# Understanding the Impact of Collection Contracts on Design

Stephen Nelson, David J. Pearce, and James Noble

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

**Abstract.** Java provides a specification for a user-defined general purpose equivalence operator for objects, but collections such as `Set` have more stringent requirements. This inconsistency breaks polymorphism: programmers must take care to follow `Set`'s contract rather than the more general `Object` contract if their object could enter a `Set`. We have dynamically profiled 30 Java applications to better understand the way programmers design their objects, to determine whether they program with collections in mind. Our results indicate that objects which enter collections behave very differently to objects which do not. Our findings should help developers understand the impact of design choices they make, and guide future language designers when adding support for collections and/or equality.

## 1 Introduction

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

In this paper we consider the effect of one particular feature of Java on program design. Java's `Object` class provides a specification for defining general purpose object equality. However, Java Collections such as `Set` and `Map` require stronger contracts on the implementation of object equality than the `Object` specification provides.

This paper addresses the question, *how do programmers satisfy equality contracts?* We examine the behaviour of objects in running Java programs, comparing objects in different Collections and outside Collections to identify differences in their design. In particular, we compare objects which enter equality collections such as `Set`, non-equality collections such as `ArrayList`, and objects which do not enter a collection at all.

The contributions of this paper are:

1. A set of object characterisations, based on equality and state mutability, which can be measured at runtime.
2. Design and implementation a runtime profiler, called *#Profiler*, that observes the way objects behave when they are and are not in collections. #Profiler employs AspectJ to intercept field reads/writes, constructors and calls to collections.

3. Results from examining 30 real-world Java programs using *#Profiler*. Our results indicate:
   - Objects which do not enter collections do not change their equality;
   - Objects which enter non-equality collections are much more likely than other objects to change their internal state;
   - Objects which enter equality collections are much less likely to change internal state than objects which enter other collections;
   - Objects which enter equality collections and do change their state are no more likely to change their equality than objects which enter non-equality collections.

The rest of this paper is organised as follows: Section 2 discusses various contracts imposed on equality implementations by Java, particularly those imposed by Collections, and outlines our approach to categorising objects according to the way they address these contracts; Section 3 discusses how the object categorisations are measured with our profiling tool, #Profiler; Section 4 presents our experimental results looking at the behaviour of objects across 30 open source Java applications; Section 5 covers related work and, we summarise our findings in the conclusion. An extended version of this paper is available as a technical report [1].

## 2 Equality for Collections

Every object is inherently distinguishable by its location in memory and many languages, like Java, expose this using an equivalence operator. However, it can be useful for objects to define their own equivalence relation for comparing internal state. In addition to reference comparisons, Java provides `equals(..)` — a method defined on the root of the class hierarchy which subclasses can override to implement their own equivalence relations. The documentation provided for this method states that it must be an equivalence relation, but also that it is consistent — that is, it will return the same result for multiple calls so long as the information it uses does not change [2].

Java also provides the Java Collections API, a group of collections for programmers to use. Almost all of these collections are capable of storing `Objects` directly, without any additional type information, yet several require contracts on `equals()` which are stronger than the requirements imposed by `Object` on the equals method. For example, documentation for `java.util.Set` states:

> *"Note: great care must be exercised if mutable objects are used as set elements. The behavior 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. A special case of this prohibition is that it is not permissible for a set to contain itself as an element."* [2]

As there is no type constraint to prevent mutable objects from entering a `Set`, programmers must take care that they obey this contract or they may encounter subtle bugs in their programs. This paper attempts to discover how much programmers use mutable objects in collections and, if they do, how they avoid violating the additional constraints

that some collections impose. We begin by discussing the collections contracts in more detail, then introduce two categorisations for objects based on equality and state, respectively. We conclude this section by discussing a unified categorisation for objects based on both equality and state which is used for the remainder of the paper.

### 2.1 Collection Contracts

The Java Collections API provides four main interfaces: `List`, `Set`, `Map` and `Queue`. There are also implementations provided and, in some cases, there are several each with different properties.

The `Set` interface imposes a particularly strict contract on the objects it contains: they cannot change while they are in the collection. `Map` requires the same of key objects, but not of value objects. `List`s do not have additional requirements on the objects they contain, but they also have a related note of caution:

> *"Note: While it is permissible for lists to contain themselves as elements, extreme caution is advised: the equals and hashCode methods are no longer well defined on such a list."* [2]

This aside is because `List`s, unlike `Queue`s, implement Java's `equals()` and `hashcode()` methods which depend on the list's contents, recursively calling equals or hashcode on each member. While they do not directly impose a contract on their members, programmers must be aware that if the list is stored in another collection which does impose a contract it will transitively apply to the list's contents.

