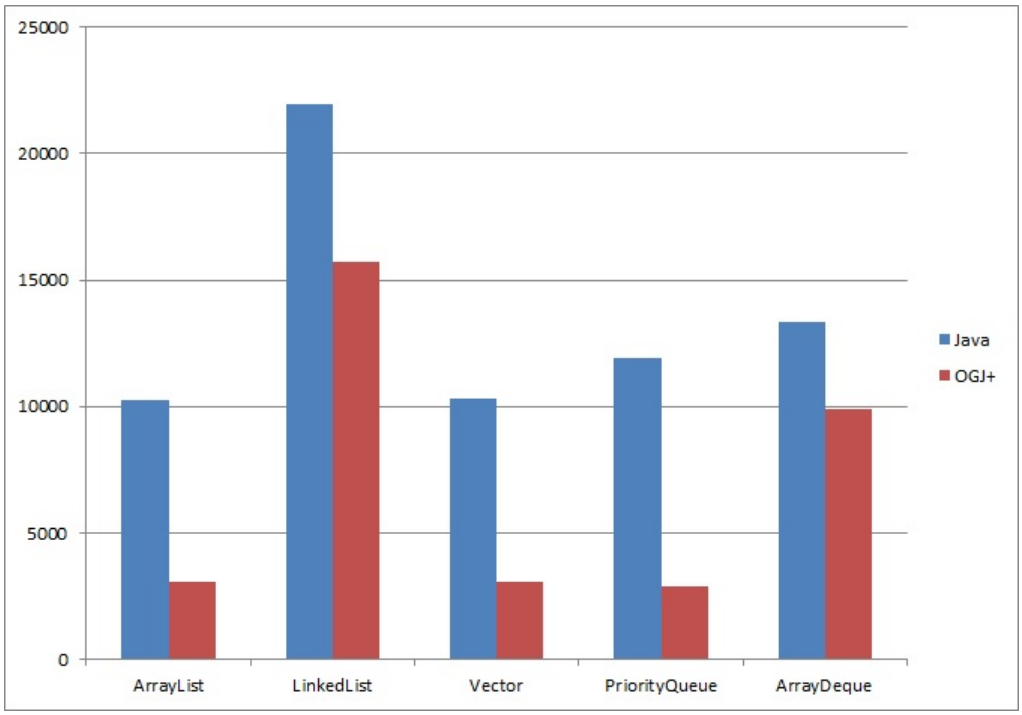

Figure 6.2: `CollectionLoops` for Lists and Queues – Nanoseconds per Operation

Fig. 6.2 shows the results for the lists and queues. There is a significant improvement in the performance of the refactored classes. This is because there is a significant decline in the number of implicit type casts. To explain this, we first recall that the original `ArrayList`, `Vector` and `PriorityQueue` are backed by arrays of type `Object`, while the corresponding refactored versions are backed by `OwnedArray` instances that store objects of type `E`, where `E` is the type variable of the collection class. `ArrayDeque` is originally backed by an array of type `E`, and the nodes of

`LinkedList` originally hold elements of type `E`. Note that the improvement in the performance of the refactored versions of the latter two implementations is not as great as that of the other three implementations. In the five classes, all of the methods that originally take `Object` as an argument were refactored to take `E` as an argument (e.g., `contains()`, `indexOf()`, `remove()`). `CollectionLoops` uses `contains()` very heavily. The implementation of `contains()`, in any of the five collections, checks if the specified object (method argument) is contained in the current collection by iterating over all of the stored objects to see whether one of them is equivalent to the specified object using the `equals()` method. Except for `ArrayDeque`, `contains()` delegates lookups to `indexOf()`. The `indexOf()` method (or `contains()` in case of `ArrayDeque`) walks the backing array (or the linked list) using a for loop. Each step invokes `equals()`. If both the receiver and the argument of `equals()` are of type `E`, then the number of the required implicit casts will be less than the number of implicit casts required to complete the operation with a receiver or argument (or both) of type `Object`.

