

# Changing Hashcodes: Objects, Intialisation, and Collections

Stephen Nelson   David J. Pearce   James Noble

Victoria University of Wellington  
{stephen,djp,kjx}@ecs.vuw.ac.nz

## Abstract

Java requires objects' hashcodes to be consistent with object equality, and to change only when an object's equality changes. Equality dependent Java collections impose a stricter contextual contract on an objects' equality, comparability, and hashcode methods: that they cannot change while those objects are in collections. By dynamically profiling 30 Java applications, we have determined that programmers use quite sophisticated techniques to ensure their objects meet these contracts. Some objects are immutable; some objects are initially mutable, but become immutable when inserted into equality- dependent collections; some objects continue to change when inserted into equality dependent collections — but those changes do not affect equality or hashcodes. On the other hand, some objects inserted into collections that do not depend on object equality, hashcodes, or comparisons will contine to change all their state, including equality and hashcode, while they are in those collections.

Programmers and educators can use these results to ensure they (or their students) are aware of these subtle conditional contracts underlying Java's collection design. These results can also guide language designers adding support for collections, relations, or equality: in particular; while immutability suffices for many cases, it does not cover all of them.

## 1. Introduction

Designing good software is hard. Designing good programming languages is much harder. Modern programming languages have evolved to include numerous high-level constructs, and to provide vast libraries of reusable code. Inheritance, polymorphism, exception handling, collections, first-class regular expressions, and the “elvis” operator are just a few examples. Many of these constructs have subtle and important effects on the way good software is written.

In this paper, we are interested in studying the effect of a specific language feature on software design: object equality and hashcodes in Java. Java's `Object` contract imposes a number of requirements on `equals` and `hashCode`. Then, the `Set` and `Map` interfaces impose stricter contracts of their own: the equality and hashcodes of the objects they contain — for `Maps`, only as keys, not as values — do not change while they are in the collections.

The question we tackle in this paper is: *how do programmers satisfy these contracts?*. To answer this question, we have profiled a large number of Java programs, measuring the way objects change, and the differences in objects' behaviour when they are inserted into collections. The contributions of this paper are:

1. we hypothesise four strategies used by programs to satisfy Java's equality contracts: full immutability, post-constructor initialisation, identity as equality, and reindexing.
2. we designed and implemented *#profiler*, a dynamic analysis that detects conformance to these contracts
3. based that analysis, we discovered how programmers fulfil those contracts: primarily by using identity as equality, and secondarily by avoiding significant changes to objects in equality-dependent collections.
4. we also found that many objects have an initialisation phase that continues after their constructor has completed — after which objects do not change their equality, although they change in other ways.

### 1.1 Organisation

The rest of this paper is organised as follows: Section 2 discusses various contracts imposed by Java, particularly those pertaining to collections; Section 3 outlines our approach to categorising objects according to the way they address these contracts; Section 4 then discusses the implementation of our profiling tool, *#Profiler*, in more detail; Section 5 presents our experimental results looking at the behaviour of objects across 30 open source Java applications; Section 6 covers related work and, finally, we summarise our findings in the conclusion.

## 2. Motivation

Deciding whether objects are equal is an important issue in any modern programming language. In Java, this boils down to a question of whether or not one should override the `equals()` method. If not, a default implementation which checks whether two objects have the same *identity* is provided. By identity, we mean the unique identifier every object is assigned (which normally corresponds to its address in memory). On the other hand, one can override the `equals()` method to provide a more flexible form of equality. The following illustrates a simple example:

```
class Add implements Expr {
    private Expr lhs, rhs;
    ...
    boolean equals(Object o) {
        if (!(o instanceof Add)) return false;
        Add b = (Add) o;
        return lhs.equals(b.lhs) &&
            rhs.equals(b.rhs);
    }
}
```

We can imagine this class being used as part of the Abstract Syntax Tree (AST) in a compiler. The idea behind overriding the `equals()` method is to ensure that two add expressions are considered to be equal iff they are *structurally equivalent*. That is, that is, although they be made up from different objects, they encode the same structure.

In the above example, the `equals()` method for two `Add` objects depends not just on the objects themselves, but also on objects reachable from them in the heap. We refer to those objects and fields upon which an object's `equals()` method depends as its *equality*. In Java, an object's equality does not necessarily depend upon all the fields and objects accessible from it (although this is true of the above example). Thus, the equality of an object is a subset of all the state accessible from it.

A common issue when overriding the `equals()` method is that one must also remember to override the `hashCode()` method. Failure to do this is a common pitfall for novice programmers although, fortunately, there are tools which can remind us to do this (see [26, 27]). The reason one must override `hashCode()` when overriding `equals()` is down to a contract in `java.lang.Object`:

*“If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.”*

Another contract from `java.lang.Object` further restricts how the `hashCode()` method may behave:

*“Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified.”*

This contract means that the `hashCode()` of an object must consistently return the same value, provided no part of its equality has changed. Whilst not explicitly mentioned above, this implies the state upon which `hashCode()` depends must be a subset of an object's equality.

Another contract with a similar effect to the `equals()` method is the `compareTo()` method defined in the class `java.lang.Comparable<T>`. This method can be used to check not just whether two objects are equal, but also whether one is less than the other. An interesting question is how this method relates to the `equals()` method, and the documentation for `Comparable<T>` says the following:

*“It is strongly recommended (though not required) that natural orderings be consistent with `equals`. This is so because sorted sets (and sorted maps) without explicit comparators behave “strangely” when they are used with elements (or keys) whose natural ordering is inconsistent with `equals`. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the `equals` method.”*

Thus, we can define *comparability* as being analogous to equality, but for the `compareTo()` method. That is, the *comparability* of an object is that state upon which the `compareTo()` method depends. If one takes the above recommendation then the comparability and equality of an object must, in fact, be the same.

At this stage, we begin to see how the contracts given in the Java API can have important consequences on the way programs are written. Whilst the above examples are fairly well understood, we will now delve deeper into those contracts required by the Collections API.

### 2.1 Collection Interfaces

The Java Collections API provides four main interfaces: `List`, `Set`, `Map` and `Queue`. While `List` is probably the most frequently used in practice, the others still play an important role. For example, the hash collection implementations (i.e. `HashMap` and `HashSet`) are often used as they provide a good balance between functionality and performance. These four interfaces impose different requirements on the objects they contain, which has a non-trivial effect on the way programs are designed and implemented.

The `Set` and `Map` interfaces impose particularly strict requirements on the objects they contain. In particular, the javadoc for `java.util.Set` states the following (and similarly for `Map`):

*“Note: great care must be exercised if mutable objects are used as set elements. The behaviour of a set is not specified if the value of an object is changed in a manner that affects `equals` comparisons while the object is an element in the set.”*

The above contract means that, whilst an object is contained in a `Set` (resp. `Map`), its equality must be immutable. This is because, while it is possible to safely update its equality whilst it is contained in a `Set` (resp. `Map`), it is hard to do this sensibly in practice. For example, one could swap some substructure of an object with something structurally equivalent — but, there seems little value in doing this.

Considering other interfaces and implementations in the Java Collections API, we find many similar requirements. For example, objects placed into a `PriorityQueue` must implement `Comparable<T>`. Thus, updates which affect an object's comparability whilst it is in a `PriorityQueue` result in undefined behaviour<sup>1</sup>. This holds similarly for implementations of `SortedSet` and `SortedMap`.

Now, let us consider those Java collection classes whose implementation depends on an object's equality or comparability. Examples include `HashMap`, `HashSet`, `TreeMap`, `TreeSet` and `PriorityQueue` to name a few. We refer to collections in this category as being *equality-dependent*. Contrasting with this, are those collections which are *equality-independent*. Examples here include `ArrayList`, `Vector`, `LinkedList`, and `IdentityHashMap`. These collections will continue to function correctly *even if the equality or comparability of a contained object changes*. However, such collections are not entirely independent of object equality as some of their methods, most notably `contains()`, use the `equals()` method, rather than a direct identity comparison.

Give the strict contracts imposed by the equality-dependent collections, the question is: *how do programmers address them?* Clearly, the compiler is of no help here, and so programmers must resort to patterns and conventions to ensure these requirements are not broken. We consider that there are several approaches that could be employed:

1. Identity as Equality: they do not implement `equals` or `hashCode`, but use the default identity-based implementations.
2. Full immutability: they ensure that an object is immutable; that is, its state, and that of those objects reachable from it *simply does not change*. Thus, the contracts imposed by equality-dependent collections are trivially satisfied.
3. Immutable Equality: they ensure the equality of objects placed into equality-dependent collections *simply never changes*. That is, the objects are immutable with respect to those fields and objects affecting `equals()` and `hashCode()`.
4. Post-Constructor Initialisation: they allow objects placed into equality-dependent objects to go through a *post-construction equality initialisation phase*. That is, the

<sup>1</sup>It's interesting to note, however, that the Java Collection API does not highlight this point explicitly. Nevertheless, it must be true if we consider how a priority queue will be implemented.

equality of the object changes for a brief period after its initial construction, but then does not change for the remainder of the program.

5. Reindexing: they carefully coordinate updates to an object's equality to ensure that such updates do not occur on an object whilst it is contained within an equality-dependent collection.

The interesting thing here is that, whilst all five approaches are quite plausible, we have found that only (1) to (4) are used in practice. Whilst this was somewhat surprising to us, there are a few reasons it might be.

Considering approach (5), we believe programmers avoid this because coordinating such updates typically requires non-local reasoning. That is, they must either know: that no equality-dependent collection contains an object before updating it; or, conversely, that an object will not be updated (outside of their control) when putting it into an equality-dependent collection.

There appear to be two patterns of using which conform to approach (4). Firstly, many mutable classes are used in an immutable fashion *when they are to be contained in equality-dependent collections*. Thus, when initialising the objects before placing them into those collections, programmers often use the mutable interface since it is convenient. Secondly, many classes are mutable *even when they don't need to be*. That is, by providing additional parameters to their constructors, those mutator methods could have been avoided altogether, resulting in fully immutable objects — but, for some reason, the authors chose not to do this.