In the rest of this paper we will refer to objects entering *equality* and *non-equality* collections. Equality collections require that the equality of objects does not change while they are in the collection. These include subclasses of `Set`, and the key-sets of `Map` and `HashTable` subclasses. Non-equality collections are `List`s, `Queue`s, and the value-sets of `Map`s and `HashTable`s.

### 2.2 Measuring Changes to Equality

An object following the contract for equality outlined by `Object` may change its equality at any point in its existence. If it is in a Collection, however, this could be an error. To determine which strategies programmers use to avoid these errors, we track objects throughout their lifetime to determine when they do change. We have identified three measurable stages in the life-cycle of an object which we can use to classify objects based on when they change their equality:

**Construction**: When an object is created the constructor is invoked to initialise the object. Even otherwise immutable objects will assign to fields in this phase, as Java allows objects to write to final fields during the constructor; so the first stage we consider ranges from the beginning to the end of the constructor.

**Initialisation**: After an object is created and the constructor has run, there may still be additional initialisation performed on the object which could change its equality. So long as this happens before the object enters a collection it will not violate any equality contracts, so our second phase is from the end of the constructor until the object first enters a collection. Some objects will never enter a collection and thus never leave the 'initialisation' phase.

| Type of Object | Constructor | Initialisation | Post-Collection |
|---|---|---|---|
| Identity as Equality | | | |
| Initialised Equality | x | | |
| Late-initialised Equality | x | x | |
| Reindexing | x | x | x |

**Fig. 1.** Four types of objects distinguished by their different behaviours in various parts of their life-cycle. *x* denotes possible changes to equality during that phase.

**Post-Collection**: After an object has entered a collection we consider it to be fully initialised; any further changes to its hash code could violate the internal consistency of the collection. A programmer would have to consider the implications of changing an object which is in, or could still be in a collection. The post-collection phase ends when the object is garbage collected or the program terminates.

These three measurable phases of an object's life-cycle lead to the following four categorisations of objects based on their changes to equality, which are also presented in Figure 1:

- **Identity as Equality**: objects in this category do not define a hash code method. They rely on reference equality for participation in equality-based collections.
- **Initialised Equality**: these objects define a hash code, but it does not change after the constructor has completed.
- **Late-initialised Equality**: late initialisation objects are distinguished by changes to their hash code after the constructor has completed but before entering a collection. They may also change their hash code during the constructor.
- **Reindexing**: finally, objects which change their hash code after entering a collection are called reindexing objects. Examples of reindexing objects are: objects which leave a hash-based collection, change their hash code, then re-enter a collection; and, objects stored in collections which do not use equality and change at will. Potentially, there are also objects which violate collection constraints and, hence, are erroneous.

These categories of objects are names for distinguishable groups of objects based on the observation points we have defined. Unless there is a reason to distinguish them, we will group these categories based on whether they change their equality after the constructor. *Identity as Equality* and *Initialised Equality* are referred to as *Immutable Equality* objects. *Late-initialised Equality* and *Reindexing* are referred to as *Mutable Equality* objects.

### 2.3 Measuring changes to State

Objects are free to define their equality based on any or all of their reachable state, so it is interesting to see whether objects change state that is not used by equality when they are in collections. This will give us further insight into the techniques programmers use to satisfy the Collection contracts by showing whether the decision to make an object's equality mutable is made with the implementation of the object in mind.

- **Immutable State** objects do not change their state after the constructor ends. This may be because it does not have any state, or all of its state is final, or there are no accessor methods for changing state, but it could also be coincidental — none of the state happened to change.
- **Mutable State** objects can be observed to change state after the constructor. This could be any field of that object or another object which is reachable from fields of the object.

### 2.4 Classifying Objects

State and Equality measurements together will give us the four broad categories of objects listed below:

| | | Equality | |
|---|---|---|---|
| | | Immutable | Mutable |
| State | Immutable | Immutable | Mutable Equality |
| | Mutable | Mutable State | Fully Mutable |

- **Immutable** is by far the simplest approach to ensuring the collection contracts are preserved. In this case, the programmer simply ensures that the state of any object placed into an equality-dependent collection never changes during its lifetime.
- **Mutable State** requires that an object's equality never changes after the constructor, but allows other state to change. This is simple to implement if the object uses *Identity as Equality*, but more challenging to maintain if the object uses *Initialised Equality*: one strategy would be to use immutable objects to determine equality and annotate the fields containing them as final.
- **Mutable Equality** would occur if the object changed its equality but not its state. This cannot occur because the equality is based on state.
- **Fully Mutable** objects change their state and their equality after the constructor. These objects must still obey collection contracts, so this category includes *Late-initialised Equality* and *Reindexing* objects, both of which satisfy the contracts for equality collections, though each requires more care on the behalf of the programmer than objects with immutable equality.