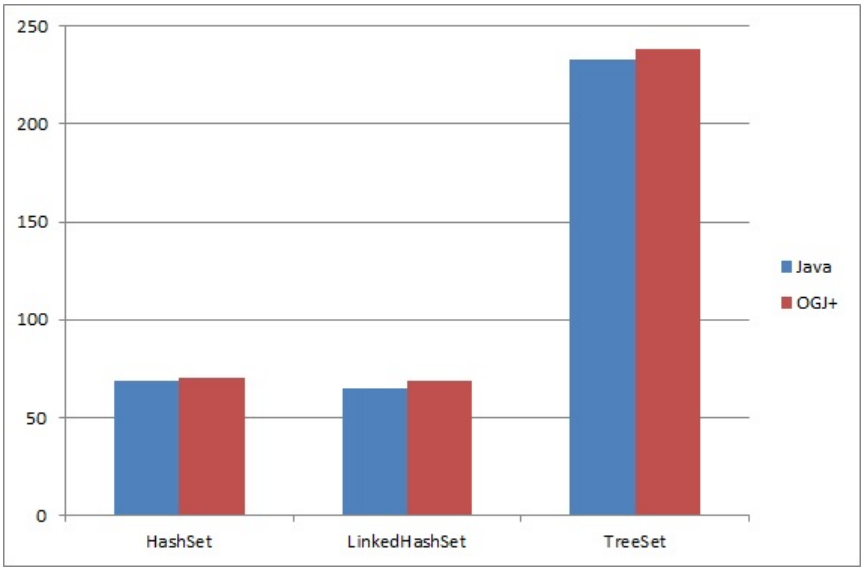

Figure 6.3: `CollectionLoops` for Sets – Nanoseconds per Operation

Fig. 6.3 shows the results for the `Set` classes. The plots show no significant variance (1-5 nanoseconds) of performance degradation. The `contains()` method of a `Set` implementation delegates lookups to `containsKey()` of the backing `Map` implementation. With respect to the hash-based `Map` implementations, `containsKey()` uses the hash of the key to go directly to the position where this key should reside and then uses `equals()` to check if it is there. That is, `equals()` is not involved in a lookup iteration process, of which each step requires a call to `equals()`; and thus, the performance of the refactored sets is just as the original sets. The `containsKey()` of `TreeMap` does not use `equals()` to determine equality; it uses `Comparable.compareTo()`. In contrast to a binary method such as `equals()`, the receiver type and the argument type of `compareTo()` do not need to coincide. Moreover, the argument type of `compareTo()` is `K` (the variable type for the keys) in both the refactored and the original `TreeMap` classes. That is, in both versions there is a single implicit cast required for each invocation during the lookup; and thus, the performance of the refactored `TreeSet` is similar to that of the original `TreeSet`.

6.3 MapLoops

`MapLoops` can test only `Map` classes. Similar to `CollectionLoops`, `MapLoops` is originally implemented to exercise multi-threaded maps, but we modified it to exercise single-threaded maps and to exercise the same number of operations and trials as `CollectionLoops`. Similar to what we did with `Set` classes, `MapLoops` were repeatedly run, then the mean of the best five repetitions were calculated to represent the performance time per operation. The plots in Fig. 6.4 show that the performance of the refactored maps is not significantly different from that of the original maps.

