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 `Object`s 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.

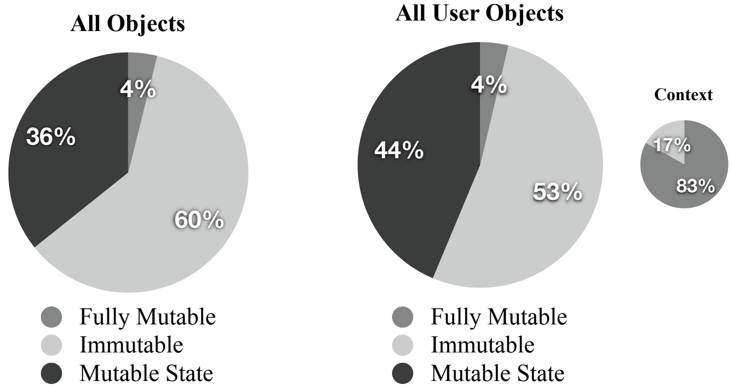


Fig. 3. An overview of all of the objects profiled. These are shown split into three categories: objects which change their equality and their state (*fully mutable*), which never change their state (*immutable*), and objects which don't change their equality but do change their state (*mutable state*). 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

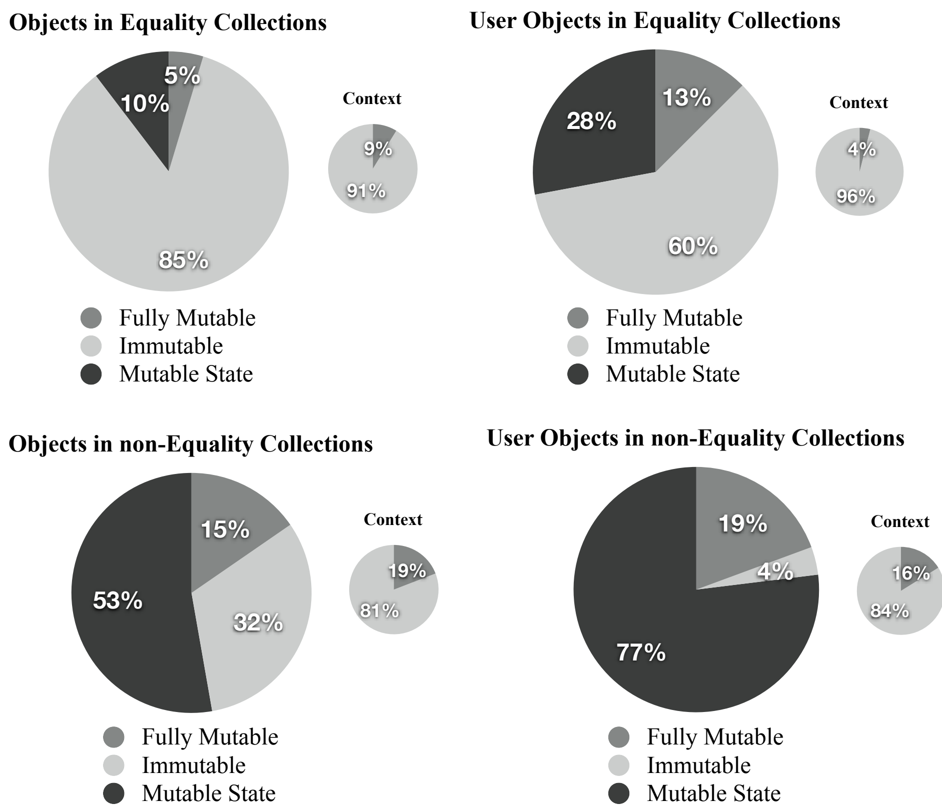


Fig. 4. An overview of all of the objects profiled, split into those which enter equality-dependent collections and those which enter non-equality dependent collections. Again, each chart splits into three categories: objects which change their equality and their state (*fully mutable*), which never 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.

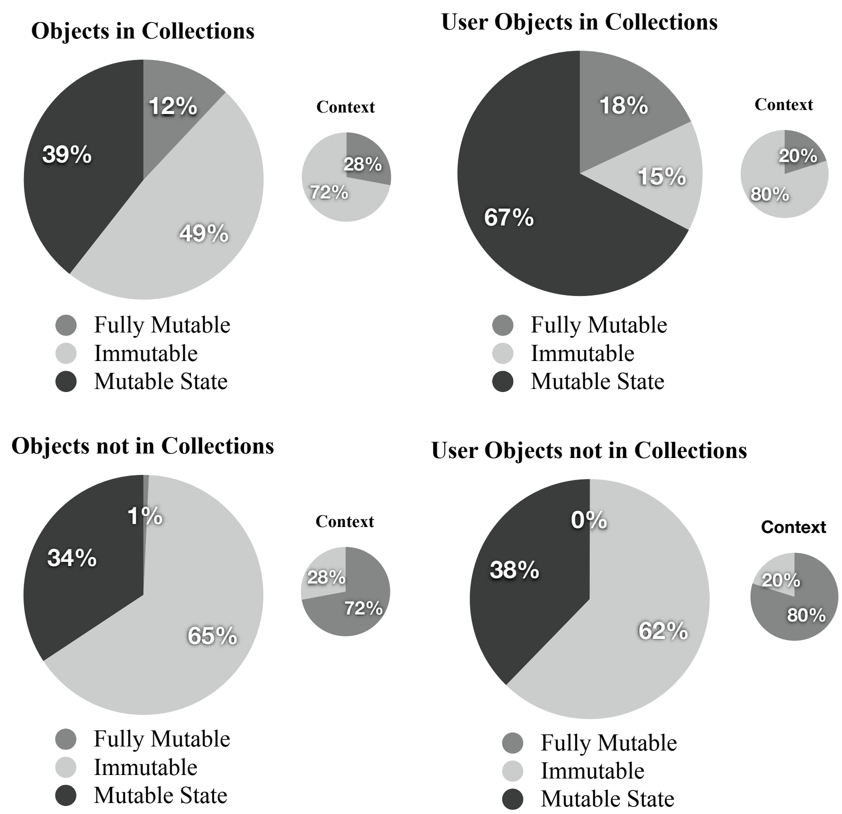


Fig. 5. An overview of all of the objects profiled, split into those which enter any collection and those which never do. Again, each chart splits into three categories: objects which change their equality and their state (*fully mutable*), which never change their state (*immutable*), and objects which 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.