The collections library itself provides examples of each of the three valid categories: `Sets`, `Lists` and `Maps` are Fully Mutable, while `Queues` are typically Mutable State objects. With appropriate care, `Collections.unmodifiableSet()` can provide an Immutable set.

In this paper, we are interested in exploring how these strategies are used 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. The next section discusses the profiling system and its implementation.

## 3 #Profiler

Detecting direct changes to object equality at runtime is difficult. Equality is inherently a binary operator, so changes to the equality of an object can only be detected by invoking the `equals()` method with another object which was previously equal. Detecting that equality has not changed would require comparing with all other objects, or knowing all possible execution paths and reachable objects, which are not feasible for a runtime profiler.

Instead of using the `equals()` method to detect changes to equality, we use `hashCode()` as a proxy. The `hashCode()` method is a unary operator which can be called without reference to other objects. This is a compromise because Java does not enforce any relationship between `hashCode()` and `equals()`. Java's documentation for developers does however specify that if two objects are equal then they must have the same hash code:

> "*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.*"[2]

That is, if the hash code of a correctly implemented object changes then the equality of that object to other objects has changed also. There are tools available to developers to ensure that they do this correctly [3].

Even if it is correctly implemented, the hash code is not a perfect proxy for equality. It is possible that a change to an object will cause its equality to change but not affect its hash code. However, this is unlikely in practice because good hash code methods are designed to avoid this kind of collision. Thus, hash code serves as a good lower-bound measure of equality changes.

### 3.1 Detecting Changes to Hash Code

Our strategy for detecting changes to an object's hash code has two parts; computing and recording the previous value, and tracking changes to objects which could cause the hash code to change. First, we compute the object's hash code. We track all method calls during this invocation, and record the objects on which methods have been invoked. This gives us a set of dependencies for computing the object's hash code. Once the `hashCode()` method completes we record the value returned.

Detecting changes requires calling the `hashCode()` method again to see whether the returned value has changed. We track all field and array writes, and when they occur we re-compute the hash code of each object which depends on the object which contains the field or array. If the hash code has changed we record the change, and if re-evaluating `hashCode()` invokes methods on objects which haven't already been encountered by that object we register the objects as dependents.

### 3.2 Detecting Changes to State

In addition to tracking changes to hash code, we also track changes to objects' fields and arrays. Tracking changes to an object's hash code requires that we monitor changes

to fields and arrays, so we mark objects whose fields and arrays change after their constructors have completed. Classes for which all instances do not change are recorded as having immutable state. The detected immutability is not deep immutability, which would require traversing all reachable objects (which is beyond the scope of our profiler); a class marked as immutable state is simply shallow-immutable for the set of instances and the run of the program that we encountered.

As a consequence, it is possible that *Mutable Equality* objects will be incorrectly detected. That is, objects which appear to change their equality but not their state. This is because a change to an object's deep state may occur without triggering the profiler's mutability detection. We detect and report this when it occurs.

### 3.3 Profiler Implementation

Our profiler is implemented using the AspectJ load-time weaver to add code around method calls, field accesses, and array accesses. In addition, we have implemented replacement classes for common Java collections which are backed by the standard implementations, but record more information than would be possible using woven versions of the standard collections.

AspectJ is not able to add code to the standard libraries, so changes that occur within the standard libraries are not recorded (except in the case of collections, which we replace with our own implementations), but standard library objects are still observable when used in user code. For this reason we provide results both including and without standard library classes. We are also unable to profile certain applications which use their own class loaders (like Eclipse) or applications which are close to the limit on method size: AspectJ does not support breaking up methods to avoid overflowing method size limit, and as our profiler adds a lot of tracking code, this can result in invalid class files.

## 4 Results

To test our hypotheses we ran our profiler on a sample of applications from the Qualitas Corpus developed at Auckland University, NZ [4]. The Qualitas Corpus brings together a large 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. Some were libraries or platforms which could not run independently, while others could not be profiled due to limitations in the profiler (Section 3.3). Of the 100 projects in the Qualitas Corpus, we chose a sample population of 30 which could be profiled relatively easily. 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 2.

### 4.1 Experimental Method

Each Java program was run 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

| 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. |

**Fig. 2.** Profiled applications. A selection of 30 applications from Qualitas Corpus release 20080603 [4]. Where multiple application versions were available the most recent was used. Where relevant, the table lists the application behaviour that was profiled.

in AspectJ into the classes as they are loaded. For each application we chose a suitable set of input designed to exercise as much functionality as possible, but without consulting source code or profiling coverage. For compilers, build tools and similar we tried to use samples distributed with the application or the application itself, while for GUI tools we run simple workflows, and attempted to use all available features, within reason. The profiler introduces significant overhead to the applications, so some interactive programs were difficult to use, while some autonomous programs ran for several hours.