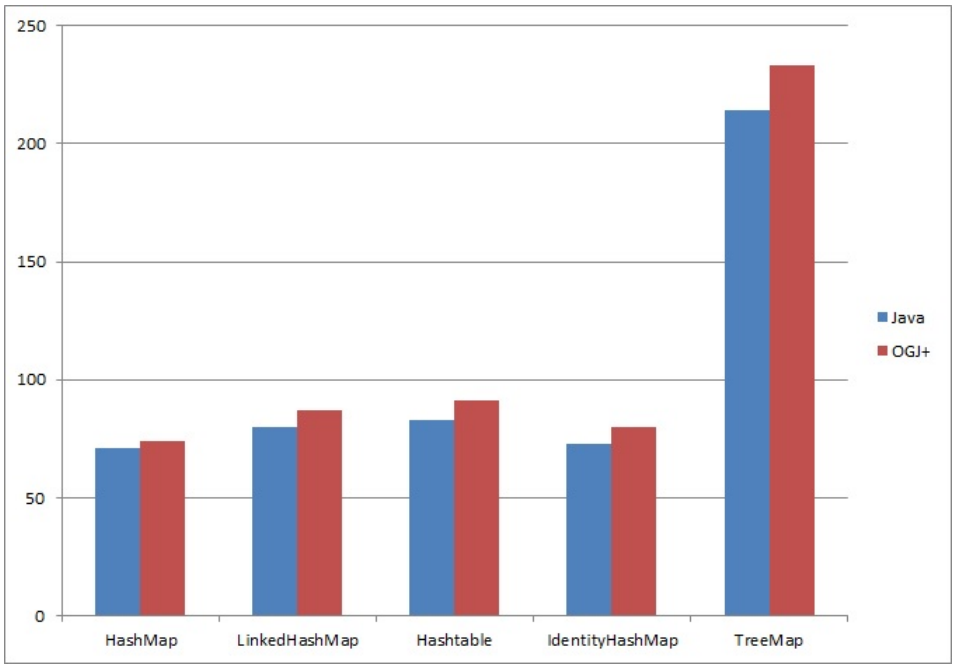

Figure 6.4: `MapLoops` – Nanoseconds per Operation

6.4 MapMicroBenchmark

`MapMicroBenchmark` is another maps specific benchmark, but it cannot run maps that do not permit insertion of non-comparable objects (objects that do not implement the `Comparable` interface). Therefore, we couldn't test `TreeMap` with `MapMicroBenchmark`. According to the very brief documentation provided, this is "a micro-benchmark with key types and operation mixes roughly corresponding to some real programs". "The main results are a table of approximate nanoseconds per element-operation (averaged across get, put etc) for each type, across a range of map sizes"; see Fig. 6.6. We did not make any modification to the default values of this class. The plots in Fig. 6.5 represent the averages calculated for the largest map size (see the boxed number in the lower-right corner of Fig. 6.6). As shown, the performance of the refactored maps is almost the same as that of the original maps.

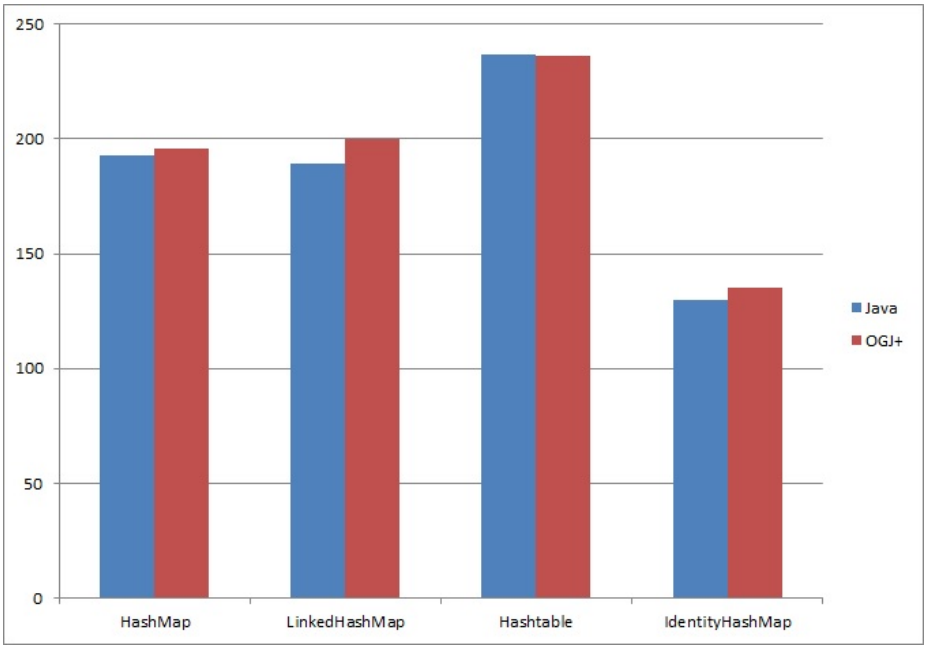

Figure 6.5: `MapMicroBenchmark` – Nanoseconds per Operation over a map size of 589,824

Class java.util.HashMap randomized searches
No word file. Using String.valueOf(i)
warm up.....
warm up.....
running.....