In this paper, we are interested in exploring which of these strategies is used more frequently in practice. We have implemented a profiling system designed to examine the way in which real programs operate and, hence, give insight into this issue. However, before discussing the profiling system, we will now consider each of the three strategies above in more detail.

## 2.2 Identity as Equality

By far the simplest approach, if the programmer does not need equality between different (non-identical) objects then they can use the default implementations.

## 2.3 Immutable Objects

In this case, the programmer prevents the equality of an object changing in the simplest way possible: by statically preventing updates to its fields or reachable objects. To illustrate, we present an example taken from sun's `javac` compiler (`openjdk6`). The example has been simplified for the purposes of our presentation, to remove unnecessary details; however, with respect to the issues we are considering, it remains faithful to the original code.

An important class in `javac` is `Type`, which represents Java types. There are a variety of different types which must be represented (i.e. primitives, arrays, references, methods),

```

public class MethType extends Type {
    List<Type> argtypes;
    Type restype;

    public MethType(List<Type> as, Type rt) {
        this.argtypes = as;
        this.restype = rt;
    }

    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (!(obj instanceof MethType)) {
            return false;
        }
        MethType m = (MethType)obj;
        return argtypes.equals(m.argtypes) &&
            restype.equals(m.restype);
    }

    public int hashCode() {
        return argtypes.hashCode() +
            this.restype.hashCode();
    }
}

```

**Figure 1.** Illustrating a class for representing method types, taken from sun’s javac compiler.

and each is implemented in its own subclass of `Type`. Figure 1 provides one such example, for representing method types. The key features of this class, from our perspective, are that: firstly, it contains a `List` of `Type` objects which represent the parameter types; secondly, that the `equals()` and `hashCode()` methods depend upon this list; finally, no (direct) mechanism is provided for updating this list and/or the return type.

The design of `javac` requires that two types objects are equal if they are structurally equivalent. This is implemented, as usual, by overriding the `equals()` and `hashCode()` methods. The reason for this requirement in the design of `javac` is, amongst other things, that the algorithms for type-checking Java source code desire to check whether certain types match. For example, that the right-hand side of the `<<<` operator has type `int`. Thus, the system needs to create types from one part of a program’s source, and compare them against those created from another part.

An important feature of `MethType`’s design is that no mechanism is provided to update a given `MethType` object. The advantage of this approach, is that the contract implied by equality-dependent collections is trivially satisfied. And, indeed, we note there are several places in `javac` where `Set<Type>` and where `Map<Type, ...>` collections are used. More common, however, is that `Type`

objects are placed in `List` objects and looked up using `List.contains()`.

Finally, the observant reader will have noticed a potential bug in `MethType` — it’s objects are not, in fact, immutable! This is because the `argtypes` list may have external references, as illustrated below:

```

List<Type> ls = new ArrayList<Type>();
MethType mt = new MethType(ls, ...);
ls.add(new IntType()); // structural change

```

Presumably, the designers of `MethType` have implicitly assumed the `argtypes` parameter is uniquely owned (i.e. that its reference is not stored elsewhere).

## 2.4 Immutable Equality

We now consider in more detail the third strategy outlined above. Recall that, in this case, the programmer complies with the requirements of equality-dependent collections by statically preventing updates to fields that affect the outcome of `equals()` and `hashCode()`.

Figure 2 provides an (artificial) example illustrating this approach. This time, we imagine this class being used as part of the Abstract Syntax Tree (AST) in an Integrated Development Environment (IDE). Thus, the highlighted field is used to signal that the expression has been highlighted as being syntactically incorrect. The key to this example, is that the programmer has statically prevented the equality of any instances from changing after construction (since no setter methods are provided for `lhs` or `rhs`).

Finally, we had expected to find real-world examples which used this approach. However, in practice, it turned out to be very difficult to find any such examples.

## 2.5 Pseudo-Immutable Equality

We now consider the forth strategy outlined previously for dealing with the requirements imposed by equality-dependent collections. In this case, the programmer does not statically prevent an object’s equality from being updated. Thus, update methods are present which can affect the outcome of `hashCode()` and `equals()`. However, these methods are not used arbitrarily; rather, they are called to update an object very soon after construction, and not used again. Thus, once this *post-construction equality initialisation* is complete, the object is treated as if it were immutable (with respect to equality). To be more precise in our discussion of this issue, we will use the term *post-construction initialisation phase* to refer to situations where an object’s initialisation is not completed by its constructor. Thus, post-construction equality initialisation is the special case where its equality is not completely initialised during construction.

The key difference between this and the previous approach is that the programmer cannot (for some reason) fully initialise an object via its constructor. The real question is

```

public class Add implements Expr {
    private Expr lhs, rhs;
    private boolean highlighted

    public Expr lhs() { return lhs; }
    public Expr rhs() { return rhs; }

    public void setHighlight(boolean hl) {
        highlighted = hl;
    }
    public boolean getHighlighted() {
        return highlighted;
    }

    public boolean equals(Object o) {
        if(!(o instanceof Add)) return false;
        Add a = (Add) o;
        return a.lhs.equals(lhs) &&
            a.rhs.equals(rhs);
    }
    public int hashCode() {
        return lhs.hashCode() + rhs.hashCode();
    }
}

```

**Figure 2.** Illustrating an example class which equality-immutable. That is, whilst the `lineNumber` field can be change, the others cannot since they determine the object's equality.

how such a situation can arise and we now consider several different scenarios that arise in practice.

The most common reason for post-construction equality initialisation is that the class being instantiated is designed for use in a variety of situations, including those where it is mutable. A good illustration here, rather ironically, is the `List` interface. Clearly, this is designed to be mutable and, furthermore, no equivalent interface is provided for describing immutable lists. Thus, when lists are contained in `HashSets`, or are keys for `HashMaps`, one cannot statically prevent their modification. Instead, this must be prevented at runtime, either by an implicit protocol where the programmer simply does not use these methods, or through an adaptor such as that provided by `Collections.unmodifiableSet()`. In this case, mutable methods provided by the `List` class are often used during initialisation, but not again after this is completed. The following illustrates this common pattern:

```

HashSet<List<String>> hset = ...;
...
ArrayList<String> item;
item = new ArrayList<String>();
item.add("Stephen");
item.add("Dave");

```

```

item.add("James");
// item never changed from now on
hset.add(item);
...

```

An interesting observation here, is that the programmer could have performed all the necessary initialisation via the constructor and, hence, avoided the post-construction identity initialisation. This is done using an array initialiser as follows:

```

HashSet<List<String>> hset = ...;
...
String[] item = {"Stephen",
                 "Dave",
                 "James"};
hset.add(Arrays.asList(item));
...

```

However, from the programmers perspective, there is little advantage to this approach over the other — one seems as convenient as the other.

There are a variety of other well-known reasons for using a post-construction initialisation phase. For example, to reduce the number of parameters required for a constructor, or factory method — as seen in numerous classes from the AWT and SWING libraries. Another common example is the presence of bidirectional references:

```

class Parent { Child child; ... }
class Child { Child parent; ... }

```

Here, we cannot completely initialise both classes via their constructor, because of the bidirectional reference. However, it is not clearly how often the equality of such objects actually depends upon such post-construction initialisation. For example, there would seem to be some legitimate situations where the equality of the parent recursively depends on that of its children — but, one would expect this to be quite unusual.

Finally, a somewhat surprising discovery *is that many classes are mutable when they don't need to be*. Figure 3 illustrates one such example, taken from the open source application `jfreechart`. Again, this has been simplified for the purposes of this presentation, but remains faithful with respect to those issues we are discussing. The key issue is that it provides setter methods for those fields which constitute its equality; however, these are only used when initialising a freshly created object, roughly as follows:

```

AxisSpace space = new AxisSpace();
if(...) {
    space.setTop(...);
    space.setBottom(...);
}

```

Thus, one could have provided an appropriate constructor for `AxisSpace` and created the object *after its initial val-*

```

public class AxisSpace {
    private double top = 0.0;
    private double bot = 0.0;

    public double getTop() {
        return top;
    }
    public void setTop(double t) {
        top = t;
    }
    public double getBottom() {
        return bot;
    }
    public void setBottom(double b) {
        bot = t;
    }
    public boolean equals(Object o) {
        if (!(o instanceof AxisSpace)) {
            return false;
        }
        AxisSpace as = (AxisSpace) o;
        return as.top == top &&
            as.bot == bot;
    }
    public int hashCode() {
        return (int) top + bot;
    }
}

```

**Figure 3.** Illustrating a simple class from the open source application `jfreechart`. Whilst this class does provide mutators, they are, in fact, not necessary — everything could be done via an appropriate constructor.

*ues had been determined.* It seems unclear why the author chose not to do this.

## 2.6 Mutable Equality

We now consider the final strategy outlined previously for dealing with the requirements imposed by equality-dependent collections. Recall that, in this case, the programmer coordinates updates to an object’s equality to ensure they do not occur when it is contained in an equality-dependent collection.

Figure 4 illustrates an example where this is done. In this case, the programmer uses the `routes` `HashMap` as a temporary store for the shortest path determined thus far between two towns (i.e. following Dijkstra’s shortest path algorithm [16]). The system is designed so that towns can be renamed (for some reason); thus, it becomes necessary to ensure that a town is not renamed whilst it is a key for `routes`, otherwise the `Map` contract would be violated.

While the example of Figure 4 is artificial, it does capture the way this strategy typically manifests itself in practice; that is, a temporary `HashMap` or `HashSet` is used during

some algorithm, and changes to the equality of its keys are forbidden whilst this executes. Clearly, employing this strategy correctly is fraught with difficulty. Indeed, we find very few instances of it being used in real-world applications (see §results). We believe the reason for this is that it is simply easier to reason about correctness when the equality of an object essentially never changes.