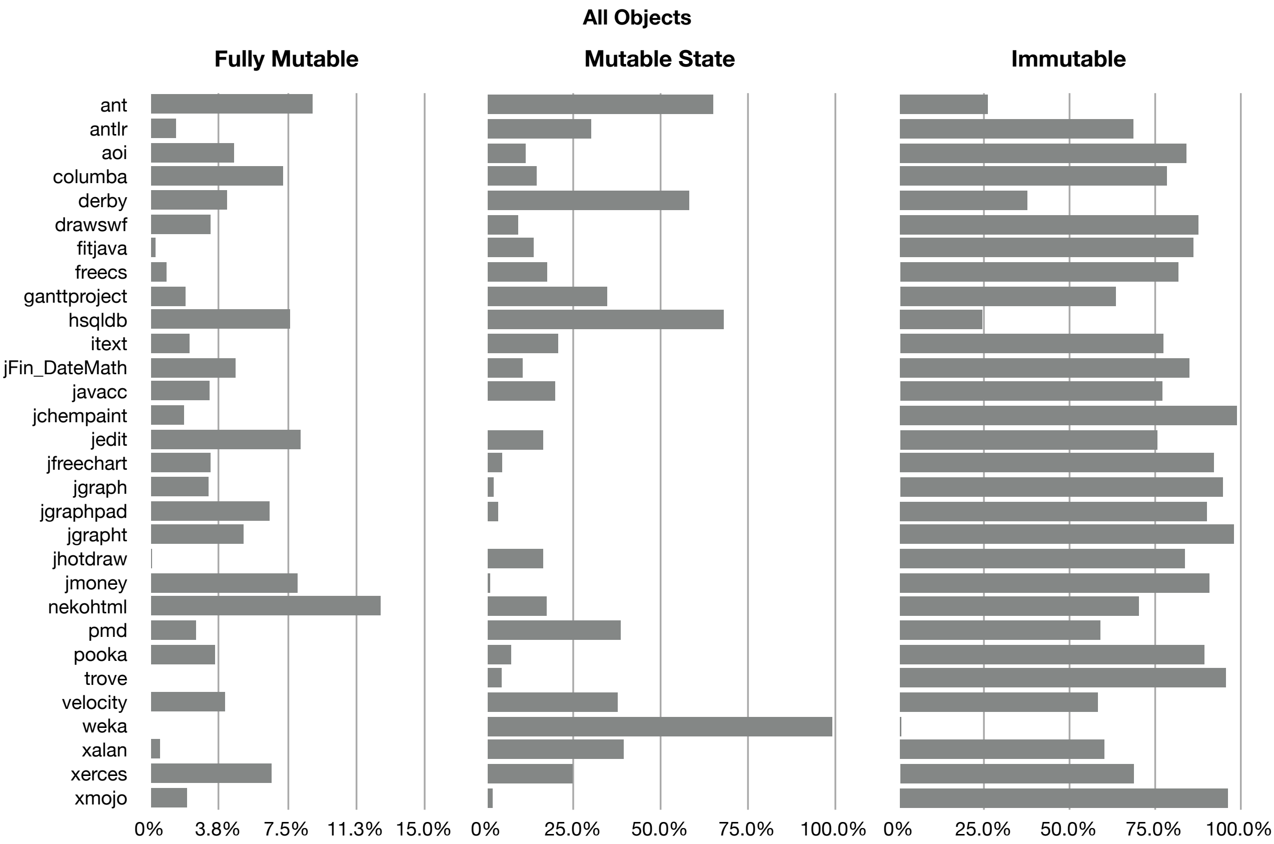
23. Unkel, C., Lam, M.S.: Automatic inference of stationary fields: a generalization of Java's final fields. In: POPL. (2008) 183–195
24. Porat, S., Biberstein, M., Koved, L., Mendelson, B.: Automatic detection of immutable fields in Java. In: Proc. CASCON. (1990)
25. Bloch, J.: Effective Java. Prentice 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)

All Objects encountered

All Objects

	Total	Fully Mutable	Immutable	Mutable State	Inverse	Fully Mutable	Immutable	Mutable State
ant	2769194	245861	720207	1803126	0	8.88%	26.01%	65.11%
antlr	49581	694	34068	14819	0	1.40%	68.71%	29.89%
aoi	23723	1084	19989	2650	0	4.57%	84.26%	11.17%
columba	34402	2495	27018	4889	0	7.25%	78.54%	14.21%
derby	4238	178	1595	2465	0	4.20%	37.64%	58.16%
drawswf	5887	194	5169	524	0	3.30%	87.80%	8.90%
fitjava	739	2	638	99	0	0.27%	86.33%	13.40%
freecs	3762	33	3079	650	0	0.88%	81.84%	17.28%
ganttproject	37397	722	23739	12936	0	1.93%	63.48%	34.59%
hsqldb	27014	2064	6574	18376	0	7.64%	24.34%	68.02%
itext	15928	339	12332	3257	0	2.13%	77.42%	20.45%
jFin_DateMath	881	41	750	90	0	4.65%	85.13%	10.22%
javacc	74335	2409	57334	14592	0	3.24%	77.13%	19.63%
jchempaint	860622	15834	853184	-8396	0	1.84%	99.14%	-0.98%
jedit	105912	8693	80168	17051	0	8.21%	75.69%	16.10%
jfreechart	2524	83	2330	111	0	3.29%	92.31%	4.40%
jgraph	74372	2367	70580	1425	0	3.18%	94.90%	1.92%
jgraphpad	90466	5887	81674	2905	0	6.51%	90.28%	3.21%
jgrapht	18591	947	18255	-611	0	5.09%	98.19%	-3.29%
jhotdraw	48073	41	40287	7745	0	0.09%	83.80%	16.11%
jmoney	33514	2692	30520	302	0	8.03%	91.07%	0.90%
nekohtml	9215	1160	6479	1576	0	12.59%	70.31%	17.10%
pmd	776269	19465	458017	298787	0	2.51%	59.00%	38.49%
pooka	58965	2077	52778	4110	0	3.52%	89.51%	6.97%
trove	2400310	0	2300215	100095	0		95.83%	4.17%
velocity	1302	53	759	490	0	4.07%	58.29%	37.63%
weka	606341	15	3962	602364	0	0.00%	0.65%	99.34%
xalan	3050	16	1836	1198	0	0.52%	60.20%	39.28%
xerces	2339	155	1609	575	0	6.63%	68.79%	24.58%
xmojo	1293	26	1247	20	0	2.01%	96.44%	1.55%
Sum	8140239	315627	4916392	2908220	0	3.88%	60.40%	35.73%
Mean	271341.3	10520.9	163879.7333333	96940.66666667	0	3.95%	73.43%	22.62%
Deviation	668929.4784935	44684.67061868	453792.1490918	344058.5890897	0	3.11%	24.10%	23.94%