Type/Size:	9	36	144	576	2304	9216	36864	147456	589824
Object	24	24	24	26	29	40	52	140	175
String	22	24	24	27	28	44	97	167	191
Integer	32	31	31	30	32	40	52	129	162
Long	23	21	21	21	23	31	42	120	155
Float	29	25	29	30	31	36	56	133	187
Double	24	27	26	25	33	39	50	148	186
BigInteger	29	30	29	30	35	47	110	181	213
BigDecimal	25	31	31	31	35	49	79	173	199
RandomInt	37	36	37	37	41	53	71	177	219
Mixed	35	38	43	48	58	72	152	212	249
average	28	28	29	30	34	45	76	158	193

Figure 6.6: `MapMicroBenchmark`'s results for JDK 1.6 `HashMap`

6.5 Discussion

Merging ownership information into generic declarations does not impose significant runtime overheads, if there are any. Since all information about generics is erased by the compiler after all checks are done, Java generics might seem to be implemented with high overhead, requiring implicit upcasts to `Object` in the process of inserting elements into a collection, and implicit downcasts to the type of the elements in the process of taking the elements out of the given collection. This is not precise, however. The fact is that the implementations of some `Collection` classes involve arrays of type `Object` rather than `E`. In the refactored collections, all of the backing arrays became of type `E`, which is the type variable of their respective `Collection` classes. We did not voluntarily choose to make the backing arrays hold elements of type `E`. As explained in subsection 5.1.1, we had no choice but to make the methods, which take `Object` as an argument, take `E` as an argument due to the owner nesting rule imposed by OGJ$_+$. Similarly, the way to make the owner of the backing array elements the same as the owner of the collection elements is by making the type of the

backing array the same as the type of the collection elements, given that Java's "no generic array creation" property is no longer a concern to OGJ$_+$.

There is no doubt that Java generics impose runtime overheads which we notice clearly as we test the refactored `HashMap` class with `Iterator-Loops`. Walking an array that preserves its type information at runtime is more efficient than walking a generic aggregate such as `OwnedArray`. The use of array types, which do not hold elements of concrete parameterized types, to back collection classes, which accept elements of concrete parameterized types, has in fact eased the cost of Java generics.

Merging ownership information into generic declarations necessitated the prevention of raw types and unbounded wildcard parameterized types; both kinds of types are permitted as types of Java arrays. Preserving the owners of both the backing aggregate and its component type is an uncompromised priority in OGJ$_+$. By using `OwnedArray` as a workaround for the nonexistent generic array type, various settings have changed in both the implementation of the collection classes and their generated byte code. In addition to the added runtime `checkcast` instructions and implicit casts, operations such as `OwnedArray.get()` and `OwnedArray.set()` had to replace the array index operator. Such operations, with others, should impose runtime overheads. Nevertheless, the experimental results we presented suggest that the cost of combing ownership types with parametric polymorphism is relatively low, and can even be avoided by takeing alternative implementation decisions.

OGJ$_+$ has the ability to enforce certain implementation decisions in favour of efficiency. For example, with the original collections one can query if an orange is contained in a basket of apples, and the answer is false because such a query is allowed; but with the refactored collections, such a query is not allowed. Our experiments turned out that allowing this kind of queries with Java generics is expensive. For OGJ$_+$ to confirm that owner nesting is preserved, the possibility of allowing this kind of queries is very limited; and if the owner of the component type is sub-

ject to context covariance, as is the case with the refactored collections, the possibility does not exist. Indeed, making an owner subject to context covariance is our decision, but maintaining readonly wildcard owner parameterised references from inner classes to their outer classes necessitated this decision. Otherwise, we should have made the component type owned by `World`, which is not a reasonable decision.

The results of `MapMicroBenchmark` confirm that real programs would not be affected by the overhead `IteratorLoops` produced for `HashMap`. Based on the entire experimental results, we believe that the OGJ$_+$ prototype has confirmed expectations, given that Java generics current implementation does not touch the JVM or the class file structure. Emplacing ownership information into generic declarations, with the required workarounds, might keep the question open: what does matter more, security or efficiency? We believe, however, that the cost of security is not high, based on these results.