On termination, the profiler output dumps were captured and stored. The raw results were then run through various scripts to extract the results presented in this section. Additional results are available in the technical report version of this paper [1]. The raw profiler output, and the profiler itself can be obtained by contacting the authors.

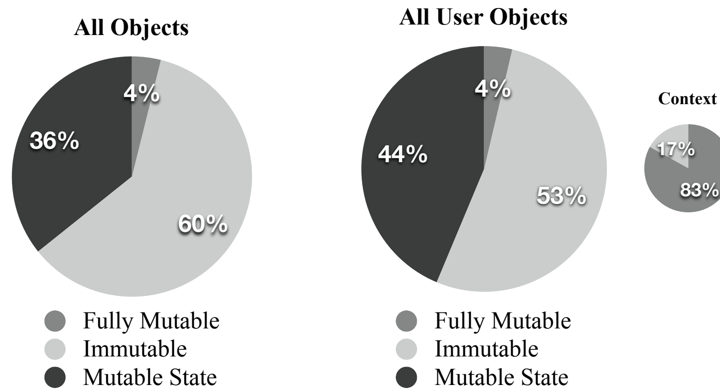


**Fig. 3.** An overview of all of the objects profiled. These are shown split into three categories: objects which change their equality and their state (*fully mutable*), which never change their state (*immutable*), and objects which don't change their equality but do change their state (*mutable state*). The large chart on the right shows the same distribution excluding Java standard library classes, and the small chart indicates how many of the objects in the chart on the left are also in the chart on the right (83%). This figure summarises 8,140,239 objects in 5,577 classes and 30 applications.

### 4.2 Experiment I: General Observations

This experiment provides a general overview of the objects profiled in the 30 sample applications. Figure 3 presents a summary of all objects encountered split into the categories defined previously: *fully mutable*, *immutable* and *mutable state*. These graphs account for the incorrectly detected *mutable equality* objects discussed in Section 3.2 by adding them to the fully mutable segment. See the error section below for a discussion.

The graph on the left of Figure 3 reports the data for all objects profiled, while that on the right only considers *user-defined* classes (i.e. excluding those from the standard libraries). The smaller pie-chart indicates what proportion of objects were user-defined (e.g. 83% of all profiled objects were user defined).

*Discussion.* Our conclusions from the data in Figure 3 are fairly straightforward: very few objects change their equality at all, and there are more objects with immutable state than mutable. This is fairly consistent between user-defined objects and the standard library objects which were profiled.

*Error.* Each segment of the charts in Figure 3 may include an extremely small error due to some objects which have immutable state (shallow) and mutable equality (deep). As we could not determine the exact number of objects in this category from the raw results, we included the error in the fully immutable segment and calculated an upper