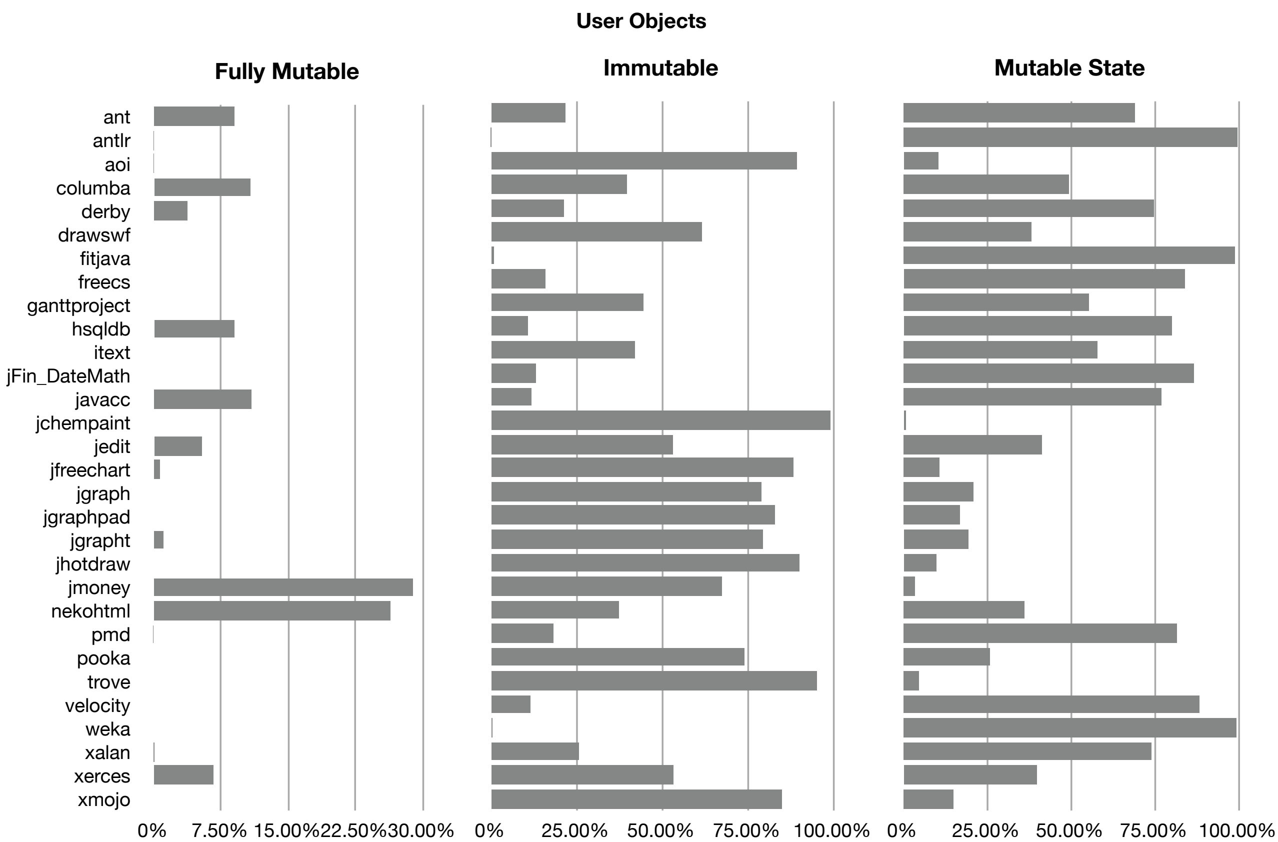
All Objects encountered



All Objects encountered

	User Objects							
	Total	**Fully Mutable**	**Immutable**	**Mutable State**	**Inverse**	**Fully Mutable**	**Immutable**	**Mutable State**
ant	2619783	236914	570796	1812073	149411	9.04%	21.79%	69.17%
antlr	15546	2	33	15511	34035	0.01%	0.21%	99.77%
aoi	18930	2	16935	1993	4793	0.01%	89.46%	10.53%
columba	11299	1220	4486	5593	23103	10.80%	39.70%	49.50%
derby	3360	128	717	2515	878	3.81%	21.34%	74.85%
drawswf	1873	0	1155	718	4014		61.67%	38.33%
fitjava	102	0	1	101	637		0.98%	99.02%
freecs	813	0	130	683	2949		15.99%	84.01%
ganttproject	24553	0	10931	13622	12844		44.52%	55.48%
hsqldb	22930	2064	2490	18376	4084	9.00%	10.86%	80.14%
itext	6204	0	2608	3596	9724		42.04%	57.96%
jFin_DateMath	151	0	20	131	730		13.25%	86.75%
javacc	19304	2111	2303	14890	55031	10.94%	11.93%	77.13%
jchempaint	799010	0	792205	6805	61612		99.15%	0.85%
jedit	53649	2884	28524	22241	52263	5.38%	53.17%	41.46%
jfreechart	1668	12	1474	182	856	0.72%	88.37%	10.91%
jgraph	3782	0	2988	794	70590		79.01%	20.99%
jgraphpad	14852	0	12318	2534	75614		82.94%	17.06%
jgrapht	1634	18	1298	318	16957	1.10%	79.44%	19.46%
jhotdraw	7262	0	6541	721	40811		90.07%	9.93%
jmoney	8785	2541	5930	314	24729	28.92%	67.50%	3.57%
nekohtml	4368	1154	1632	1582	4847	26.42%	37.36%	36.22%
pmd	389493	3	71241	318249	386776	0.00%	18.29%	81.71%
pooka	23803	0	17618	6185	35162		74.02%	25.98%
trove	2100189	0	2000094	100095	300121		95.23%	4.77%
velocity	614	0	71	543	688		11.56%	88.44%
weka	605493	0	3114	602379	848		0.51%	99.49%
xalan	1636	1	422	1213	1414	0.06%	25.79%	74.14%
xerces	1566	105	836	625	773	6.70%	53.38%	39.91%
xmojo	299	0	254	45	994		84.95%	15.05%
Sum	6762951	249159	3559165	2954627	0	3.68%	52.63%	43.69%
Mean	225431.7	8305.3	118638.8333333	98487.56666667	45909.6	3.76%	47.15%	49.09%
Deviation	613422.8301037	43185.7422656	395597.2920291	345832.78686	88056.66000252	7.42%	33.10%	33.25%