Chapter 7

Conclusions

Supporting efficient, true and handy encapsulation of software components and their associated data types is a lively concern in object-oriented research. In this thesis we have presented OGJ$_+$, an actual usable language implementation with deep ownership types extended with support for readonly references. OGJ$_+$ extends Java with a form of statically enforceable instance-level hierarchical encapsulation, rather than the class-level private field encapsulation. We claim to have taken advantage of Java Generics to the extreme. We have utilised the safe form of covariance provided by the wildcard feature of Java generics to provide flexible referencing without compromising the encapsulation or the nested structure of the heap. Some features, such as inner classes, equals, statics and exception handling, would have been impaired under the strict model of deep ownership, if wildcards were not utilised to favour ownership.

OGJ$_+$ proved to have the potentials of supporting safe programming with aliasing. For the practical evaluation, we refactored an essential portion of the Collections Framework, showing that programming with OGJ$_+$ will not hamper the construction of realistic software. While maintaining the expressiveness of aliasing properties of the fields, OGJ$_+$ was able to precisely detect all breaches of encapsulation. We showed that the refactoring was made without violating the traditional Java programming

style; programmers are still able to take advantage of the full potentials of inner classes. Therefore, we believe we managed to find a viable solution for the iterator pattern. In comparison with the ad hoc approach proposed by Boyapati et al. [9], OGJ$_+$ does not permit an inner class and its outer class to share a common representation; ownership information is strictly class-specific. The internal representation of an outer class can only be referred to, form an inner class, through wildcard-owned references, which are readonly.

During the refactoring, OGJ$_+$ showed its ability to enforce constraints on data structures; during the experiments, these constraints proved to mitigate or reduce potential negative effects of implicit casting. Nevertheless, the nature of generic ownership as a language level mechanism associated with Java generics makes all of our solutions coupled with the effects and side effects of Java generics.

The undesired effects of aliasing (see example in Fig. 2.2) are known as a problem [31] that programmers should be aware of. Researchers suggest that thinking about ownership structures can lead to better code structures [58]. We claim that programming with OGJ$_+$ allows only positive thoughts of aliasing, and requires knowledge about the meaning of ownership types.

7.1 Limitations

The most notable limitation of OGJ$_+$ is that the provided stronger encapsulation guarantees comes at the cost of programming overhead. We discussed in the thesis a number of limitations that might impact the way software is written in OGJ$_+$. Section 3.5 describes how casting is treated in OGJ$_+$ and explains the idea of using wildcards to overcome the limitation of not keeping track of the owners at run-time. Since casting cannot be dealt with in isolation from the equals idiom, section 4.5 takes the discussion further afield than in section 3.5. Section 4.2 describes a limita-

tion with the event-listeners design pattern; the pattern can still be implemented in OGJ$_+$, however. Subsection 5.4.2 highlights another limitation with the template-method design pattern. Note that the aim of this thesis project is not to evaluate OGJ$_+$ with design patterns. Finally, section 4.4 involves a discussion on the restrictive nature of OGJ$_+$'s deep cloning.

7.2 Future Directions

We will continue to improve OGJ$_+$ and evaluate it in large programs. We think of evaluating the language with the concurrent implementations of the Collections Framework. We are also interested in evaluating the language with design patterns and see if more extensions would be required. Initially, we think that a feature for ownership transfer [48] would be a great addition to the language. Another interesting future work might be allowing an object to have multiple owners [14].

We also think that relying on genericity should not be everything for OGJ$_+$. Finding a more reasonable solution for arrays would be a great restoration. Supporting immutability could also be an interesting future work, in this regard.

Bibliography

[1] Collections framework overview. In *Java SE Documentation*, Oracle Technology Network. Available Online: http://download.oracle.com/ javase/6/docs/technotes/guides/collections/overview.html. Viewed 16 Apr 2011.

[2] Java Collections API Design FAQ. In *Java SE Documentation*, Oracle Technology Network. Available Online: http://download.oracle.com/ javase/6/docs/technotes/guides/collections/designfaq.html. Viewed 26 May 2011.