## 3. #Profiler Overview

To investigate how developers address the contracts imposed by equality-dependent collections, we have developed a custom profiler for Java called `#Profiler` (pronounced “hash profiler”). The tool uses `AspectJ` to add code for tracking information about the target program.

Our profiler is designed to monitor an object’s equality as it changes throughout its life. In particular, we’re interested in detecting the following kinds of object:

1. Objects which are equality-immutable. That is, none of the fields and reachable objects that their equality depends upon changes at any point after the object’s construction.
2. Objects which go through a post-construction equality initialisation phase. That is, the object’s equality is changed briefly after construction.
3. Objects whose equality changes after they have been in a collection (possibly numerous times). This indicates the developer is co-ordinating updates to the object with its inclusion in equality-dependent collections.
4. Objects whose equality changes whilst they are in a collection. If this occurs whilst in an equality-dependent collection, then one can conclude the program does not meet the equality-collection contracts discussed in §2, and that a potential bug exists.

To detect objects in these categories, our profiler makes two kinds of measurement: firstly, it records when changes to an object’s equality occur; secondly, it records when an object is contained in a particular collection. Correlating these two measurements yields useful indications of an object’s category. In particular, if we see no equality changes for an object, then this indicates category (1). Likewise, if all equality changes occur before it is assigned to a field, or passed as a parameter to a method, then this indicates category (2). For category (3), we can simply examine when an object was in a collection, compared with when its equality changed. Similarly, for category (4), the timings of our two measurements tell us when an object has gone into a collection, emerged, and then changed its equality.

We will now outline how our profiler makes these two measurements, before discussing the limitations of this categorisation approach.

```

class Town {
    public String name;
    public String county;

    public boolean equals(Object o) {
        return o instanceof City &&
            ((City)o).name.equals(name) &&
            ((City)o).county.equals(county);
    }

    public int hashCode() {
        return name.hashCode() +
            county.hashCode();
    }
}

class Road {
    public Town from;
    public Town to;
    public int distance;
}

class Network {
    private List<Town> towns = ...;
    private List<Road> connections = ...;
    ...
    public synchronized
    findShortestRoute(Town from, Town to) {
        HashMap<Town, Integer> routes = ...;
        ...
    }

    public synchronized
    void rename(Town town, String name) {
        Town t = towns.find(town);
        t.name = name;
    }
}

```

**Figure 4.** Illustrating example code which coordinates updates to an object's equality (in this case, a `Town`) with its use as a key for a `HashMap`. The key here, is that towns may occasionally be renamed for various reasons. However, renaming a town whilst it is contained in the routes `HashMap` would violate the `Map` contract. Therefore, the programmer ensures that `findShortestRoute` and `rename` are never called concurrently. Note, getters and setters have been omitted for brevity.

### 3.1 Profiling Object Equality

To determine when the equality of an object changes, we monitor its `hashCode()` value. Thus, changes in its `hashCode()` are used to determine that its equality has changed (which follows from the contracts discussed in §2). We choose to monitor `hashCode()`, rather

than `equals()`, because it does not accept parameters and, hence, can be called at will. That is, we can invoke `hashCode()` on an object at any moment we like to check whether or not its value has changed. In contrast, the `equals()` method is more challenging since it requires an object to be compared against. Furthermore, it must be known whether or not the two objects are equal (otherwise, how can we tell whether the identity has changed!). Clearly, from our perspective, this would be very challenging.

Roughly speaking, we determine when the `hashCode()` value for an object has changed as follows: immediately after the object is constructed, we cache its `hashCode()` value; then, as the program executes, we monitor updates to relevant fields (including those of reachable objects); when such an update is observed, we invoke `hashCode()` again and compare against the cached value. The results are logged and used to generate a summary of the number of objects in each category discussed above for the application in question.

Determining when an update to some field may affect the `hashCode()` of a particular object is challenging. To do this, we essentially track the full dependence tree for an object's `hashCode()` method (i.e. the objects accessed when this method executes). Whilst this is expensive, it is important to realise that performance is not our main consideration; rather, we're only interested in categorising objects as above.

### 3.2 Profiling Object Containment

The second measurement made by our profiler, is to determine when an object is contained within a collection and, in particular, whether or not it is an equality dependent collection. In this way, we can determine whether changes to an object's equality occur before, during or after it has been contained in some collection. Thus, we can identify objects whose equality changes occur entirely before they enter any collection, which suggests some form of post-construction equality initialisation; likewise, we can see objects whose equality changes inbetween their containment in a collection, which suggests the programmer is carefully coordinating these changes. Finally, we can spot objects whose equality changes whilst contained within an equality-dependent collection, suggesting they do not obey the collection contract properly. We would consider this latter case to be a bug in the program, although to be completely certain would require a detailed analysis of the source code.

To monitor which objects are in which collections, we modify (automatically) the collection objects used in the program to record the necessary information. Whilst this also imposes some overhead, it is fairly mild in this case.

### 3.3 Discussion

There are several points to make about the way our profiler categorises objects. Since it is observing a particular run of a program, those observations may not hold in other runs. However, we generally try to use inputs that exercise our tar-

get programs to a reasonable degree; furthermore, observations which are consistent across different applications suggest something other than coincidence is at play.

For example, consider category (1) above. An object will be placed into this category if our profiler detects no changes to its equality during our run of the program. But, this does not mean it's equality will not change in other runs. However, we are not interested in whether or not a particular object is in fact immutable; rather, we're interested in whether or not there is a large proportion of objects whose equality does not change, and whether or not this observation is seen across all benchmarks.

In some cases, our profiler can produce more definite results. For example, if it observes an object whose equality changes whilst in an equality-dependent collection, then there it is definitely failing to obey the collection contracts. Likewise, if we were to observe lots of objects whose equality changes inbetween being contained within equality-dependent collections, we could safely assume that programmers do this somewhat complex task routinely.

## 4. #Profiler Implementation

AspectJ is a language extension to Java allowing new functionality to be systematically added to an existing program (see e.g. [20, 32]). To this end, AspectJ provides several language constructs for describing where the program should be modified and in what way. The conceptual idea is that, as a program executes, it triggers certain events and AspectJ allows us to introduce new code immediately before or after these points. Under AOP terminology, an event is referred to as a *join point*, whilst the introduced code is called *advice*. The different join points supported by AspectJ include method execution, method call and field access (read or write). We can attach advice to a single join point or to a set of join points by designating them with a *pointcut*.

We have implemented our profiler using AspectJ, since this provides an ideal platform for building profilers [43]. The #Profiler tool uses the AspectJ load-time weaving facility to instrument Java bytecode as it is loaded into the JVM. This provides easy deployment of #Profiler, since it does not require recompiling the target source code and, instead, can be run directly using a command-line script.

In implementing #Profiler, there were several issues we had to address:

1. How to intercept object creation in order to begin recording information about an object.
2. How to intercept object death in order to aggregate data from a given object into our global statistics.
3. How to monitor changes to an object's equality, including those to objects on which it depended.
4. How to monitor an object's containment in a collection, in order to determine how an object's equality changes in relation to this.

We will now examine in more detail each of the technical issues arising in how we achieved the above.

### 4.1 Intercepting Object Creation with AspectJ

Using AspectJ, it is easy enough to intercept object creation points, using advice similar to the following:

```
after () returning (Object o) :
    call(*.new(..)) && !within(profiler.*) {
    ...
}
```

Here, *after* advice is used, which will be run after the object's constructor. The pointcut for the advice is `call(*.new(..))` which matches any call to a constructor, although we add `!within(profiler.*)` to prevent intercepting object creations arising in our profiler. One problem here, is that this will not match creation points arising from code within Java's standard library, as AspectJ will not weave against these.

### 4.2 Intercepting Object Death with AspectJ

An important issue faced in implementing #Profiler, is how to efficiently maintain the dependence structure between objects and collections. We must record two kinds of dependence information: firstly, whether changes to an object reachable from another can affect its `hashCode()`; secondly, whether an object is currently contained within a particular collection or not.

A key difficulty here, is that garbage collection can affect this structure. For example, a collection object may become unreachable and subsequently garbage collected, hence requiring updates to the profile information recorded for the objects it contained. The difficulty lies in developing a notification mechanism for when an object is collected. In Java there are two obvious constructs to use: *weak references* and *finalizers*. The latter approach turns out to be rather difficult to implement in AspectJ, since one cannot add a *finalizer* to a class which already has one; furthermore, *finalizers* are not always accurate since they can resurrect their objects! Therefore, we choose to use weak references since these have been used for this purpose before [1, 43].

Weak references have the characteristic that they do not prevent the referenced object, called the *referent*, from being collected. That is, if the only references to an object are weak, it is open to collection. When creating a weak reference, we indicate a `ReferenceQueue` onto which it will be placed (by the garbage collector) when cleared (roughly when the reference is collected). Thus, we have a form of callback mechanism, allowing notification of when the object is collected.

The basic outline of our scheme is now becoming clear: at object creation, we attach a weak reference and associate this with a global reference queue. Then, whenever a thread enters our profiler, we check this queue and purge those

objects which are now dead, aggregating the information recorded for them into our global statistics. Since our profiler is monitoring frequently firing events, such as field writes, there is little lag between entry on the reference queue and it being processed.

### 4.3 Profiling Object Equality with AspectJ

Our #Profiler tool detects changes to an object which may affect the result of `hashCode()`. This includes updates to any of the object's fields, or to those of objects it references. To do this requires building a dependency tree for each object so that the effects of a change on one object can be propagated to all objects which reference it.

Whenever an object is created, our `after` advice (see above) is executed. This, amongst other things, will invoke the object's `hashCode()` method. The aim here is to initialise our dependence information for the object; thus, we record which method calls and field accesses are made during this call (using AspectJ's `cflow` construct). This builds up a dependence tree, allowing us to determine when a subsequent change to some object might affect that object's equality.