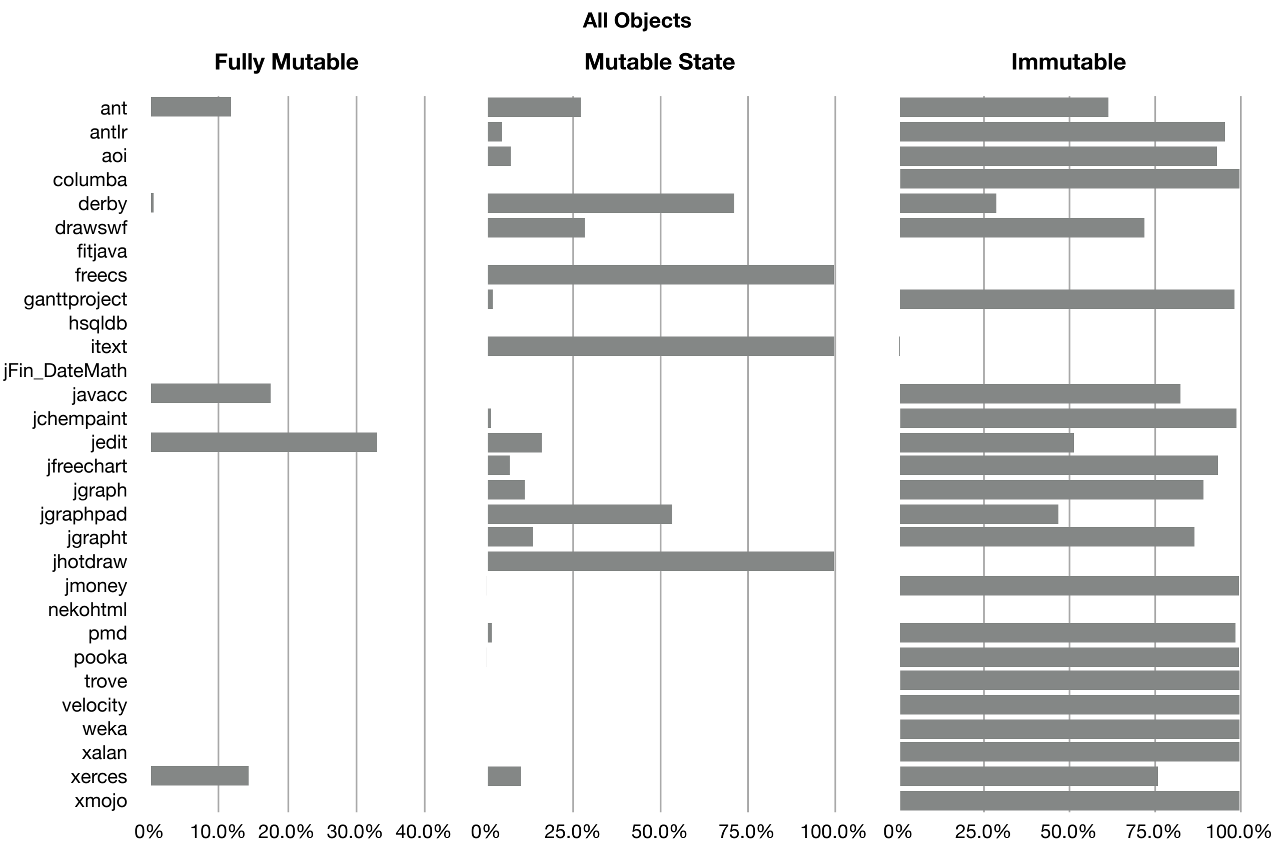
All Objects encountered



Objects in Equality Collections

	All Objects							
	Total	Fully Mutable	Immutable	Mutable State	Inverse	Fully Mutable	Immutable	Mutable State
ant	253110	29710	155418	67982	0	11.74%	61.40%	26.86%
antlr	17028	2	16278	748	0	0.01%	95.60%	4.39%
aoi	104	0	97	7	0		93.27%	6.73%
columba	2381	0	2381	0	0		100.00%	
derby	411	2	117	292	0	0.49%	28.47%	71.05%
drawswf	389	0	280	109	0		71.98%	28.02%
fitjava	0	0	0	0	0			
freecs	2	0	0	2	0			100.00%
ganttproject	730	0	718	12	0		98.36%	1.64%
hsqldb	0	0	0	0	0			
itext	1452	0	2	1450	0		0.14%	99.86%
jFin_DateMath	0	0	0	0	0			
javacc	3062	536	2526	0	0	17.50%	82.50%	
jchempaint	1432	0	1416	16	0		98.88%	1.12%
jedit	8725	2884	4473	1368	0	33.05%	51.27%	15.68%
jfreechart	46	0	43	3	0		93.48%	6.52%
jgraph	65	0	58	7	0		89.23%	10.77%
jgraphpad	1559	0	728	831	0		46.70%	53.30%
jgrapht	30	0	26	4	0		86.67%	13.33%
jhotdraw	1	0	0	1	0			100.00%
jmoney	12210	0	12208	2	0		99.98%	0.02%
nekohtml	0	0	0	0	0			
pmd	91248	0	89985	1263	0		98.62%	1.38%
pooka	20787	0	20781	6	0		99.97%	0.03%
trove	300000	0	300000	0	0		100.00%	
velocity	2	0	2	0	0		100.00%	
weka	20	0	20	0	0		100.00%	
xalan	47	0	47	0	0		100.00%	
xerces	658	94	499	65	0	14.29%	75.84%	9.88%
xmojo	188	0	188	0	0		100.00%	
Sum	715687	33228	608291	74168	0	4.64%	84.99%	10.36%
Mean	23856.23333333	1107.6	20276.36666667	2472.266666667	0	2.57%	65.74%	18.35%
Deviation	71023.91825278	5428.197018253	61811.58372103	12380.61658799	0	7.29%	40.97%	32.17%

Objects in Equality Collections

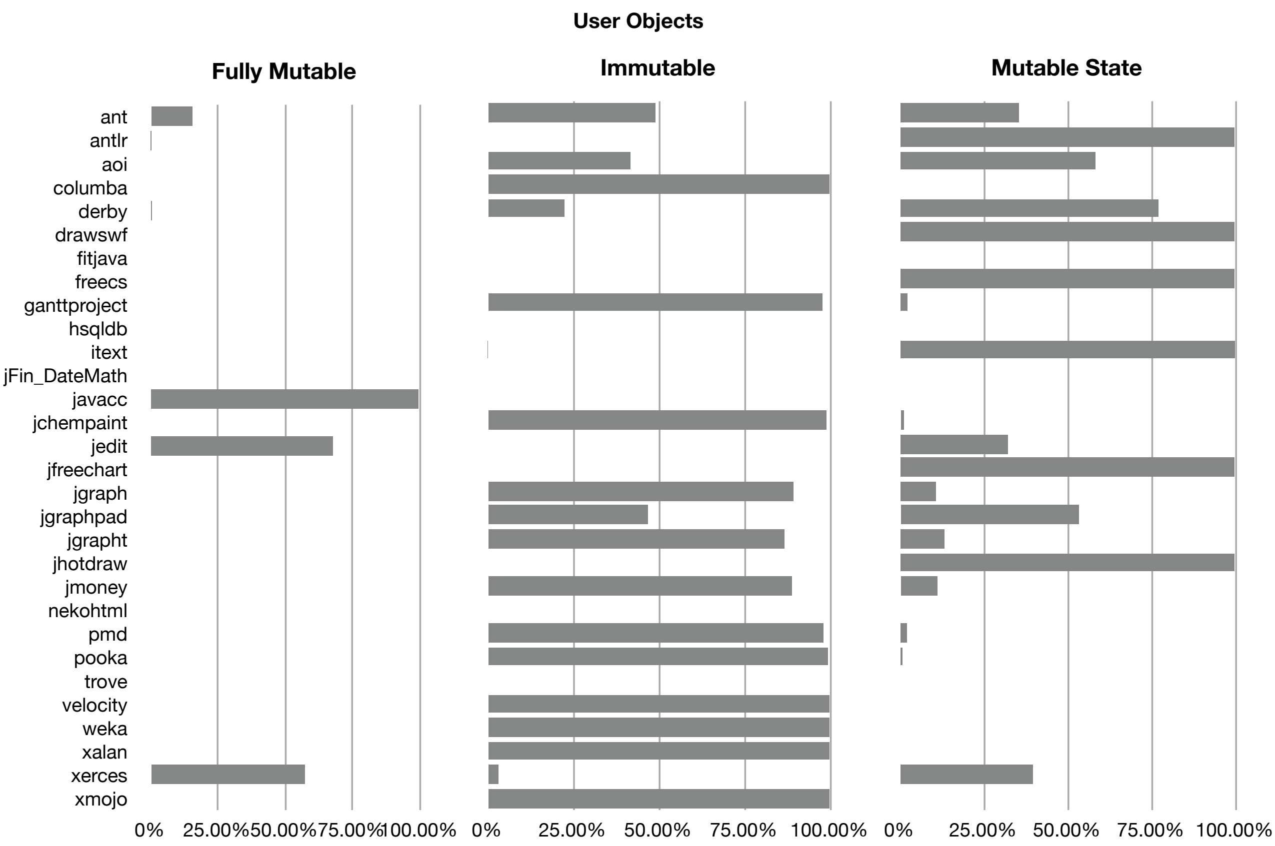


Objects in Equality Collections

User Objects

	Total	Fully Mutable	Immutable	Mutable State	Inverse	Fully Mutable	Immutable	Mutable State
ant	191394	29710	93702	67982	2577800	15.52%	48.96%	35.52%
antlr	750	2	0	748	48831	0.27%		99.73%
aoi	12	0	5	7	23711		41.67%	58.33%
columba	19	0	19	0	34383		100.00%	
derby	379	2	85	292	3859	0.53%	22.43%	77.04%
drawswf	109	0	0	109	5778			100.00%
fitjava	0	0	0	0	739			
freecs	2	0	0	2	3760			100.00%
ganttproject	530	0	518	12	36867		97.74%	2.26%
hsqldb	0	0	0	0	27014			
itext	1452	0	2	1450	14476		0.14%	99.86%
jFin_DateMath	0	0	0	0	881			
javacc	536	536	0	0	73799	100.00%		
jchempaint	1432	0	1416	16	859190		98.88%	1.12%
jedit	4252	2884	0	1368	101660	67.83%		32.17%
jfreechart	3	0	0	3	2521			100.00%
jgraph	65	0	58	7	74307		89.23%	10.77%
jgraphpad	1559	0	728	831	88907		46.70%	53.30%
jgrapht	30	0	26	4	18561		86.67%	13.33%
jhotdraw	1	0	0	1	48072			100.00%
jmoney	18	0	16	2	33496		88.89%	11.11%
nekohtml	0	0	0	0	9215			
pmd	61559	0	60296	1263	714710		97.95%	2.05%
pooka	838	0	832	6	58127		99.28%	0.72%
trove	0	0	0	0	2400310			
velocity	2	0	2	0	1300		100.00%	
weka	20	0	20	0	606321		100.00%	
xalan	47	0	47	0	3003		100.00%	
xerces	164	94	5	65	2175	57.32%	3.05%	39.63%
xmojo	188	0	188	0	1105		100.00%	
Sum	265361	33228	157965	74168	0	12.52%	59.53%	27.95%
Mean	8845.366666667	1107.6	5265.5	2472.266666667	262495.9333333	8.05%	44.05%	31.23%
Deviation	36246.61057589	5428.197018253	19989.18086116	12380.61658799	643241.3258565	23.62%	45.77%	40.31%