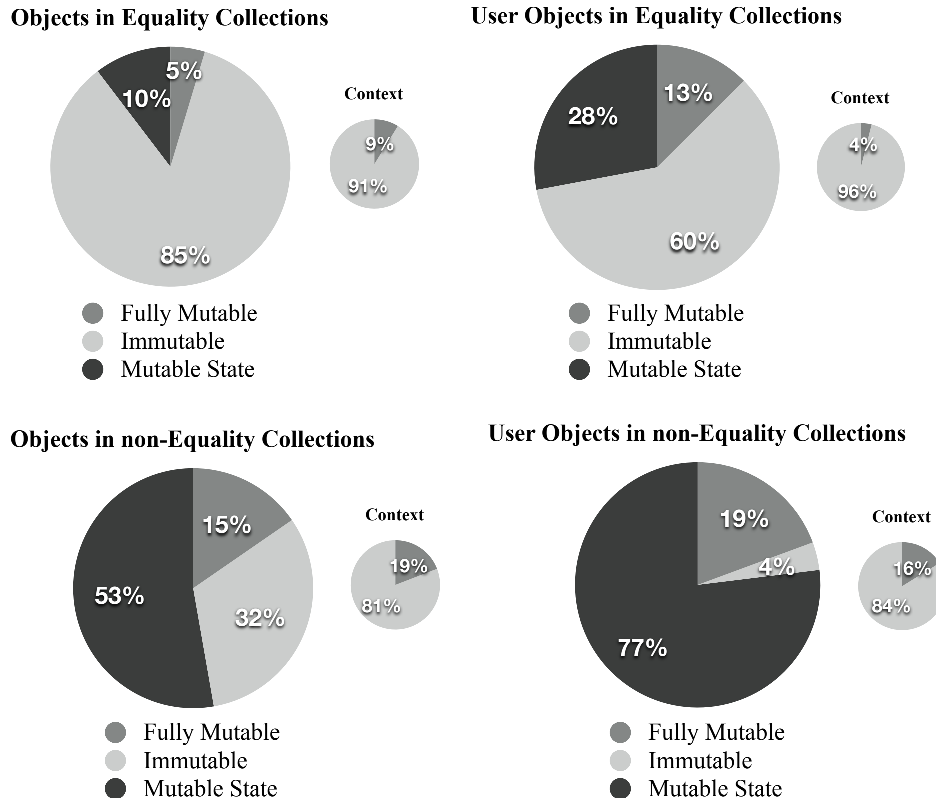


**Fig. 4.** An overview of all of the objects profiled, split into those which enter equality-dependent collections and those which enter non-equality dependent collections. Again, each chart splits into three categories: objects which change their equality and their state (*fully mutable*), which never change their state (*immutable*), and objects which don't change their equality but do change their state (*mutable state*). Equality collections include hash and tree sets, and the key-sets for maps and tables. Non-equality collections include lists, vectors, and queues, as well as value-sets for maps and tables.

bound for the error using the breakdown of classes by program. For all the results presented in this paper, this error never reached one hundredth of a percentage point (197 objects of 5403518 total in the worst case).

### 4.3 Experiment II: Collection Contracts

This experiment examines the behaviour of objects which enter collections, comparing equality collections such as `Set` with non-equality collections like `List`. Figure 4 provides the same categorisations as before for these two categories. The top row of the figure illustrates data for those objects which do enter equality dependent collections, while the bottom row shows data for those which do not. Again, the smaller pie-charts illustrate the relative proportion to all objects (respectively, all user-defined objects).

Thus, we see that only 9% of all objects enter an equality-dependent collection. Likewise, only 4% of all user-defined objects enter an equality-dependent collection.

*Discussion.* The results from Figure 4 demonstrate a clear difference between the behaviour of objects which enter equality collections and those that enter non-equality collections. We surmise that programmers prefer to use immutable objects in equality collections, even though the Collections contract permits them to change fields which do not affect the equality of the object. In particular, there is a large distinction between the number of immutable objects from standard libraries and user code. Further analysis of the results shows that most of these are `Integer` or `String` objects.

When we consider only user-defined objects, the bias towards immutable objects in equality collections is much lower; closer to the proportion in the whole population. This was surprising because these objects are a very small percentage of the whole population, and we expected most of them to be immutable, to easily satisfy the Collections contracts. While this is not the case further analysis of the results showed that objects did not change their equality at all after entering an equality collection. This is not so surprising, but this leads us to conclude that almost all Fully Mutable objects are actually the Late-initialised Equality strategy outlined in Section 2. This could pose a problem for researchers developing type systems for immutability: they will need to support late initialisation, or demonstrate that it can be removed without substantial burden to programmers. There were no broken objects — no objects changed their equality while in a collection.

Objects in non-equality collections show very different characteristics to the general population. The vast majority are not immutable, particularly when standard libraries are excluded, and there are a surprising number which both define and change their equality. The correlation between the relatively large number of objects changing their equality may indicate that programmers make a decision to define equality based on whether an object enters a collection at all, rather than whether the object will enter an equality collection specifically.

### 4.4 Experiment III: Objects in Collections

This final experiment contrasts objects which enter a collection with objects which do not. Figure 5 presents objects which enter a collection on the top row, and objects which do not on the bottom row. Again, the left column contains all objects, while the right column excludes standard library classes, and the small charts indicate the number of objects in each category as a fraction of the whole program.

*Discussion.* These figures show even more clearly the distinction between objects which enter collections and those which do not. The number of immutable objects in the no-collection set is close to the proportion in the general population, while the number of objects which modify their equality disappears completely. This was a very surprising result for us because we expected to see at least some types of objects defining equality unnecessarily. Note that the 1% of mutable state objects in the non-collection graph on the left does not appear on the right; further inspection of the raw results revealed that these are almost exclusively collection objects which define their equality recursively on their contents.