[3] ALDRICH, J., AND CHAMBERS, C. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP '04 — Object-Oriented Programming European Conference* (Oslo, Norway, 2004), M. Odersky, Ed., vol. 3086 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–25.

[4] ALMEIDA, P. S. Balloon types: Controlling sharing of state in data types. In *ECOOP* (1997), pp. 32–59.

[5] BLOCH, J. Collections. In *The Java Tutorials*, Oracle Technology Network. Available Online: http://download.oracle.com/ javase/tutorial/collections/. Viewed 16 Apr 2011.

[6] BOOCH, G., MAKSIMCHUK, R., ENGLE, M., YOUNG, B., CONALLEN, J., AND HOUSTON, K. *Object-oriented analysis and design with applications, third edition*, third ed. Addison-Wesley Professional, 2007.

[7] BOYAPATI, C. *SafeJava : a unified type system for safe programming*. PhD thesis, Massachusetts Institute of Technology, 2004.

[8] BOYAPATI, C., LEE, R., AND RINARD, M. Safe runtime downcasts with ownership types. In *International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming* (July 2003), D. Clarke, Ed., UU-CS-2003-030, Utrecht University.

[9] BOYAPATI, C., LISKOV, B., AND SHRIRA, L. Ownership types for object encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2003), POPL '03, ACM, pp. 213–223.

[10] BOYLAND, J. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience 31*, 6 (May 2001), 533–553.

[11] BOYLAND, J., NOBLE, J., AND RETERT, W. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming* (London, UK, 2001), Springer-Verlag, pp. 2–27.

[12] CAMERON, N., AND DROSSOPOULOU, S. Existential quantification for variant ownership. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009* (Berlin, Heidelberg, 2009), ESOP '09, Springer-Verlag, pp. 128–142.

[13] CAMERON, N., AND NOBLE, J. Ogj gone wild. In *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming* (New York, NY, USA, 2009), IWACO '09, ACM, pp. 7:1–7:10.

[14] CAMERON, N. R., DROSSOPOULOU, S., NOBLE, J., AND SMITH, M. J. Multiple ownership. In *Proceedings of the 22nd annual ACM SIGPLAN*

conference on Object-oriented programming systems and applications (New York, NY, USA, 2007), OOPSLA '07, ACM, pp. 441–460.

[15] CAMERON, N. R., AND NOBLE, J. Encoding ownership types in java. In *TOOLS (48)* (2010), J. Vitek, Ed., vol. 6141 of *Lecture Notes in Computer Science*, Springer, pp. 271–290.

[16] CLARKE, D. *Object Ownership & Containment*. PhD thesis, University of New South Wales, 2001.

[17] CLARKE, D., AND DROSSOPOULOU, S. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2002), OOPSLA '02, ACM, pp. 292–310.

[18] CLARKE, D., SHELSWELL, R., POTTER, J., AND NOBLE, J. Object ownership to order. Tech. rep., Microsoft Research Institute, 1998.

[19] CLARKE, D., AND WRIGSTAD, T. External uniqueness is unique enough. In *In European Conference for Object-Oriented Programming (ECOOP* (2003), Springer-Verlag, pp. 176–200.

[20] CLARKE, D. G., NOBLE, J., AND POTTER, J. Overcoming representation exposure. In *ECOOP Workshops* (1999), A. M. D. Moreira and S. Demeyer, Eds., vol. 1743 of *Lecture Notes in Computer Science*, Springer, pp. 149–151.

[21] CLARKE, D. G., POTTER, J. M., AND NOBLE, J. Ownership Types for Flexible Alias Protection. *ACM SIGPLAN Notices 33*, 10 (Oct. 1998), 48–64.

[22] DETLEFS, D. L., LEINO, K. R. M., AND NELSON, G. Wrestling with rep exposure. SRC Research Report 156, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, July 1998.

[23] DIETL, W., DROSSOPOULOU, S., AND MÜLLER, P. Generic universe types. In *ECOOP* (2007), E. Ernst, Ed., vol. 4609 of *Lecture Notes in Computer Science*, Springer, pp. 28–53.