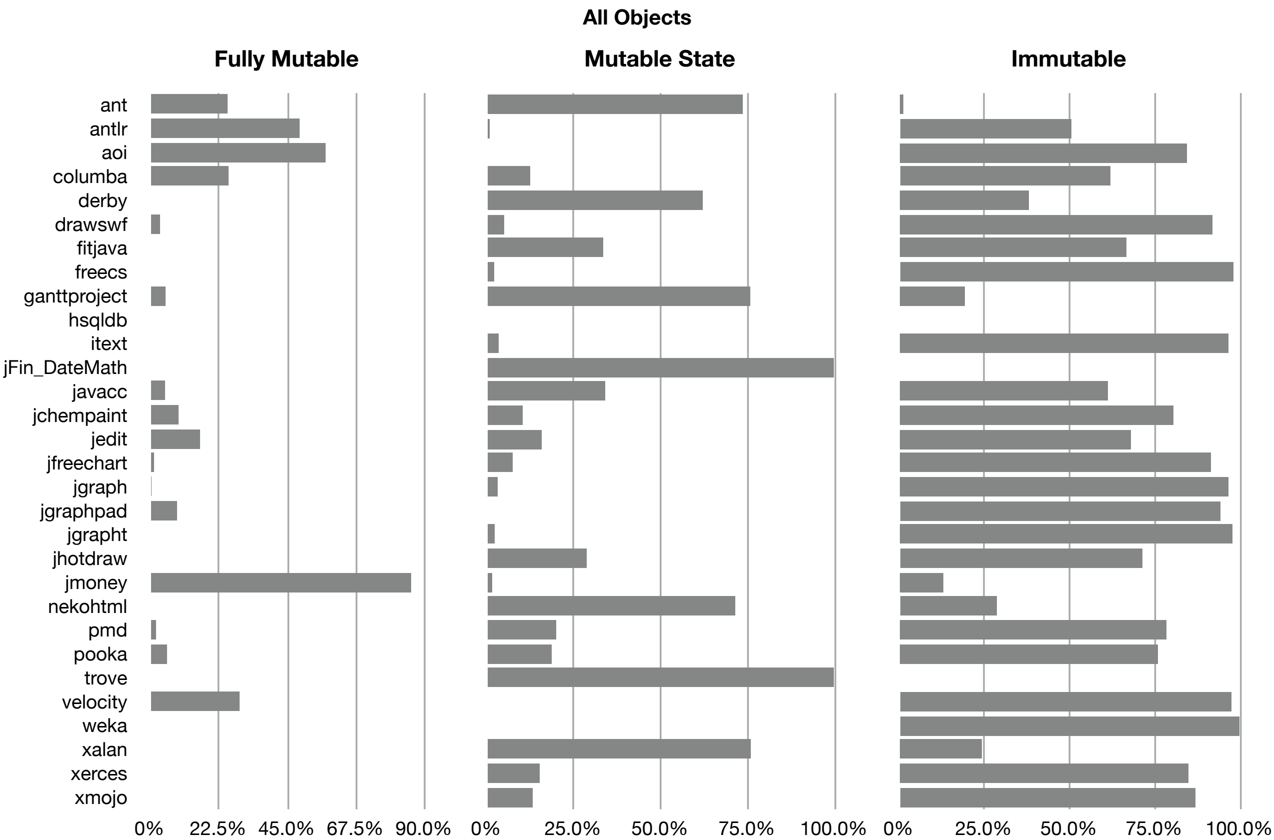
Objects in Equality Collections



Objects in non-Equality Collections

All Objects								
	Total	Fully Mutable	Immutable	Mutable State	Inverse	Fully Mutable	Immutable	Mutable State
ant	836075	210949	10202	614924	0	25.23%	1.22%	73.55%
antlr	1138	556	574	8	0	48.86%	50.44%	0.70%
aoi	1876	1079	1584	-787	0	57.52%	84.43%	-41.95%
columba	7291	1869	4513	909	0	25.63%	61.90%	12.47%
derby	408	0	155	253	0		37.99%	62.01%
drawswf	1937	61	1781	95	0	3.15%	91.95%	4.90%
fitjava	3	0	2	1	0		66.67%	33.33%
freecs	1980	0	1940	40	0		97.98%	2.02%
ganttproject	13016	656	2509	9851	0	5.04%	19.28%	75.68%
hsqldb	0	0	0	0	0			
itext	2583	0	2495	88	0		96.59%	3.41%
jFin_DateMath	131	0	0	131	0			100.00%
javacc	38743	1873	23699	13171	0	4.83%	61.17%	34.00%
jchempaint	26329	2429	21188	2712	0	9.23%	80.47%	10.30%
jedit	34999	5713	23787	5499	0	16.32%	67.96%	15.71%
jfreechart	338	4	309	25	0	1.18%	91.42%	7.40%
jgraph	59929	203	57894	1832	0	0.34%	96.60%	3.06%
jgraphpad	50122	4359	47196	-1433	0	8.70%	94.16%	-2.86%
jgrapht	9446	0	9239	207	0		97.81%	2.19%
jhotdraw	11718	0	8353	3365	0		71.28%	28.72%
jmoney	3079	2635	398	46	0	85.58%	12.93%	1.49%
nekohtml	7	0	2	5	0		28.57%	71.43%
pmd	333216	6018	261161	66037	0	1.81%	78.38%	19.82%
pooka	21449	1181	16270	3998	0	5.51%	75.85%	18.64%
trove	100000	0	0	100000	0			100.00%
velocity	154	45	150	-41	0	29.22%	97.40%	-26.62%
weka	28	0	28	0	0		100.00%	
xalan	806	0	195	611	0		24.19%	75.81%
xerces	311	0	264	47	0		84.89%	15.11%
xmojo	129	0	112	17	0		86.82%	13.18%
Sum	1557241	239630	496000	821611	0	15.39%	31.85%	52.76%
Mean	51908.03333333	7987.666666667	16533.33333333	27387.03333333	0	10.94%	61.95%	23.78%
Deviation	160782.8642957	38370.44552897	48326.16048258	113004.9922606	0	20.43%	35.12%	35.33%