As the program runs, our profiler intercepts all field writes (using AspectJ's `set` pointcut). When a field write is intercepted, we identify any objects whose `hashCode()` value depends upon this object, and rerun their `hashCode()` method. By comparing the value returned against the value returned originally, we can determine whether or not the equality of an object has changed. When doing this, we again record all method calls and field accesses made during object's the `hashCode()` call to update its dependence tree.

One issue with the mechanism described is that the `hashCode()` for an object might not be computable immediately after construction. This arises because objects often initialise their fields post-construction. For this reason, we catch any exceptions thrown during the `hashCode()` call used for dependence tree initialisation. If this occurs, we have incomplete dependence information about the object in question; but, this will be updated as new dependencies become known since we will still track changes to those objects in the (incomplete) dependence tree.

### 4.4 Profiling Object Containment with AspectJ

The profiler needs to keep track of which objects are contained in collections in order to correlate changes in an object's equality with its containment in a collection. To monitor whether or not an object is contained within a collection, we employ proxy collection objects. Thus, we intercept collection creation points in the program using AspectJ, and wrap these within special proxy objects which are otherwise indistinguishable from the original object. The proxy objects can then monitor the collection interface, and track objects as they are added and removed from the collection.

We implemented proxy objects for the following: `HashSet`, `HashMap`, `LinkedHashMap`, `Hashtable`, `LinkedHashSet`, `TreeSet`, `TreeMap`, `PriorityQueue`, `ArrayList`, `LinkedList` and `Vector`.

To do this, we simply extended each class directly. So, for example, we implemented a `ProxyHashMap` by extending directly from the `HashMap` class. Doing this has some interesting advantages: firstly, it gives good performance since method calls do not cross multiple object boundaries; secondly, any `instanceof` tests used in the program on collections will still return the same results; finally, many of the collection classes provide mutable views (e.g. `HashMap.values()`) and dealing with this is essentially for free. However, it is possible for a program's behaviour to be altered by our use of proxy objects. This could happen if the program uses reflection, and then does specific things based on the actual name of a class — but, we expect this to be extremely unlikely in practice. Also, a problem arises when the program itself implements classes which extend from the standard collections. In such cases, we exploit AspectJ's `declare parents` functionality to introduce a new class into the hierarchy between the collection class and the application-defined subclass. For the most part, this trick works well; however, in a small number of cases it fails because of an outstanding bug in the AspectJ compiler. We have manually inspected these cases, and conclude that they would not adversely affect our results.

Finally, we must repeat the same caveat as before that we are unable to intercept collection object creations arising from code within the Java standard library itself because AspectJ cannot weave against it.

### 4.5 Other Issues

A challenging task for the #Profiler tool is the problem of efficiently associating state with objects during profiling (see [43] for more on this). Such state is used to record the last `hashCode()` value for an object, and to record which collections currently contain it. One approach we considered was to use AspectJ's *intertype declaration* mechanism for inserting fields into a class (and, hence, all of its instances). However, in our case, we wished to access that state after the object was garbage collected. Therefore, we instead employed a global store containing a simple record for each active object. This was implemented using a `HashMap` keyed on object address. Thus, when an object is created, our `after` advice (discussed above) creates an appropriate record for the object in question, and enters it into the global store. Then, when some event of interest is intercepted (for example, an object enters a collection), we must perform a hash-lookup to access the object's record. While this clearly affects performance, we are not so concerned with this as our focus lies solely on generating the necessary profiling data. Furthermore, since each record exists independently of the object it mirrors, we can still access it after that object is collected.

Most Java programs are concurrent, at least to some degree. The profiler, however, relies on being able to compute

hash codes after a change without other changes occurring while it is doing so. To ensure this, the profiler uses monitors to ensure that only one thread at a time can be within the profiler. This does not guarantee the program is thread-safe; another thread may still change a value while another thread is inspecting that object, but the first thread must then enter the profiler and wait for the other thread to finish before continuing. Thus, there is certainly potential for deadlock when inspecting programs with the profiler. For example, a thread holding a particular resource while waiting to enter the profiler may prevent the thread within the profiler from accessing that resource. We have not, however, detected any instances of this occurring.

Finally, it's worth mentioning how we determine that an object implements a non-trivial `hashCode()` method, since this is used to simplify our results in §5. Essentially, we conclude that an object implements a non-trivial `hashCode()` method if its dependence tree is non-empty. So, for example, an object whose `hashCode()` method reads some of its fields will have a non-empty dependence tree; however, an object whose `hashCode()` method returns a constant, or where the default `hashCode()` implementation is used, will have an empty dependence tree.

## 5. Results

To test our hypotheses we ran our profiler on a sample of applications from the Qualitas Corpus developed at Auckland University, NZ [47]. The Qualitas Corpus brings together a larger number of open source Java applications to aid empirical research on Java. However, as the corpus was designed primarily for static analysis, not all of the applications could be profiled. Also, many are, in fact, libraries, platforms, or otherwise, and as such are unsuitable for profiling. The primary requirement for profiling is that the application can be run on a standard JVM, and could be weaved with AspectJ. The version of AspectJ we used (1.6.2) has some limitations. For example, it can cause particularly large code blocks to exceed the allowable code size, resulting in invalid classes. Of the 100 projects in the Qualitas Corpus, we chose a sample population of 30 which were relatively simple to profile. These included compilers, command-line utilities, graphical tools, sample applications for libraries, and test suites. The complete list of applications profiled, with a short description of each, is presented in Figure 5.

### 5.1 Collection Results

The profiler results are presented in Figures 6-9. These results were obtained by running various Java programs within a standard Java HotSpot(TM) Server VM (build 1.5.0\_15-b04, mixed mode) on an Intel machine running NetBSD 5.0\_RC2. The programs were loaded using AspectJ's class-loader which weaves our profiler code written in AspectJ into the classes as they are loaded.

Application	Synopsis
ant	Ant is a Java build system. Benchmarked building ant, included javac.
antlr	Antlr is a compiler-generator. Tested compiling Java grammar.
aoi	Art of Illusion, a 3D editor with raytracer. Built a simple model and rendered it.
columba	Java mail client. Connected to an imap server, browsed mail and sent a message.
derby	Java database. Ran tutorial on in-memory DB.
drawswf	SWF animation editor. Generated a small animation and exported to SWF.
fitjava	Testing framework. Ran tests distributed with framework.
freecs	Chat server. Ran server and connected several clients.
ganttproject	Graphical tool for task management.
hsqldb	Database tool. Created in-memory database and run various test scripts.
itext	Collection of tools for PDFs. Ran several tools.
jFin_DateMath	Date math library. Ran tests.
jasml	Java assembly compiler. Bootstrapped.
javacc	Java Compiler Compiler. Compiled JavaCC grammar.
jchempaint	Graphical molecule editor. Created and edited simple molecules.
jedit	Text editor. Created java class, edited, searched, saved etc.
jfreechart	Graphical tool for creating charts. Tested UI.
jgraph	Library for drawing graphs. Ran several examples.
jgraphpad	Uses jgraph for drawing graphs. Created small graphs.
jgrapht	Views graphs, uses jgraph.
jhotdraw	Graphics framework. Tested sample application.
jmoney	Personal finance. Created sample accounts. Tested import/export, saving, editing, and reporting.
nekohtml	HTML parser. Ran samples.
pmd	Source code analyser. Tested on various projects.
pooka	Java email client. Tested connecting to IMAP server, reading mail, sending mail.
velocity	Templating engine. Ran sample application.
weka	Data mining tool. Ran sample application.
xalan	XSLT processor. Ran some examples.
xerces	XML parser. Ran some examples.
xmojo	JMX implementation. Ran sample application.

Figure 5. Sample Applications

	hash collections			lists			sorted collections			changes						other			
	in	out	value	in	in	out	value	init	list	sorted	hash	value	errors	object	sum	#defined	imm.	objects	classes
ant	75641	72600	24511	83100	0	0	0	31431	0	0	0	8	0	245861	3584954	19	194	2769194	367
antlr	1218	180	1183	5876	0	0	0	2	0	0	0	551	0	694	14618	2	20	49581	72
aoi	17	7	120	1391	0	0	0	127	2	0	0	11	0	3884	218573	3	225	122086	300
columba	127	74	1624	1923	0	0	0	452	541	0	0	3	0	2495	11770	31	452	34402	607
derby	169	40	259	465	0	0	0	54	0	0	0	0	0	178	1040	35	159	4238	280
drawswf	109	0	118	1235	145	0	145	0	53	0	0	0	0	194	1562	0	126	5887	146
fitjava	0	0	3	0	0	0	0	1	0	0	0	0	0	2	18	0	15	739	22
freecs	2	2	261	19	0	0	0	6	0	0	0	0	0	33	1342	7	87	3763	106
ganttproject	113	105	411	7822	0	0	0	488	0	0	0	0	0	722	13553	4	235	37397	294
hsqldb	0	0	0	0	0	0	0	2064	0	0	0	0	0	2064	2064	3	65	27014	116
itext	794	0	842	2669	2	0	6	5	0	0	0	0	0	339	4061	3	60	15928	77
jFin_DateMe	0	0	0	90	0	0	0	41	0	0	0	0	0	41	90	0	9	881	10
jasml	0	0	201	616	0	0	0	19	0	0	0	0	0	22	1523	0	27	3565	32
javacc	942	0	1503	11865	0	0	0	2240	19	0	0	3	0	2409	6884	19	17	74335	50
jchempaint	889	13	3772	4461	0	0	856	10411	0	0	0	196	0	15834	73752	2	181	860622	241
jedit	287	247	1922	2151	1	1	1	6720	0	0	0	584	0	8693	23650	7	374	105912	501
jfreechart	3	0	11	316	2	0	2	35	0	0	0	0	0	83	982	39	103	2524	140
jgraph	65	28	735	1700	0	0	0	2057	0	7437	0	2	0	2367	9495	2	91	74372	117
jgraphpad	876	198	2163	7251	0	0	0	5089	0	0	0	4	0	5887	28767	2	212	90466	257
jgraphpt	30	8	268	813	0	0	0	769	0	0	0	0	0	947	2672	4	77	18591	94
jhotdraw	1	1	40	353	0	0	0	0	0	0	0	0	0	41	64	1	127	48073	164
jmoney	36	0	307	2669	0	0	0	24	0	0	0	4	0	2692	4662	3	254	33514	297
nekohtml	0	0	2	5	0	0	0	0	0	0	0	0	0	1160	7268	3	36	9215	55
pmd	29748	1106	1208	97683	1	0	0	3738	19	0	0	803	0	19465	116172	11	100	776269	216
pooka	2632	0	6943	13684	0	0	2	322	0	0	0	8	0	2077	34878	0	370	58965	440
velocity	1	0	54	2	0	0	0	19	0	0	0	15	0	53	194	4	44	1302	70
weka	10	0	28	10	0	0	0	0	0	0	0	0	0	15	138	1	25	606341	41
xalan	45	0	213	526	0	0	0	0	0	0	0	0	0	16	525	5	83	3050	161
xerces	385	0	196	440	0	0	0	42	0	0	0	0	0	155	1082	8	97	2339	169
xmojo	26	0	24	45	0	0	0	8	0	0	0	0	0	26	96	2	76	1293	95
<b>Totals</b>	<b>114166</b>	<b>74609</b>	<b>48922</b>	<b>249180</b>	<b>151</b>	<b>1</b>	<b>1012</b>	<b>66164</b>	<b>634</b>	<b>0</b>	<b>0</b>	<b>2192</b>	<b>0</b>	<b>318449</b>	<b>4166449</b>	<b>220</b>	<b>3941</b>	<b>5841858</b>	<b>5537</b>