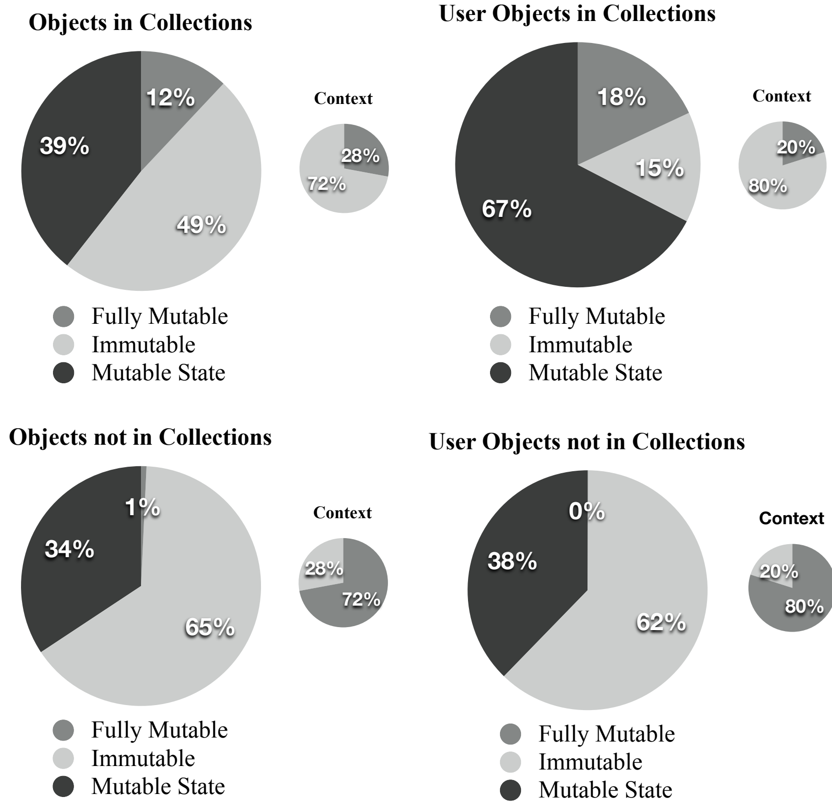


**Fig. 5.** An overview of all of the objects profiled, split into those which enter any collection and those which never do. Again, each chart splits into three categories: objects which change their equality and their state (*fully mutable*), which never change their state (*immutable*), and objects which don't change their equality but do change their state (*mutable state*).

The conclusion that we draw from these results is that programmers design their objects differently when they are going to enter a collection. This does not seem to be related to the contracts imposed by collections, because the trend is much more pronounced in non-equality collections. We do not have a clear understanding of why this should be. It is possible that the sample of applications has introduced some bias, for example a large proportion of objects were contributed by non-interactive programs like Ant. It would be interesting for future work to split applications by type to see whether the trend is consistent. Even so, programming language designers might consider ways to indicate that particular objects are designed for use in collections, as there seem to be large differences in the way they are used. Authors of optimising compilers could use these results to implement caching policies: the likelihood of an object in a collection changing is much higher than the full population, while objects which do not enter collections are extremely unlikely to change their equality, if they define it at all.

## 5 Related Work

We will now discuss various works of relevance to this paper, split into those relevant to object equality, and those related to profiling.

### 5.1 Object Identity and Equality

Object identity and equality has been studied since the first OOPSLA conference [5]. In the beginning, SIMULA provided only support for identity comparison [6], 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" [7]. 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 [8] first described the distinction between values and objects in programming languages: that objects have identity and mutable state, while values are immutable and any identity they possess is merely an implementation detail. Khoshafian and Copeland [5] 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 [9] 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` [10]. 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 [11] 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 [12] 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.

Hovemeyer and Pugh [3] 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 [13] 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.

### 5.2 Object Initialisation and Immutability

Various OO languages have support for immutability via, for example, *final* or *const* fields. CLU [14] 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 [15].

More recently, Zibin's IGJ language [16] provides explicit support for both object and class level immutability, and allows code to be parameterised in mutability. So for example, an IGJ map class can require its keys to be immutable, 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. [17] use an ownership type system to obtain similar flexibility.

Immutable objects must be initialised before they can be used. Fähndrich and Xia's Delayed Types [18] 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 [19] have shown how these techniques can be applied specifically to immutability, and Leino et al. [20] show how ownership transfer (rather than nesting) can achieve a similar result. Qi and Myers' Masked Types [21] use type-states to address this problem by incorporating a list of uninitialised fields ("masked fields") into object types. Gil and Shragai [22] address the related problem of ensuring correct initialisation between subclass and superclass constructors within individual objects. 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 object immutability, Unkel and Lam [23] consider individual fields: a Stationary Field is one where all writes precede 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. [24] 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, Joshua Block [25] 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.