[24] DIETL, W., AND MÜLLER, P. Exceptions in ownership type systems. In *ECOOP Workshop FTfJP'2004 Formal Techniques for Java-like Programs* (June 2004), E. Poll, Ed., pp. 49–54.

[25] DIETL, W., AND MÜLLER, P. Universes: Lightweight ownership for JML. *Journal of Object Technology 4*, 8 (2005), 5–32.

[26] ELIËNS, A. *Principles of Object-Oriented Software Development*, second ed. Addison-Wesley, 2000.

[27] FAHNDRICH, M., AND DELINE, R. Adoption and focus: practical linear types for imperative programming. *SIGPLAN Not. 37*, 5 (2002), 13–24.

[28] GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[29] GROGONO, P., AND SAKKINEN, M. Copying and comparing: Problems and solutions. In *Proceedings of the 14th European Conference on Object-Oriented Programming* (2000), ECOOP '00, Springer-Verlag, pp. 226–250.

[30] HOGG, J. Islands: aliasing protection in object-oriented languages. *ACM SIGPLAN Notices 26*, 11 (Nov. 1991), 271–285.

[31] HOGG, J., LEA, D., WILLS, A., DECHAMPEAUX, D., AND HOLT, R. The Geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess. 3*, 2 (1992), 11–16.

[32] HUANG, W., AND MILANOVA, A. Towards effective inference and checking of ownership types. In *Proceedings of the International Workshop on Aliasing, Confinement and Ownership at ECOOP (IWACO 2011)* (July 2011).

[33] LEA, D. Jsr-166 loops microbenchmarks. http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/test/loops/.

[34] LEAVENS, G. T., BAKER, A. L., AND RUBY, C. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, H. Kilov, B. Rumpe, and I. Simmonds, Eds. Kluwer Academic Publishers, Boston, 1999, pp. 175–188.

[35] LEAVENS, G. T., BAKER, A. L., AND RUBY, C. Preliminary design of JML: A behavioral interface specification language for Java. Tech. Rep. 98-06y, Iowa State University, Department of Computer Science, 2003. Revised June 2004.

[36] LEAVENS, G. T., POLL, E., CLIFTON, C., CHEON, Y., RUBY, C., COK, D. R., MÜLLER, P., KINIRY, J., CHALIN, P., ZIMMERMAN, D. M., AND DIETL, W. JML reference manual. Available from `http://www.jmlspecs.org`, May 2008.

[37] LEINO, K. R. M., AND STATA, R. Virginity: a contribution to the specification of object-oriented software. *Inf. Process. Lett. 70*, 2 (Apr. 1999), 99–105.

[38] LI, P., CAMERON, N., AND NOBLE, J. Cloning in ownership. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion* (New York, NY, USA, 2011), SPLASH '11, ACM, pp. 63–66.

[39] LISKOV, B., AND GUTTAG, J. *Abstraction and specification in program development*. MIT Press, Cambridge, MA, USA, 1986.

[40] LISKOV, B., SNYDER, A., ATKINSON, R., AND SCHAFFERT, C. Abstraction mechanisms in CLU. *Commun. ACM 20*, 8 (August 1977), 564–576.

[41] LISKOV, B., AND ZILLES, S. Programming with abstract data types. In *ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN Notices* (Apr. 1974), vol. 9, pp. 50–59.

[42] MEYER, B. *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[43] MINSKY, N. H. Towards alias-free pointers. In *ECCOP '96: Proceedings of the 10th European Conference on Object-Oriented Programming* (London, UK, 1996), Springer-Verlag, pp. 189–209.

[44] MÜLLER, P. *Modular Specification and Verification of Object-Oriented Programs*, vol. 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[45] MÜLLER, P., AND POETZSCH-HEFFTER, A. Universes: A type system for controlling representation exposure. In *Programming Languages and Fundamentals of Programming* (1999), A. Poetzsch-Heffter and J. Meyer, Eds., Fernuniversität Hagen.

[46] MÜLLER, P., AND POETZSCH-HEFFTER, A. A type system for controlling representation exposure in Java. In *Formal Techniques for Java Programs* (2000), S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, Eds., Technical Report 269, Fernuniversität Hagen.