Figure 6. Experimental data for all objects encountered in the target program.

	hash collections			lists			sorted collections			changes						other			
	in	out	value	in	in	out	value	init	list	sorted	hash	value	errors	object	sum	#defined	imm.	objects	classes
ant	75641	72600	24499	83100	0	0	0	29844	0	0	0	0	0	236914	3156993	16	189	2742004	362
antlr	1218	180	630	5876	0	0	0	0	0	0	0	0	0	2	7256	1	17	48308	69
aoi	17	7	109	1387	0	0	0	2	0	0	0	0	0	1464	214689	3	219	119399	294
columba	127	74	1604	1923	0	0	0	156	541	0	0	0	0	1220	5439	29	447	31011	602
derby	169	40	259	465	0	0	0	52	0	0	0	0	0	114	495	33	149	4044	269
drawswf	109	0	118	1182	145	0	145	0	0	5324	0	0	0	0	0	0	120	5324	140
fitjava	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	12	732	19
freecs	2	2	261	19	0	0	0	0	0	0	0	0	0	0	0	5	84	3569	103
ganttproject	113	105	404	7822	0	0	0	0	0	0	0	0	0	0	0	4	229	35939	288
hsqldb	0	0	0	0	0	0	0	2064	0	0	0	0	0	2064	2064	3	63	26997	114
itext	794	0	842	2669	2	0	6	0	0	1507	0	0	0	0	0	1	55	15507	72
jFin_DateMe	0	0	0	90	0	0	0	0	0	0	0	0	0	0	0	0	8	840	9
jasml	0	0	201	616	0	0	0	0	0	0	0	0	0	0	0	0	24	3542	29
javacc	942	0	1383	11432	0	0	0	2111	0	0	0	0	0	2111	4346	17	15	52368	48
jchempaint	889	13	2744	3456	0	0	856	0	0	0	0	0	0	0	0	1	172	829765	232
jedit	287	247	1334	2151	1	1	1	2101	0	0	0	0	0	2884	3998	5	366	82353	493
jfreechart	3	0	11	316	2	0	2	9	0	0	0	0	0	12	49	37	99	2405	136
jgraph	65	28	733	1700	0	0	0	0	0	0	0	0	0	0	0	1	86	70309	112
jgraphpad	876	198	2097	7251	0	0	0	0	0	0	0	0	0	0	0	1	206	79908	251
jgraphpt	30	8	268	813	0	0	0	11	0	0	0	0	0	11	16	2	70	16814	87
jhotdraw	1	1	40	353	0	0	0	0	0	0	0	0	0	0	0	1	124	38154	161
jmoney	36	0	287	2669	0	0	0	0	0	0	0	0	0	2541	2541	2	249	33177	292
nekohtml	0	0	2	5	0	0	0	0	0	0	0	0	0	1154	6858	3	34	9204	53
pmd	29748	1106	33	97140	1	0	0	3	0	0	0	0	0	3	18	8	94	503565	210
pooka	2632	0	6701	13684	0	0	2	0	0	0	0	0	0	0	0	0	363	51848	433
velocity	1	0	38	2	0	0	0	0	0	0	0	0	0	0	0	3	39	1186	65
weka	10	0	28	10	0	0	0	0	0	0	0	0	0	0	0	1	23	606315	39
xalan	45	0	213	526	0	0	0	0	0	0	0	0	0	1	224	4	80	2786	158
xerces	385	0	195	440	0	0	0	38	0	0	0	0	0	105	170	7	93	2196	165
xmojo	26	0	24	45	0	0	0	0	0	0	0	0	0	0	0	1	72	1238	91
<b>Totals</b>	<b>114166</b>	<b>74609</b>	<b>45061</b>	<b>247142</b>	<b>151</b>	<b>1</b>	<b>1012</b>	<b>36391</b>	<b>541</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>250600</b>	<b>3405156</b>	<b>189</b>	<b>3801</b>	<b>5420807</b>	<b>5396</b>

Figure 7. In this case, data excludes collection objects. For example, when comparing comparing the results for ant with Figure 6, we see that 31431 objects were initialised post-construction, of which 29844 were not collection objects.

	hash collections			lists	sorted collections			changes								other			
	in	out	value		in	out	value	init	list	sorted	hash	value	errors	object	sum	#defined	imm.	classes	
ant	75641	71193	3548	83100	0	0	0	29710	0	0	0	0	0	0	29710	29710	9	17	37
antlr	1218	180	208	5876	0	0	0	0	0	0	0	0	0	2	7256	1	2	6	
aoi	17	7	95	1391	0	0	0	117	2	0	0	9	0	2388	3534	0	45	84	
columba	127	74	1023	1923	0	0	0	74	541	0	0	0	0	623	1401	3	39	62	
derby	169	28	223	465	0	0	0	1	0	0	0	0	0	2	283	8	12	31	
drawswf	109	0	109	1235	145	0	0	0	53	0	0	0	0	61	61	0	15	19	
fitjava	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
freecs	2	2	7	19	0	0	0	0	0	0	0	0	0	0	0	2	1	6	
ganttproject	113	105	182	7822	0	0	0	0	0	0	0	0	0	0	0	0	19	38	
hsqldb	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
itext	794	0	830	2669	2	0	0	0	0	0	0	0	0	0	0	1	7	13	
jFin_DateMa	0	0	0	90	0	0	0	0	0	0	0	0	0	0	0	0	0	1	
jasml	0	0	0	616	0	0	0	0	0	0	0	0	0	0	0	0	1	1	
javacc	942	0	1271	11865	0	0	0	2240	19	0	0	3	0	2409	6884	19	4	30	
jchempaint	889	13	1064	4461	0	0	0	393	0	0	0	196	0	2429	11406	1	7	22	
jedit	287	5	259	2151	1	1	1	2101	0	0	0	0	0	2884	3998	4	18	42	
jfreechart	3	0	1	316	2	0	2	3	0	0	0	0	0	4	39	12	14	24	
jgraph	65	28	507	1700	0	0	0	0	0	0	0	0	0	0	0	1	7	13	
jgraphpad	876	195	1381	7251	0	0	0	0	0	0	0	0	0	0	0	1	16	28	
jgrapht	30	8	215	813	0	0	0	0	0	0	0	0	0	0	0	1	8	12	
jhotdraw	1	0	0	353	0	0	0	0	0	0	0	0	0	0	0	0	43	55	
jmoney	36	0	6	2669	0	0	0	0	0	0	0	0	0	2541	2541	1	5	16	
nekohtml	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	5	
pmd	29748	1106	1175	97683	1	0	0	3735	19	0	0	803	0	6018	71574	7	20	109	
pooka	2632	0	1807	13684	0	0	2	0	0	0	0	0	0	0	0	0	34	54	
velocity	1	0	0	2	0	0	0	0	0	0	0	0	0	0	0	1	1	2	
weka	10	0	0	10	0	0	0	0	0	0	0	0	0	0	0	0	1	1	
xalan	45	0	41	526	0	0	0	0	0	0	0	0	0	0	0	2	4	12	
xerces	385	0	127	440	0	0	0	27	0	0	0	0	0	94	156	4	13	22	
xmojo	26	0	1	45	0	0	0	0	0	0	0	0	0	0	0	1	4	5	
Totals	114166	72944	14080	249180	151	1	5	38401	634	0	0	1011	0	49165	138843	79	357	750	

Figure 8. Data excluding objects which never enter a collection. For example, when comparing comparing results for ant with Figure 6, we see that 81431 objects were initialised post-construction, of which 29710 entered a collection at some point.