### 5.3 Profiling

Numerous works have focused on profiling object lifetimes for pretenuring in virtual machines (e.g. [26–29]). Hirzel *et al.* studied a suite of benchmarks and concluded that object connectivity correlates strongly with object lifetime [30]. Contrasting with this, others have shown how stack state at the point of object allocation correlates with object lifetime [31]. Singer *et al.* studied a small benchmark suite in an effort to identify good predictions of long-lived objects [29]. 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 [32]. Similar work studied field lifetimes for the SpecJVM98 benchmark suite, and found on average a 14% reduction in heap space was possible [33]. Shankar *et al.* profiled Java programs in an effort to identify short-live objects suitable for stack allocation [34]. Dieckmann and Hözle performed a detailed study of the allocation behaviour of the SpecJVM98 benchmarks [35]. Pearce *et al.* evaluated AspectJ as a profiling platform by considering different case studies [36]. They considered profiling execution time, heap usage, object lifetime and more.

Röjemo and Runciman introduced the notions of *lag*, *drag* and *use* to describe the lifetime of objects during execution [37]. 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 [38].

Perhaps the most relevant work to this paper, is that of Marinov and O'Callahan who considered object equality profiling [39]. 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 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 [40]. His approach exploits the dominates relation for objects in the heaps. Finally, Potanin *et al.* used the JVMPI interface [41] to profile object graphs in Java programs, concluding that these exhibit the property of being scale-free [42]. In particular, they observed a power-law distribution for edge degrees in the object graph of large programs: some objects were very highly connected, whilst most had low connectivity.

## 6 Conclusion

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

(after George Orwell, [43])

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 may 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 objects' equality generally does not change: with a few exceptions, objects which do not enter collections either do not change or do not define their hash code. Of objects which do enter collections, 19% changed their hash code after the constructor completed.

Surprisingly, objects which enter collections exhibit a strong tendency to change fields which are not used to determine hash code: 77% of user objects do this. Combined with the objects which do change their hash code, only 4% of objects which enter non-equality collections do not change; a huge difference to the general population where well over half are immutable. It is heartening though to find that none of these objects change their equality while actually in an equality collection, as such a change would be a bug in the programs we studied!

Equality, then, does seem important to Java programmers. More to the point, programmers make good use of equality in collections, 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 [9], Relation Types [12], and the various schemes for managing object initialisation [18, 20, 21] may well provide good language support for objects which enter collections, so long as they can cope with the relatively high number of objects performing delayed initialisation; while objects which do not enter collections seem to be adequately served by object identity, as they do not change their equality.

The exception to this rule — oddly enough – seem to be the collection objects themselves, whose equality changes whether or not they are in collections. 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.

**Acknowledgements** Thanks go to the anonymous reviewers for TOOLS who provided valuable insights and comments. Thanks also to Victoria University VC's Strategic PhD Scholarship for supporting this research, and New Zealand's BuildIT for providing funding for the first author to present this paper.

## References

1. Nelson, S., Pearce, D.J., Noble, J.: Understanding the impact of collection contracts on design. Technical Report 10-09, School of Engineering and Computer Science, Victoria University of Wellington, New Zealand (2010)
2. Chan, P., Lee, R.: The Java Class Libraries, Second Edition, Volume 1. Addison-Wesley (1999)
3. Hovemeyer, D., Pugh, W.: Finding bugs is easy. In: OOPSLA Companion. (2004)
4. Qualitas Research Group: Qualitas corpus release 20080603. `http://www.cs.auckland.ac.nz/~ewan/corpus/` The University of Auckland (June 2008)
5. Khoshafian, S.N., Copeland, G.P.: Object identity. In: Proc. OOPSLA. (1986)
6. Birtwistle, G.M., Dahl, O.J., Myhrhaug, B., Nygaard, K.: Simula Begin. Studentlitteratur (1979)
7. Goldberg, A., Robson, D.: Smalltalk-80: The Language and its Implementation. Addison-Wesley (1983)
8. MacLennan, B.J.: Values and objects in programming languages. SIGPLAN Notices **17**(12) (1982) 70–79
9. Baker, H.G.: Equal rights for functional objects or, the more things change, the more they are the same. OOPS Messenger **4**(4) (October 1993)
10. Steele, G.L.: Common Lisp the Language. 2nd edn. edn. Digital Press (1990)
11. Grogono, P., Sakkinen, M.: Copying and comparing: Problems and solutions. In: Proc. ECOOP. (2000) 226–250
12. Vaziri, M., Tip, F., Fink, S., Dolby, J.: Declarative object identity using relation types. In: Proc. ECOOP. Volume 4609 of Lecture Notes in Computer Science., Springer (2007) 54–78
13. Rupakheti, C.R., Hou, D.: An empirical study of the design and implementation of object equality in Java. In: Proc. CASCON. (2008) 9ff
14. Liskov, B., Guttag, J.V.: Abstraction and Specification in Program Development. MIT Press/McGraw-Hill (1986)
15. Odersky, M.: Programming in Scala. Artima, Inc (2008)
16. Zibin, Y., Potanin, A., Ali, M., Artzi, S., Kiezun, A., Ernst, M.D.: Object and reference immutability using Java generics. In: ESEC/SIGSOFT FSE. (2007) 75–84
17. Östlund, J., Wrigstad, T., Clarke, D., Åkerblom, B.: Ownership, uniqueness, and immutability. In: TOOLS (46). (2008) 178–197
18. Fähndrich, M., Xia, S.: Establishing object invariants with delayed types. In: Proc. OOPSLA. (2007) 337–350
19. Haack, C., Poll, E.: Type-based object immutability with flexible initialization. Technical Report ICIS-R09001, Radboud University Nijmegen (January 2009)
20. Leino, K.R.M., Müller, P., Wallenburg, A.: Flexible immutability with frozen objects. In: VSTTE. (2008) 192–208
21. Qi, X., Myers, A.C.: Masked types for sound object initialization. In: POPL. (2009) 53–65
22. Gil, J., Shragai, T.: Are we ready for a safer construction environment? In: ECOOP. (2009) To Appear.