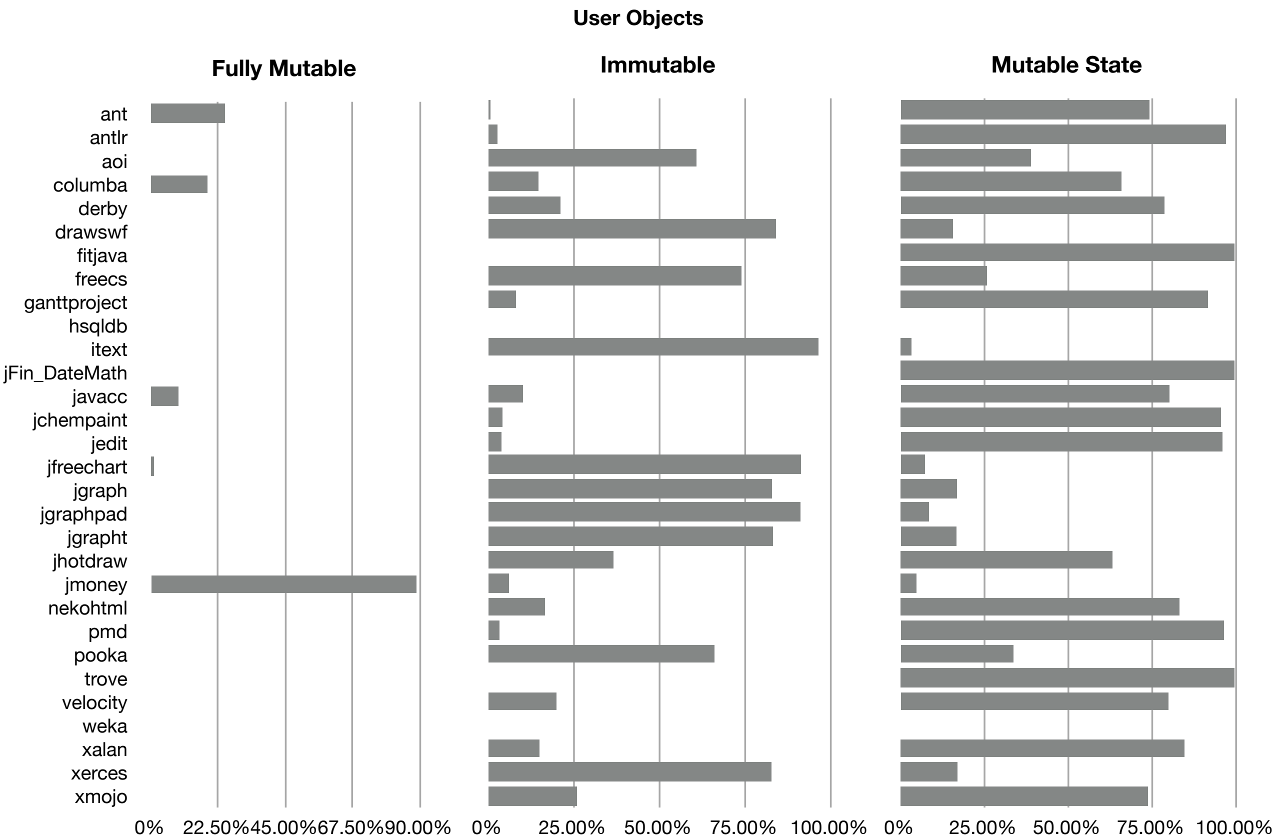
Objects in non-Equality Collections



Objects in non-Equality Collections

	User Objects							
	Total	Fully Mutable	Immutable	Mutable State	Inverse	Fully Mutable	Immutable	Mutable State
ant	832206	207198	6333	618675	1936988	24.90%	0.76%	74.34%
antlr	580	0	16	564	49001		2.76%	97.24%
aoi	748	0	456	292	22975		60.96%	39.04%
columba	3260	623	482	2155	31142	19.11%	14.79%	66.10%
derby	321	0	68	253	3917		21.18%	78.82%
drawswf	987	0	831	156	4900		84.19%	15.81%
fitjava	1	0	0	1	738			100.00%
freecs	154	0	114	40	3608		74.03%	25.97%
ganttproject	11443	0	936	10507	25954		8.18%	91.82%
hsqldb	0	0	0	0	27014			
itext	2525	0	2437	88	13403		96.51%	3.49%
jFin_DateMath	131	0	0	131	750			100.00%
javacc	16776	1575	1732	13469	57559	9.39%	10.32%	80.29%
jchempaint	4725	0	201	4524	855897		4.25%	95.75%
jedit	11669	0	457	11212	94243		3.92%	96.08%
jfreechart	338	4	309	25	2186	1.18%	91.42%	7.40%
jgraph	2979	0	2472	507	71393		82.98%	17.02%
jgraphpad	9886	0	9025	861	80580		91.29%	8.71%
jgrapht	1232	0	1025	207	17359		83.20%	16.80%
jhotdraw	273	0	100	173	47800		36.63%	63.37%
jmoney	2857	2541	176	140	30657	88.94%	6.16%	4.90%
nekohtml	6	0	1	5	9209		16.67%	83.33%
pmd	74626	0	2571	72055	701643		3.45%	96.55%
pooka	15327	0	10148	5179	43638		66.21%	33.79%
trove	100000	0	0	100000	2300310			100.00%
velocity	5	0	1	4	1297		20.00%	80.00%
weka	0	0	0	0	606341			
xalan	720	0	109	611	2330		15.14%	84.86%
xerces	274	0	227	47	2065		82.85%	17.15%
xmojo	23	0	6	17	1270		26.09%	73.91%
Sum	1094072	211941	40233	841898	0	19.37%	3.68%	76.95%
Mean	36469.06666667	7064.7	1341.1	28063.26666667	234872.2333333	4.78%	33.46%	55.08%
Deviation	151904.9105823	37803.04543119	2598.189905061	113683.7277365	557573.7038191	16.92%	36.01%	37.96%