	hash collections			lists	sorted collections			changes								other			
	in	out	value		in	out	value	init	list	sorted	hash	value	errors	object	sum	#defined	imm.	classes	
ant	67359	64030	0	69853	0	0	0	29710	0	0	0	0	0	0	29710	29710	9	7	9
antlr	27	0	0	27	0	0	0	0	0	0	0	0	0	2	7256	1	0	1	
aoi	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
columba	0	0	153	696	0	0	0	74	541	0	0	0	0	623	1401	3	0	3	
derby	161	27	118	161	0	0	0	1	0	0	0	0	0	2	283	8	3	8	
drawswf	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
fitjava	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
freecs	2	2	7	9	0	0	0	0	0	0	0	0	0	0	0	2	0	2	
ganttproject	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
hsqldb	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
itext	794	0	182	836	0	0	0	0	0	0	0	0	0	0	0	1	0	1	
jFin_DateMa	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
jasml	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
javacc	301	0	641	2398	0	0	0	2240	19	0	0	3	0	2409	6884	19	2	19	
jchempaint	71	0	0	71	0	0	0	0	0	0	0	0	0	0	0	1	1	1	
jedit	232	0	25	232	0	0	0	2101	0	0	0	0	0	2884	3998	4	0	4	
jfreechart	2	0	0	182	0	0	0	3	0	0	0	0	0	4	39	12	7	12	
jgraph	30	0	0	30	0	0	0	0	0	0	0	0	0	0	0	1	1	1	
jgraphpad	446	8	0	446	0	0	0	0	0	0	0	0	0	0	0	1	1	1	
jgrapht	18	0	0	18	0	0	0	0	0	0	0	0	0	0	0	1	1	1	
jhotdraw	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
jmoney	0	0	0	2541	0	0	0	0	0	0	0	0	0	2541	2541	1	0	1	
nekohtml	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
pmd	29235	926	1175	33131	0	0	0	3735	0	29	0	803	0	5997	71415	7	6	7	
pooka	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
velocity	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	
weka	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
xalan	45	0	41	45	0	0	0	0	0	0	0	0	0	0	0	2	2	2	
xerces	14	0	0	15	0	0	0	27	0	0	0	0	0	94	156	4	2	4	
xmojo	26	0	0	26	0	0	0	0	0	0	0	0	0	0	0	1	1	1	
Totals	98764	64993	2342	110718	0	0	0	37891	560	0	0	806	0	44266	123683	79	35	79	

Figure 9. Data from Figure 8 excluding objects which define only trivial hashCode () method.

Each of the four figures shows different breakdowns of the same data. Figure 6 shows the cumulative results for all objects profiled. Figure 7 presents the same data, but excluding collection objects. Figure 8 shows only those objects which enter collections, and Figure 9 shows only those objects which enter collections and have a non-trivial hash function. All of the tables have the same format.

The figures present three groups of results. The first group details the number of objects detected entering each type of collection profiled (see §4.4). The first seven columns summarise the data that the profiler gathered on collections, separated into three groups: hash collections, lists, and sorted collections. The `in`, `out` and `value` columns respectively give the number of distinct objects which enter, leave, or are stored as a value in a hash collection. The fourth column lists the number of objects that were added to list. We did not profile removing objects from lists as it is not an error to change an object's equality while it is in a list. Columns four to six list the number of objects added, removed from, or stored as a value in a sorted collection. The profiler also records the number of times an object enters a collection — up to ten times per object on average for some programs — but this result was not particularly relevant to our hypothesis so is omitted. The results presented for collections list distinct objects: each object contributes to the count at most once, even if it enters a particular type of collection multiple times.

### 5.1.1 Hash Changes

The central section of the figure presents the major results for this paper; the number of objects which change their hash codes, and where in the program they change. There are seven columns, each reporting the number of distinct objects which change their hash code while in a particular phase of their life-cycle. Objects which use the system-defined hash code will never change their hash code so these results are contributed to only by objects which define a non-trivial hash code (or who extend an object which changes its hash code).

The first column, *init*, records the number of objects which change their hash code while they are in an initialisation phase. We define this using confinement ideas to be while there is a unique reference to the object. To detect this we conservatively end the initialisation phase when the object is passed as a method parameter or stored in a field. This allows us to identify objects which are constructed then immediately initialised in a way that changes their equality, as opposed to objects which are created then passed to another object to be initialised. Note, however, that an object reported in the *init* column may subsequently go on to change its equality. This column only indicates that objects often do immediate post-construction initialisation.

The second, third, and fourth columns show the number of objects which change their hash code after entering a list, sorted collection or hash collection, respectively. When an object enters one of these collections the profiler marks it with a flag that indicates that it has entered one of these

collections and it will contribute to these results if it subsequently changes its hash code, even if it first leaves the collection. The *value* column lists the number of objects which change their hash code after being stored as the value of a map — either a `HashMap` or a `TreeMap`.

The *error* column lists the number of contract violations that the profiler has detected. This is when the an object's `hashCode()` (hence, equality) changes whilst contained in a hash collection (see §2). We also consider a change to the `hashCode()` of an object in a sorted collection to be an error. Note, this case is not as clear, as a change to `hashCode()` does not necessarily invalidate the ordering of the collection, but it is a strong indication that there may be a problem (see §2).

The final two columns, *object* and *sum* list the number of objects which change their hash code, and the number of changes to object hash codes that the profiler detected. These give some perspective to the other results, as well as catching changes which do not fit into any of the other categories.

### 5.1.2 General Properties

The final group of columns contain some properties of interest which give some context for the results. The first column in the group, *#defined*, shows number of classes encountered which had a non-trivial hash code method, as discussed in §4.5.

The second column in this section, *imm.* records the number of classes that the profiler encountered which were *shallow-immutable*, that is, none of the profiled instances of that class changed any of their fields after their constructor returned. This is an indication that for the run of program which was profiled, all the fields in class could have been final. Unfortunately, we do not have the results of a source code analysis to show how many of these classes actually had entirely final fields. This result is interesting as it provides a measure of the number of classes which actually change in the program, and, when combined with the results of our hash analysis, the number of hash-constant classes which were actually entirely constant.

The final two columns record the number of objects and the number of classes which the profiler encountered.

## 5.2 Results Analysis

A total of 30 benchmarks were run through the profiler ranging from chemical drawing tools to compiler-generators. The profiler encountered over 5000 classes and almost 6 million objects. Approximately half a million (around 8%) of those objects entered a collection at some point. Only 220 (around 4%) of the classes encountered by the profiler had a non-trivial `hashCode()` method, but approximately 5.5% of the objects encountered changed their hash code at least once during the program. If we ignore collections (Figure 7), then the number of objects which change their hash codes drops to 4.5%. In fact, if we exclude one application, *ant*, only an astonishing 0.5% of non-collection objects change

their hash code. Many applications do not have any objects which change their hash codes other than collections. Excluding *ant* from the results with collections gives 2% of objects which change their hash codes.

This surprising result continues when we consider when objects change their hash code. Around 20% of objects which change their hash code will do so in their (immediate) initialisation phase (though not necessarily exclusively). If we consider only objects which enter collections then the vast majority of objects change during their initialisation phase, and these changes account for more than 25% of all hash code changes observed.

Hash maps and unsorted lists are both used a reasonable amount in the applications we profiled. Around 4% of objects are stored in lists and about half that enter hash maps. Sorted collections are used very little in the applications in our sample. The number of objects seen as keys is an order of magnitude greater than those used as values, suggesting that different keys are used to store the same value. However, closer inspection reveals that these results are skewed by two applications in particular: *ant* and *pmd*, both of which have far more unique objects as keys than as values. Many other applications have more value objects than keys, suggesting that keys are reused between collections, or the values are updated.

Looking at Figure 6 tells us that 318K objects change equals (ever) and 66K of those will do so during their immediate initialisation phase. Unfortunately can't say for how many their equality never changes again, but we can determine how many don't change their equality having entered a collection. Figure 8 says 25% of changes to objects which enter collections occur during immediate initialisation, and that 78% of object which change hashcode will do so during initialisation.

Finally, looking at Figure 6 tells us several surprising things: first, no objects change their hashcode after entering an equality based collection; this means the reindexing did not happen in any of the programs we profiled. Second, there are very few changes to hashcode for objects in value collections (lists, or values in maps). Looking at Figure 7 goes further and shows that almost all of these objects which change in value collections are collections. There is only one program which has non-collection objects which change in a collection, and closer inspection of the results revealed that this was a single library class that was not being used in an equality manner. These results indicate strongly that programmers do not change hashcodes of objects which enter collections. They also indicate that there is a significant distinction between unitary objects and collections in the programs we studied.

## 6. Related Work and Discussion

Given the general importance of equality and identity to object-oriented programming, it is surprising that there are

relatively few studies of programmers' use of these constructs.

Hovemeyer and Pugh [26], for example, show how very straightforward checks can detect Java equality bugs (such as an incorrect covariant signature for `equals` or a missing definition of `hashCode`) along with many other types of bugs, and report the results of an automatic static study of six Java applications. Rupakheti and Hou [49] present an observational study of the use of equality across five Java applications. Working within the existing Java equality contract (and generally not considering issues of mutability) they identify a number of recurring problems in the definition of equality. The study presented in this paper is both significantly larger, and focused explicitly on the mutability aspects of Java's equality contracts.

In the remainder section we discuss our results in the context of the related work on object identity, initialisation, and profiling.

### 6.1 Object Identity and Equality

Object identity and equality has been studied since the first OOPSLA conference [30]. In the beginning, SIMULA provided only support for identity comparison [7], written `==`, while Smalltalk provides two operators to compare objects: `==` (identity comparison) described as testing “whether two objects are equal”, and `=` (equality) described as testing “whether two objects represent the same component” [19]. Smalltalk's `==` is generally not overridden by programmers while `=` certainly can be overridden. These two operators have survived essentially unchanged as Java's `==` and `equals()` — leading to all the issues we have identified earlier.

MacLennan [37] clearly described the distinction between values and objects in programming languages that we now take for granted: that objects have identity and mutable state, while values are immutable and any identity they possess is merely an implementation detail. Khoshafian and Copeland [30] then provided one of the earliest definitions of object identity, shallow equality, and deep equality. Aiming to encompass databases as well as programming languages, their definitions explicitly incorporate sets and tuples.