23. Unkel, C., Lam, M.S.: Automatic inference of stationary fields: a generalization of Java's final fields. In: POPL. (2008) 183–195
24. Porat, S., Biberstein, M., Koved, L., Mendelson, B.: Automatic detection of immutable fields in Java. In: Proc. CASCON. (1990)
25. Bloch, J.: Effective Java. Prentice Hall PTR (2008)
26. Cheng, P., Harper, R., Lee, P.: Generational stack collection and profile-driven pretenuring. In: Proc. of the ACM Conference on Programming Language Design and Implementation, ACM Press (1998) 162–173
27. Agesen, O., Garthwaite, A.: Efficient object sampling via weak references. In: Proc. ISMM, ACM Press (2000) 121–126
28. Jump, M., Blackburn, S.M., McKinley, K.S.: Dynamic object sampling for pretenuring. In Diwan, A., ed.: Proc. ISMM, ACM Press (2004)
29. Singer, J., Brown, G., Lujan, M., Watson, I.: Towards intelligent analysis techniques for object pretenuring. In: Principles and Practice of Programming in Java, Lisbon, ACM Press (September 2007)
30. Hirzel, M., Henkel, J., Diwan, A., Hind, M.: Understanding the connectivity of heap objects. In: Proc. ISMM. (2002) 143–156
31. Inoue, H., Stefanovic, D., Forrest, S.: On the prediction of Java object lifetimes. IEEE Trans. Computers **55**(7) (2006) 880–892
32. Chen, G., Kandemir, M., Vijaykrishnan, N., Irwin, M.J.: Field level analysis for heap space optimization in embedded Java environments. In Diwan, A., ed.: ISMM'04 Proc. of the Fourth International Symposium on Memory Management, Vancouver, ACM Press (October 2004)
33. Guo, Z., Amaral, J.N., Szafron, D., Wang, Y.: Utilizing field usage patterns for java heap space optimization. In: Proc. of the conference of the Centre for Advanced Studies on Collaborative Research, IBM (2006) 67–79
34. Shankar, A., Arnold, M., Bodik, R.: Jolt: Lightweight dynamic analysis and removal of object churn. In: Proc. OOPSLA, ACM Press (2008) 127–142
35. Dieckman, S., Hoelzle, U.: A study of the allocation behavior of the SPECjvm98 Java benchmarks. Lecture Notes in Computer Science **1628** (1999) 92–115
36. Pearce, D.J., Webster, M., Berry, R., Kelly, P.H.J.: Profiling with AspectJ. Software: Pracice and Experience **37**(7) (2007) 747–777
37. Röjemo, N., Runciman, C.: Lag, drag, void and use — heap profiling and space-efficient compilation revisited. In: Proc. ICFP, ACM Press (1996) 34–41
38. Shaham, R., Kolodner, E.K., Sagiv, M.: Heap profiling for space-efficient Java. In: Proc. PLDI, ACM Press (2001) 104–113
39. Marinov, D., O'Callahan, R.: Object equality profiling. SIGPLAN Not. **38**(11) (2003) 313–325
40. Mitchell, N.: The runtime structure of object ownership. In: Proc. ECOOP. Volume 4067 of Lecture Notes in Computer Science., Springer (2006) 74–98
41. Liang, S., Viswanathan, D.: Comprehensive profiling support in the Java Virtual Machine. In: Proc. of the USENIX Conference On Object Oriented Technologies and Systems, USENIX Association (1999) 229–240
42. Potanin, A., Noble, J., Frean, M.R., Biddle, R.: Scale-free geometry in OO programs. Communications of the ACM **48**(5) (2005) 99–103
43. Orwell, G.: Animal Farm. Secker & Warburg (1945)