[47] MÜLLER, P., AND POETZSCH-HEFFTER, A. Universes: A type system for alias and dependency control. Tech. Rep. 279, Fernuniversität Hagen, 2001.

[48] MÜLLER, P., AND RUDICH, A. Ownership transfer in universe types. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications* (New York, NY, USA, 2007), ACM, pp. 461–478.

[49] NAFTALIN, M., AND WADLER, P. *Java Generics and Collections*. O'Reilly Media, Inc., 2006.

[50] NÄGELI, S. Ownership in design patterns. Master's thesis, Software Component Technology Group, Department of Computer Science, ETH Zurich, 2006.

[51] NOBLE, J., CLARKE, D., AND POTTER, J. Object ownership for dynamic alias protection. In *Proceedings TOOLS '99* (Nov. 1999).

[52] NOBLE, J., VITEK, J., AND POTTER, J. Flexible Alias Protection. In *ECOOP '98—Object-Oriented Programming* (1998), E. Jul, Ed., vol. 1445 of *Lecture Notes in Computer Science*, Springer, pp. 158–185.

[53] PAGH, A., PAGH, R., AND RUZIC, M. Linear probing with constant independence. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing* (New York, NY, USA, 2007), STOC '07, ACM, pp. 318–327.

[54] PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Commun. ACM 15*, 12 (Dec. 1972), 1053–1058.

[55] PARNAS, D. L. A technique for software module specification with examples. *Commun. ACM 15*, 5 (May 1972), 330–336.

[56] PIERCE, B. C. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[57] POTANIN, A. *Generic Ownership: A Practical Approach to Ownership and Confinement in OO Programming Languages*. PhD thesis, Victoria University of Wellington, 2007.

[58] POTANIN, A., DAMITIO, M., AND NOBLE, J. Are your incoming aliases really necessary? counting the cost of object ownership. In *Proceedings of the 2013 International Conference on Software Engineering* (Piscataway, NJ, USA, 2013), ICSE '13, IEEE Press, pp. 742–751.

[59] POTANIN, A., NOBLE, J., CLARKE, D., AND BIDDLE, R. Defaulting Generic Java to Ownership. In *In Proceedings of the Workshop on Formal Techniques for Java-like Programs in European Conference on Object-Oriented Programming (FTfJP* (2004), Springer-Verlag.

[60] POTANIN, A., NOBLE, J., CLARKE, D., AND BIDDLE, R. Generic ownership. In *In 7th Workshop on Formal Techniques for Java-like Programs - FTfJP2005* (2004).

[61] POTANIN, A., NOBLE, J., CLARKE, D., AND BIDDLE, R. Generic Ownership for Generic Java. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications* (New York, NY, USA, 2006), OOPSLA '06, ACM, pp. 311–324.

[62] POTTER, J., NOBLE, J., AND CLARKE, D. The ins and outs of objects. In *Proceedings of the Australian Software Engineering Conference* (Washington, DC, USA, 1998), ASWEC '98, IEEE Computer Society, pp. 80–.

[63] SUMMERS, A. J., DROSSOPOULOU, S., AND MÜLLER, P. Universe-type-based verification techniques for mutable static fields and methods. *Journal of Object Technology 8*, 4 (2009), 85–125.

[64] VITEK, J., AND BOKOWSKI, B. Confined types in Java. *Software — Practice and Experience 31*, 6 (2001), 507–532.

[65] WADLER, P. Linear types can change the world! In *Working Conference on Programming Concepts and Methods* (Sea of Galilee, Israel, April 1990), IFIP TC 2.

[66] WEGNER, P. Dimensions of object-based language design. In *Conference proceedings on Object-oriented programming systems, languages and applications* (New York, NY, USA, 1987), OOPSLA '87, ACM, pp. 168–182.

[67] WRIGSTAD, T. *Ownership-Based Alias Management*. PhD thesis, Royal Institute of Technology, Sweden, May 2006.

[68] WRIGSTAD, T., AND CLARKE, D. Existential owners for ownership types. *Journal of Object Technology 6*, 4 (2007).