Baker [6] presents a very comprehensive conceptual discussion of equality in imperative languages: although phrased in terms of Lisp his discussion is directly relevant to all object-oriented programming languages. Common Lisp, of course, has at least five different equality functions: `eq`, `eql`, `equal`, `equalp`, `=`, along with a range of type specific functions such as `char-equal`, `string-equal`, and `tree-equal` [54]. Baker suggests replacing all these separate notions of equality with single `EGAL` predicate, which is a recursive equality for immutable state terminating with identity comparison for mutable objects.

Grogono and Sakkinen [22] discuss equality in conjunction with object-copying in a C++ like language. There is

clearly a relationship here that we have not addressed: a copy of an object should be equal to the object from which it was copied. Grogono and Sakkinen survey equality operations across a range of language and propose four different equalities: identity; shallow (one-level) equality; infinite deep equality; and a structural equality that distinguishes between cycles and their unfoldings as trees.

Vaziri et al [56] describe Relation Types, special kinds of classes whose equality and hash codes are automatically computed based on their “key” fields, which must be final. Relation Types use hash-consing to ensure that each of their instances are unique as far as values for these key fields are concerned. The resulting equality operation is quite similar to Baker’s EGAL: objects are equal up to mutable state.

Our results tend to support a similar design for equality: we found that mutable objects mostly rely on the default definition of equality as identity (from `java.lang.Object`), and those few objects who did define their own `equals` and `hashCode` — by recursing into their own subsidiary objects — did not change their equality, at least once they entered collections. The main exception we found were the collection objects themselves, most of which (in Java) have mutable equality: and which we found were regularly (and safely) mutated.

Finally, Costanza [14] proposed reifying object identities as explicit *comparands* — both in a language design and as a design pattern. Comparands allow an object’s identity to be changed, so that a decorator, for example, can share the identity of its enclosed target object. Techniques similar to comparands are often used to extend identity comparisons across networked applications, or to implement identity hash codes for objects under copying garbage collectors [13]. Unfortunately, at the programming language level, comparands bring all the issues of equality comparisons and mutable hash codes to collections that are based on object identity. Given that our results show that most Java objects’ equality is immutable — either by those objects being immutable, or by their using identity to implement equality — it is not clear the added complication of mutable comparands would be justified in Java.

## 6.2 Object Initialisation and Immutability

We have described how immutable objects avoid most of the interactions between mutability and equality. Object-oriented programming languages have supported immutability in various ways — many languages, for example, support “final” or “const” fields. CLU [36] also supports immutable versions of primitive data structures — although clusters (classes) are always mutable. A similar design has been adopted in Scala, where the library provides mutable and immutable versions of most collections [40].

More recently, Zibin’s IGJ language [58] provides explicit support for both object and class level immutability, and allows code to be parameterised in mutability. So for example, an IJG map class can require its keys to be im-

mutable, but could permit its values to be either mutable or immutable, and these restrictions will be statically enforced by a generic type system. Östlund et al [42] use an ownership type system to obtain similar flexibility.

Immutable objects must be initialised before they can be used. Fändrich and Xia’s Delayed Types [17] use dynamically nested regions in an ownership-style type system to represent this post-construction initialisation phase, and ensure that programs do not access uninitialised fields. Haack and Poll [24] have show how these techniques can be applied specifically to immutability, and Leino et al [34] show how ownership transfer (rather than nesting) can achieve a similar result. Qi and Myers’ Masked Types [46] use type-states to address this problem by incorporating a list of uninitialised fields (“masked fields”) into object types. Gil and Shragai [18] address the related problem of ensuring correct initialisation between subclass and superclass constructors within individual object. Given that our profiling has shown that the initialisation phase of an object is not bounded by the execution of its constructor, these kinds of type systems should be of benefit to real programs.

Rather than concentrating on whole objects and true immutability, Unkel and Lam [55] consider individual fields: a Stationary Field is one where all writes proceed all reads — that is, where a field is initialised (perhaps multiple times, during or after the constructor) but is not modified thereafter. They present a static corpus analysis study of 26 Java applications, backed by a dynamic analysis of 9 programs, and find that 40-60% of Java fields are stationary. Earlier, Porat et al [44] conducted a similar analysis looking for “deeply immutable” fields (where neither the field itself nor any object reachable from that field is modified after the object’s constructor completes) and found that around 60% of `static` fields were immutable. These results compare with our (dynamic) profile finding that a large fraction of Java objects are immutable after full construction.

Finally, in Java practice, Joshua Block [8] advises programmers to “prefer immutability”, that is to use immutable objects wherever possible, and to ensure constructors create objects fully initialised. While we found many immutable objects in our study, we also found many objects whose life-cycle includes a post-construction initialisation stage, which breaches the letter (if not the spirit) of these guidelines.

## 6.3 Profiling

Profiling is a well known strategy for monitoring execution behaviour, and has been studied extensively in the past. The majority of previous works have focused on monitoring standard metrics, such as execution time (e.g. [21, 2, 3, 57, 10, 9, 53]) and heap usage (e.g. [48, 59, 33, 43]). A well-known example here, is `gprof` [21] which uses a combination of CPU sampling and instrumentation to approximate a call-path profile; that is, it reports time spent by each method along with a distribution of that incurred by its callees.

The rise of the virtual machine and, in particular, just-in-time compilation has motivated much work in profiling (e.g. [57, 5, 4, 31]). For example, Whaley described a system for profiling method execution time in a JIT using a compact representation of calling context [57]. Similarly, Arnold *et al.* use profiling information to guide JIT optimisations, such as method inlining [4].

Numerous works have focused on profiling object lifetimes for pretenuring in virtual machines (e.g. [12, 1, 29, 52]). Hirzel *et al.* studied a suite of benchmarks and concluded that object connectivity correlates strongly with object lifetime [25]. Contrasting with this, others have shown how stack state at the point of object allocation correlates with object lifetime [28]. Singer *et al.* studied a small benchmark suite in an effort to identify good predictions of long-lived objects [52]. Chen *et al.* consider the lifetime of object fields, rather than whole objects, since a field may not be active for the duration of its enclosing object's life; thus, fields with disjoint lifetimes can occupy the same memory, thereby reducing object footprint [11]. Similar work studied field lifetimes for the SpecJVM98 benchmark suite, and found on average a 14% reduction in heap space was possible [23]. Shankar *et al.* profiled Java programs in an effort to identify short-live objects suitable for stack allocation [51]. Dieckmann and Hözle performed a detailed study of the allocation behaviour of the SpecJVM98 benchmarks [15]. Pearce *et al.* evaluated AspectJ as a profiling platform by considering different case studies [43]. They considered profiling execution time, heap usage, object lifetime and more.

Røjemo and Runciman introduced the notions of *lag*, *drag* and *use* to describe the lifetime of objects during execution [48]. Under this terminology, *lag* is the time between creation and first use, *drag* is that between last use and collection, while *use* covers the rest. They focused on improving memory consumption in Haskell programs and relied upon compiler support to enable profiling. Building on this, Shaham *et al.* looked at reducing object drag in Java programs [50].

Perhaps the most relevant work to this paper, is that of Marinov and O'Callahan who considered object equality profiling [38]. Essentially, their aim was to expose situations where two identical objects could be reduced to one, thereby saving memory by avoiding redundant objects. To do this, their tool profiles the heap activity of the a program, and then applies a post-mortem analysis once execution is complete. This analysis essentially examines the object graph, searching for sub-graphs which are structurally equivalent (i.e. isomorphic). They applied their tool to several programs from the SpecJVM benchmark suite, and found that several exhibited large numbers of equivalent objects.

Mitchell presented a novel approach to compacting the typically huge amounts of data generated during profiling [39]. His approach exploits the dominates relation for objects in the heaps. Finally, Potanin *et al.* used the JVMPI

interface [35] to profile object graphs in Java programs, concluding that these exhibit the property of being scale-free [45]. In particular, they observed a power-law distribution for edge degrees in the object graph of large programs. Thus, they found that some objects were very highly connected, whilst most had low connectivity.

## 7. Conclusion

ALL OBJECTS ARE EQUAL  
BUT SOME OBJECTS ARE MORE  
EQUAL THAN OTHERS

(after George Orwell, [41])

Every Java object, one way or another, must participate in equality: it must implement the `equals` and `hashCode` methods according to a relatively straightforward contract. Objects can either inherit the default behaviour from class `Object` — and use their identity as their equality — or can override these methods to provide a more rarefied notion of equality. Objects that will participate in equality dependent collections — in hash sets, as keys in hash maps, or in their close cousins the sorted collections — must fulfil a more arduous contract: that their equality, their `hashCode`, their comparability *must never change* while they are within such a collection.

In this paper, we present the results of a study of Java programs with respect to these contracts. We hypothesized that programers could adopt a range of approaches to fulfilling these contracts, from using equality as their identity; via full immutability; or equality immutability, or ensuring their equality is immutable after construction; or finally to removing and reinserting changed objects in their collections. To test these hypotheses, we built a dynamic analysis tool, `#profiler`, that determines when and how objects are constructed, initialised, and how they fulfil these equality contracts.

Using `#profiler` to investigate 30 applications, we discovered that object's equality generally does not change — because a large majority (85%) of objects' equality is directly derived from their identity. For objects that enter collections that depend upon their identity, this proportion is even greater (90%). Whether objects use identity as equality, or whether they have an overriding definition for equality, we found almost no objects whose equality changed while they were part of a collection dependent upon that equality, such as sets or as hashtable values. This is heartening as such a change would be a bug in the programs we studied!

In contrast, we did discover that a significant (although still relatively small) number of objects changed their equality while they were within collections that did *not* impose the stricter equality contract — such as straightforward Lists, or when stored as values in a hash table. This indicates that some Java programmers, at least, do take advantage of the

different equality contracts offered by different collection classes. Similarly, many objects' equality changes *before* they are entered into a collection, supporting the hypothesis that objects often have an initialisation phase that may outlast their constructor, but that results in a stable equality eventually. On the other hand, we did not detect any objects being removed from a collection, changed, and then reinstated back into the same collection. We also found that the majority of objects whose equality changes after an initialisation phase are themselves collections.