Objects in non-Equality Collections

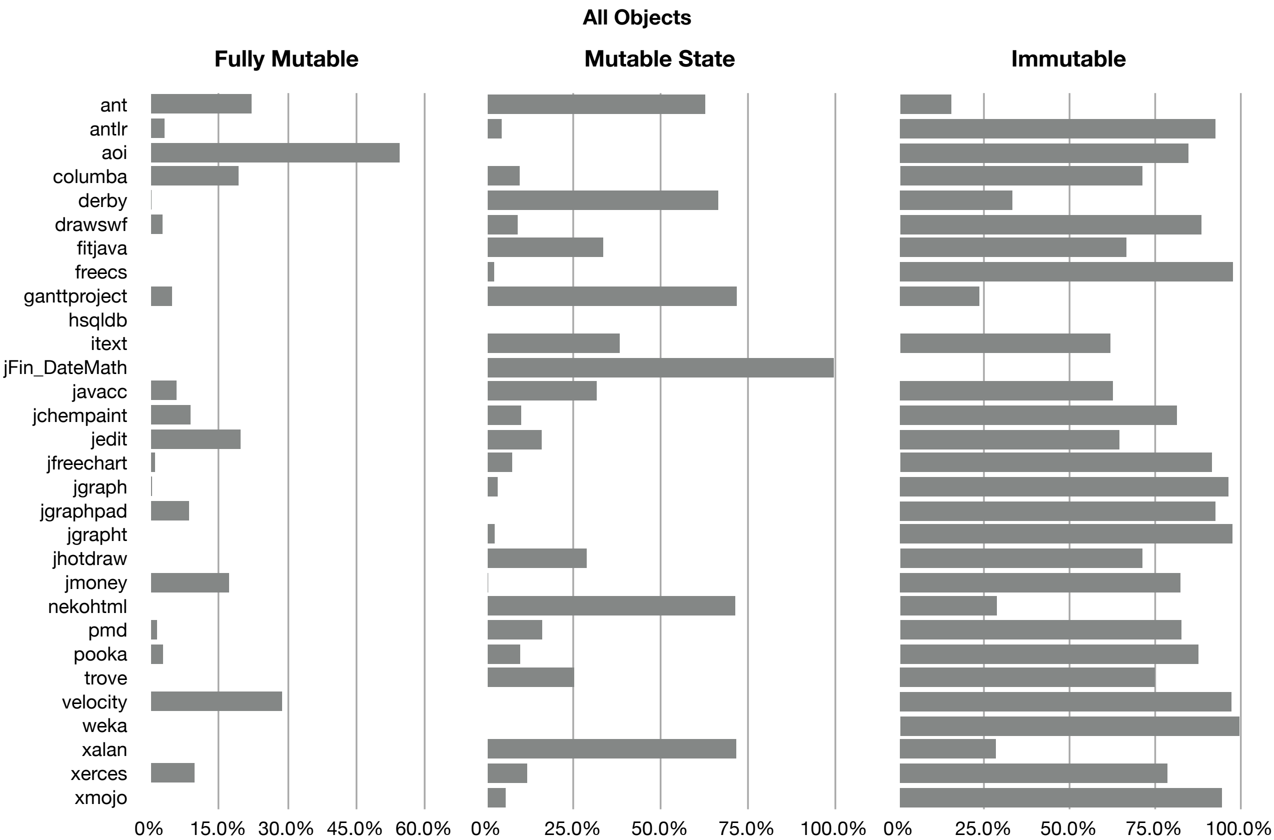


Objects in Collections

All Objects

	Total	Fully Mutable	Immutable	Mutable State	Inverse	Fully Mutable	Immutable	Mutable State
ant	1089185	240659	165620	682906	0	22.10%	15.21%	62.70%
antlr	18166	558	16852	756	0	3.07%	92.77%	4.16%
aoi	1980	1079	1681	-780	0	54.49%	84.90%	-39.39%
columba	9672	1869	6894	909	0	19.32%	71.28%	9.40%
derby	819	2	272	545	0	0.24%	33.21%	66.54%
drawswf	2326	61	2061	204	0	2.62%	88.61%	8.77%
fitjava	3	0	2	1	0		66.67%	33.33%
freecs	1982	0	1940	42	0		97.88%	2.12%
ganttproject	13746	656	3227	9863	0	4.77%	23.48%	71.75%
hsqldb	0	0	0	0	0			
itext	4035	0	2497	1538	0		61.88%	38.12%
jFin_DateMath	131	0	0	131	0			100.00%
javacc	41805	2409	26225	13171	0	5.76%	62.73%	31.51%
jchempaint	27761	2429	22604	2728	0	8.75%	81.42%	9.83%
jedit	43724	8597	28260	6867	0	19.66%	64.63%	15.71%
jfreechart	384	4	352	28	0	1.04%	91.67%	7.29%
jgraph	59994	203	57952	1839	0	0.34%	96.60%	3.07%
jgraphpad	51681	4359	47924	-602	0	8.43%	92.73%	-1.16%
jgrapht	9476	0	9265	211	0		97.77%	2.23%
jhotdraw	11719	0	8353	3366	0		71.28%	28.72%
jmoney	15289	2635	12606	48	0	17.23%	82.45%	0.31%
nekohtml	7	0	2	5	0		28.57%	71.43%
pmd	424464	6018	351146	67300	0	1.42%	82.73%	15.86%
pooka	42236	1181	37051	4004	0	2.80%	87.72%	9.48%
trove	400000	0	300000	100000	0		75.00%	25.00%
velocity	156	45	152	-41	0	28.85%	97.44%	-26.28%
weka	48	0	48	0	0		100.00%	
xalan	853	0	242	611	0		28.37%	71.63%
xerces	969	94	763	112	0	9.70%	78.74%	11.56%
xmojo	317	0	300	17	0		94.64%	5.36%
Sum	2272928	272858	1104291	895779	0	12.00%	48.58%	39.41%
Mean	75764.26666667	9095.266666667	36809.7	29859.3	0	7.02%	68.35%	21.30%
Deviation	217182.3236365	43782.24322945	85052.64047114	125190.0876899	0	12.03%	30.68%	31.34%

Objects in Collections

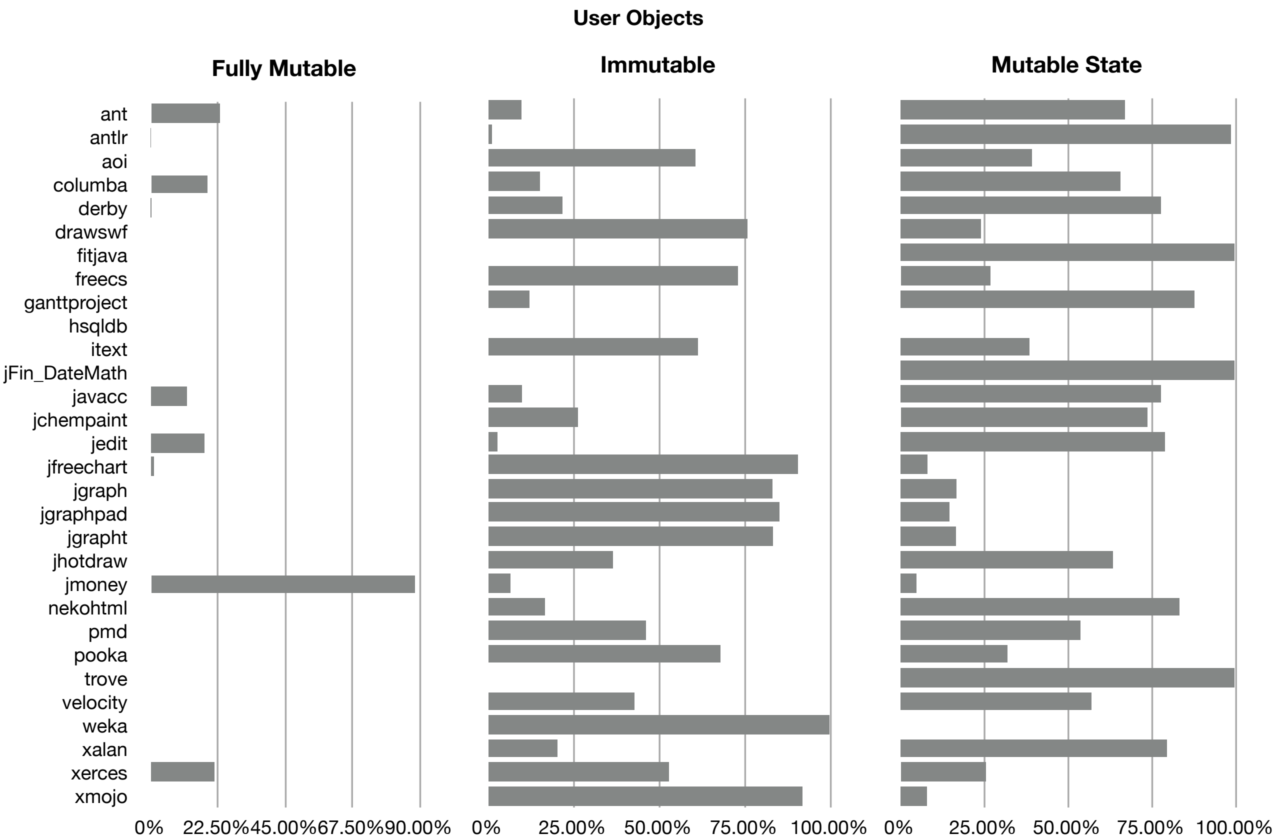


Objects in Collections

User Objects

	Total	Fully Mutable	Immutable	Mutable State	Inverse	Fully Mutable	Immutable	Mutable State
ant	1023600	236908	100035	686657	1745594	23.14%	9.77%	67.08%
antlr	1330	2	16	1312	48251	0.15%	1.20%	98.65%
aoi	760	0	461	299	22963		60.66%	39.34%
columba	3279	623	501	2155	31123	19.00%	15.28%	65.72%
derby	700	2	153	545	3538	0.29%	21.86%	77.86%
drawswf	1096	0	831	265	4791		75.82%	24.18%
fitjava	1	0	0	1	738			100.00%
freecs	156	0	114	42	3606		73.08%	26.92%
ganttproject	11973	0	1454	10519	25424		12.14%	87.86%
hsqldb	0	0	0	0	27014			
itext	3977	0	2439	1538	11951		61.33%	38.67%
jFin_DateMath	131	0	0	131	750			100.00%
javacc	17312	2111	1732	13469	57023	12.19%	10.00%	77.80%
jchempaint	6157	0	1617	4540	854465		26.26%	73.74%
jedit	15921	2884	457	12580	89991	18.11%	2.87%	79.02%
jfreechart	341	4	309	28	2183	1.17%	90.62%	8.21%
jgraph	3044	0	2530	514	71328		83.11%	16.89%
jgraphpad	11445	0	9753	1692	79021		85.22%	14.78%
jgrapht	1262	0	1051	211	17329		83.28%	16.72%
jhotdraw	274	0	100	174	47799		36.50%	63.50%
jmoney	2875	2541	192	142	30639	88.38%	6.68%	4.94%
nekohtml	6	0	1	5	9209		16.67%	83.33%
pmd	136185	0	62867	73318	640084		46.16%	53.84%
pooka	16165	0	10980	5185	42800		67.92%	32.08%
trove	100000	0	0	100000	2300310			100.00%
velocity	7	0	3	4	1295		42.86%	57.14%
weka	20	0	20	0	606321		100.00%	
xalan	767	0	156	611	2283		20.34%	79.66%
xerces	438	94	232	112	1901	21.46%	52.97%	25.57%
xmojo	211	0	194	17	1082		91.94%	8.06%
Sum	1359433	245169	198198	916066	0	18.03%	14.58%	67.39%
Mean	45314.43333333	8172.3	6606.6	30535.53333333	226026.8666667	6.13%	39.82%	50.72%
Deviation	187163.8510147	43208.17024125	21081.425401	125864.1073199	536490.9875962	17.14%	34.21%	33.91%