Equality, then, does seem important to Java programmers. More to the point, programmers make quite sophisticated use of equality, and (at least in our sample) generally navigate Java's equality contracts successfully: equality is generally based on fully initialised immutable state, and collections can safely rely on stable equality. Proposals such as Baker's EGAL [6], Relation Types [56], and the various schemes for managing object initialisation [17, 34, 46] may well provide good language support for these programming styles.

The exception to this rule — oddly enough — seem to be the collection objects themselves, whose equality changes much more than other objects. Collections, indeed, are simultaneously more equal than other objects — because they all provide a specialised definition of `equal` — and less equal — because they change more often.

## References

- [1] O. Agesen and A. Garthwaite. Efficient object sampling via weak references. In *Proc. ISMM*, pages 121–126. ACM Press, 2000.
- [2] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. A. Leung, R. L. Sites, M. T. Vandervoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proc. of the Symposium on Operating Systems Principles*, pages 1–14. ACM Press, 1997.
- [3] T. E. Anderson and E. D. Lazowska. Quartz: a tool for tuning parallel program performance. In *Proc. of the ACM Conference on Measurement and modeling of computer systems*, pages 115–125. ACM Press, 1990.
- [4] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of Java. In *Proc. OOPSLA*, pages 111–129. ACM Press, 2002.
- [5] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. In *Proc. OOPSLA*, pages 168–179. ACM Press, 2001.
- [6] H. G. Baker. Equal rights for functional objects or, the more things change, the more they are the same. *OOPS Messenger*, 4(4), Oct. 1993.
- [7] G. M. Birtwistle, O. J. Dahl, B. Myrhaug, and K. Nygaard. *Simula Begin*. Studentlitteratur, 1979.
- [8] J. Bloch. *Effective Java*. Prentice Hall PTR, 2008.
- [9] D. J. Brear, T. Weise, T. Wiffen, K. C. Yeung, S. A. Bennett, and P. H. J. Kelly. Search strategies for Java bottleneck location by dynamic instrumentation. *IEE Proc. — Software*, 150(4):235–241, 2003.
- [10] H. W. Cain, B. P. Miller, and B. J. Wylie. A callgraph-based search strategy for automated performance diagnosis. In *Proc. of the European Conference on Parallel Processing (Euro-Par)*, pages 108–122. Springer-Verlag, 2001.
- [11] G. Chen, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Field level analysis for heap space optimization in embedded Java environments. In A. Diwan, editor, *ISMM'04 Proc. of the Fourth International Symposium on Memory Management*, Vancouver, Oct. 2004. ACM Press.
- [12] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *Proc. of the ACM Conference on Programming Language Design and Implementation*, pages 162–173. ACM Press, 1998.
- [13] P. Constanza and A. Hasse. The comparand pattern: Cheap identity testing using dedicated values. In *Pattern Languages of Program Design 5*, chapter 8, pages 169–188. Addison-Wesley, 2006.
- [14] P. Costanza. *Transmigration of Object Identity*. PhD thesis, Institut für Informatik III, Universität Bonn, 2004.
- [15] S. Dieckman and U. Hoelzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. *Lecture Notes in Computer Science*, 1628:92–??, 1999.
- [16] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [17] M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *Proc. OOPSLA*, pages 337–350, 2007.
- [18] J. Gil and T. Shragai. Are we ready for a safer construction environment? In *ECOOP*, 2009. To Appear.
- [19] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.
- [20] J. D. Gradecki and N. Lesiecki. *Mastering AspectJ : Aspect-Oriented Programming in Java*. Wiley, 2003.
- [21] S. L. Graham, P. Kessler, and M. McKusick. gprof: a call graph execution profiler. In *ACM Symposium on Compiler Construction*, pages 120–126. ACM Press, 1982.
- [22] P. Grogono and M. Sakkinen. Copying and comparing: Problems and solutions. In *Proc. ECOOP*, pages 226–250, 2000.
- [23] Z. Guo, J. N. Amaral, D. Szafron, and Y. Wang. Utilizing field usage patterns for java heap space optimization. In *Proc. of the conference of the Centre for Advanced Studies on Collaborative Research*, pages 67–79. IBM, 2006.
- [24] C. Haack and E. Poll. Type-based object immutability with flexible initialization. Technical Report ICIS-R09001, Radboud University Nijmegen, Jan. 2009.
- [25] M. Hirzel, J. Henkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects. In *Proc. ISMM*, pages 143–156, 2002.
- [26] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA Companion*, 2004.

- [27] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *Proc. PASTE*, pages 13–19. ACM, 2005.
- [28] H. Inoue, D. Stefanovic, and S. Forrest. On the prediction of Java object lifetimes. *IEEE Trans. Computers*, 55(7):880–892, 2006.
- [29] M. Jump, S. M. Blackburn, and K. S. McKinley. Dynamic object sampling for pretenuring. In A. Diwan, editor, *Proc. ISMM*. ACM Press, 2004.
- [30] S. N. Khoshafian and G. P. Copeland. Object identity. In *Proc. OOPSLA*, 1986.
- [31] R. V. Kumar, B. L. Narayanan, and R. Govindarajan. Dynamic path profile aided recompilation in a JAVA just-in-time compiler. In *Proc. of High Performance Computing (HiPC)*, volume 2552 of *LNCS*, pages 495–505. Springer, 2002.
- [32] R. Laddad. *AspectJ in Action*. Manning Publications Co., Greenwich, Conn., 2003.
- [33] W. H. Lee and J. M. Chang. An integrated dynamic memory tracing tool for C++. *Information Sciences*, 151:27–49, 2003.
- [34] K. R. M. Leino, P. Müller, and A. Wallenburg. Flexible immutability with frozen objects. In *VSTTE*, pages 192–208, 2008.
- [35] S. Liang and D. Viswanathan. Comprehensive profiling support in the Java Virtual Machine. In *Proc. of the USENIX Conference On Object Oriented Technologies and Systems*, pages 229–240. USENIX Association, 1999.
- [36] B. Liskov and J. V. Guttag. *Abstraction and Specification in Program Development*. MIT Press/McGraw-Hill, 1986.
- [37] B. J. MacLennan. Values and objects in programming languages. *SIGPLAN Notices*, 17(12):70–79, 1982.
- [38] D. Marinov and R. O’Callahan. Object equality profiling. *SIGPLAN Not.*, 38(11):313–325, 2003.
- [39] N. Mitchell. The runtime structure of object ownership. In *Proc. ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 74–98. Springer, 2006.
- [40] M. Odersky. *Programming in Scala*. Artima, Inc, 2008.
- [41] G. Orwell. *Animal Farm*. Secker & Warburg, 1945.
- [42] J. Östlund, T. Wrigstad, D. Clarke, and B. Åkerblom. Ownership, uniqueness, and immutability. In *TOOLS (46)*, pages 178–197, 2008.
- [43] D. J. Pearce, M. Webster, R. Berry, and P. H. J. Kelly. Profiling with AspectJ. *Software: Practice and Experience*, 37(7):747–777, 2007.
- [44] S. Porat, M. Biberstein, L. Koved, and B. Mendelson. Automatic detection of immutable fields in Java. In *Proc. CASCON*, 1990.
- [45] A. Potanin, J. Noble, M. R. Frean, and R. Biddle. Scale-free geometry in OO programs. *Communications of the ACM*, 48(5):99–103, 2005.
- [46] X. Qi and A. C. Myers. Masked types for sound object initialization. In *POPL*, pages 53–65, 2009.
- [47] Qualitas Research Group. Qualitas corpus release 20080603. <http://www.cs.auckland.ac.nz/~ewan/corpus/> The University of Auckland, June 2008.
- [48] N. Røjemo and C. Runciman. Lag, drag, void and use — heap profiling and space-efficient compilation revisited. In *Proc. ICFP*, pages 34–41. ACM Press, 1996.
- [49] C. R. Rupakheti and D. Hou. An empirical study of the design and implementation of object equality in Java. In *Proc. CASCON*, page 9ff, 2008.
- [50] R. Shaham, E. K. Kolodner, and M. Sagiv. Heap profiling for space-efficient Java. In *Proc. PLDI*, pages 104–113. ACM Press, 2001.
- [51] A. Shankar, M. Arnold, and R. Bodik. Jolt: Lightweight dynamic analysis and removal of object churn. In *Proc. OOPSLA*, pages 127–142. ACM Press, 2008.
- [52] J. Singer, G. Brown, M. Lujan, and I. Watson. Towards intelligent analysis techniques for object pretenuring. In *Principles and Practice of Programming in Java*, Lisbon, Sept. 2007. ACM Press.
- [53] M. Spivey. Fast, accurate call graph profiling. *Software — Practice and Experience*, 34:249–264, 2004.
- [54] G. L. Steele. *Common Lisp the Language*. Digital Press, 2nd edn. edition, 1990.
- [55] C. Unkel and M. S. Lam. Automatic inference of stationary fields: a generalization of Java’s final fields. In *POPL*, pages 183–195, 2008.
- [56] M. Vaziri, F. Tip, S. Fink, and J. Dolby. Declarative object identity using relation types. In *Proc. ECOOP*, volume 4609 of *Lecture Notes in Computer Science*, pages 54–78. Springer, 2007.
- [57] J. Whaley. A portable sampling-based profiler for Java Virtual Machines. In *Proc. of the ACM Java Grande Conference*, pages 78–87. ACM Press, 2000.
- [58] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kiezun, and M. D. Ernst. Object and reference immutability using Java generics. In *ESEC/SIGSOFT FSE*, pages 75–84, 2007.
- [59] B. Zorn and P. Hilfinger. A memory allocation profiler for C and Lisp programs. In *Proc. of the USENIX Conference*, pages 223–237. USENIX Association, 1988.