Objects in Collections

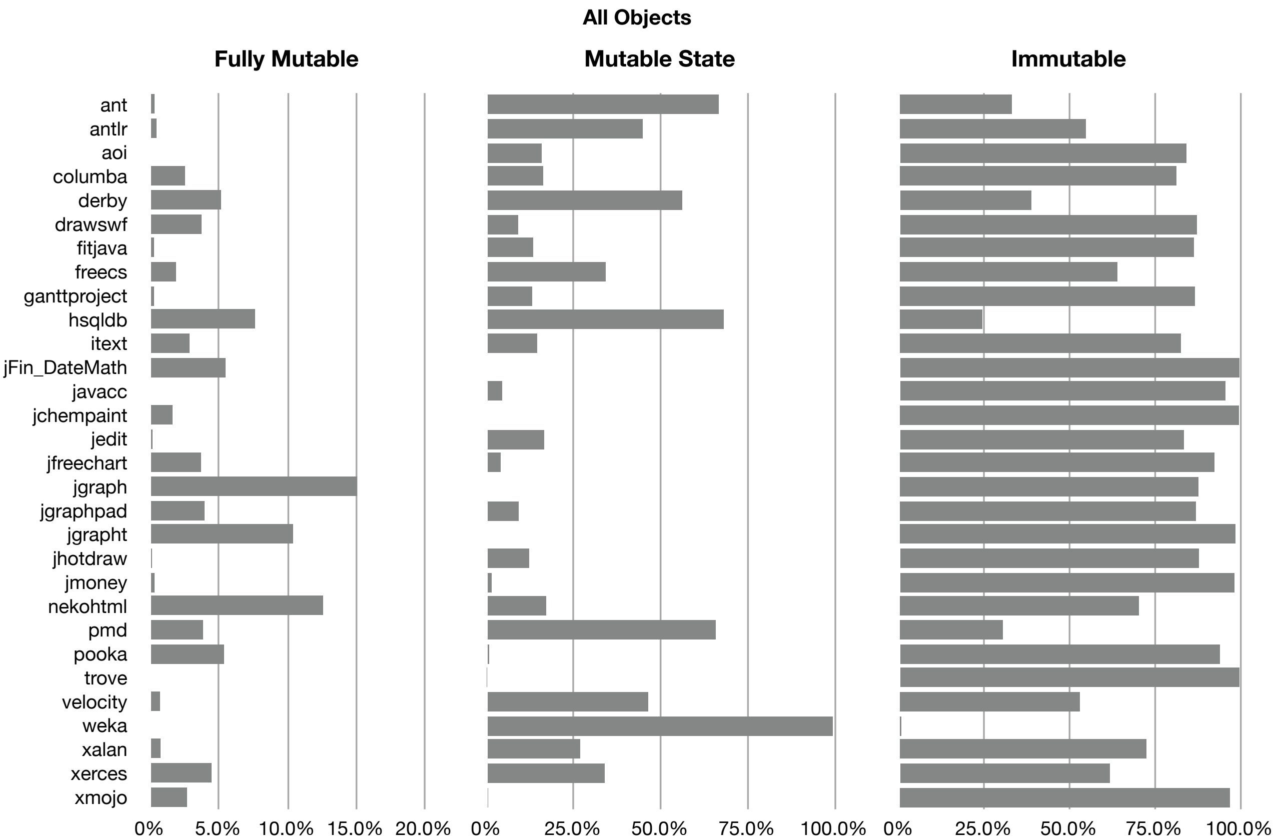


Objects in not in Collections

All Objects

	Total	Fully Mutable	Immutable	Mutable State	Inverse	Fully Mutable	Immutable	Mutable State
ant	1680009	5202	554587	1120220	0	0.31%	33.01%	66.68%
antlr	31415	136	17216	14063	0	0.43%	54.80%	44.77%
aoi	21743	5	18308	3430	0	0.02%	84.20%	15.78%
columba	24730	626	20124	3980	0	2.53%	81.37%	16.09%
derby	3419	176	1323	1920	0	5.15%	38.70%	56.16%
drawswf	3561	133	3108	320	0	3.73%	87.28%	8.99%
fitjava	736	2	636	98	0	0.27%	86.41%	13.32%
freecs	1780	33	1139	608	0	1.85%	63.99%	34.16%
ganttproject	23651	66	20512	3073	0	0.28%	86.73%	12.99%
hsqldb	27014	2064	6574	18376	0	7.64%	24.34%	68.02%
itext	11893	339	9835	1719	0	2.85%	82.70%	14.45%
jFin_DateMath	750	41	750	-41	0	5.47%	100.00%	-5.47%
javacc	32530	0	31109	1421	0		95.63%	4.37%
jchempaint	832861	13405	830580	-11124	0	1.61%	99.73%	-1.34%
jedit	62188	96	51908	10184	0	0.15%	83.47%	16.38%
jfreechart	2140	79	1978	83	0	3.69%	92.43%	3.88%
jgraph	14378	2164	12628	-414	0	15.05%	87.83%	-2.88%
jgraphpad	38785	1528	33750	3507	0	3.94%	87.02%	9.04%
jgrapht	9115	947	8990	-822	0	10.39%	98.63%	-9.02%
jhotdraw	36354	41	31934	4379	0	0.11%	87.84%	12.05%
jmoney	18225	57	17914	254	0	0.31%	98.29%	1.39%
nekohtml	9208	1160	6477	1571	0	12.60%	70.34%	17.06%
pmd	351805	13447	106871	231487	0	3.82%	30.38%	65.80%
pooka	16729	896	15727	106	0	5.36%	94.01%	0.63%
trove	2000310	0	2000215	95	0		100.00%	0.00%
velocity	1146	8	607	531	0	0.70%	52.97%	46.34%
weka	606293	15	3914	602364	0	0.00%	0.65%	99.35%
xalan	2197	16	1594	587	0	0.73%	72.55%	26.72%
xerces	1370	61	846	463	0	4.45%	61.75%	33.80%
xmojo	976	26	947	3	0	2.66%	97.03%	0.31%
Sum	5867311	42769	3812101	2012441	0	0.73%	64.97%	34.30%
Mean	195577.0333333	1425.633333333	127070.0333333	67081.36666667	0	3.20%	74.47%	22.33%
Deviation	486842.6249393	3431.764244006	395387.7946931	230229.9445001	0	3.87%	26.30%	26.74%

Objects in not in Collections



Objects in not in Collections

	User Objects							
	Total	Fully Mutable	Immutable	Mutable State	Inverse	Fully Mutable	Immutable	Mutable State
ant	1596183	6	470761	1125416	1173011	0.00%	29.49%	70.51%
antlr	14216	0	17	14199	35365		0.12%	99.88%
aoi	18170	2	16474	1694	5553	0.01%	90.67%	9.32%
columba	8020	597	3985	3438	26382	7.44%	49.69%	42.87%
derby	2660	126	564	1970	1578	4.74%	21.20%	74.06%
drawswf	777	0	324	453	5110		41.70%	58.30%
fitjava	101	0	1	100	638		0.99%	99.01%
freecs	657	0	16	641	3105		2.44%	97.56%
ganttproject	12580	0	9477	3103	24817		75.33%	24.67%
hsqldb	22930	2064	2490	18376	4084	9.00%	10.86%	80.14%
itext	2227	0	169	2058	13701		7.59%	92.41%
jFin_DateMath	20	0	20	0	861		100.00%	
javacc	1992	0	571	1421	72343		28.66%	71.34%
jchempaint	792853	0	790588	2265	67769		99.71%	0.29%
jedit	37728	0	28067	9661	68184		74.39%	25.61%
jfreechart	1327	8	1165	154	1197	0.60%	87.79%	11.61%
jgraph	738	0	458	280	73634		62.06%	37.94%
jgraphpad	3407	0	2565	842	87059		75.29%	24.71%
jgrapht	372	18	247	107	18219	4.84%	66.40%	28.76%
jhotdraw	6988	0	6441	547	41085		92.17%	7.83%
jmoney	5910	0	5738	172	27604		97.09%	2.91%
nekohtml	4362	1154	1631	1577	4853	26.46%	37.39%	36.15%
pmd	253308	3	8374	244931	522961	0.00%	3.31%	96.69%
pooka	7638	0	6638	1000	51327		86.91%	13.09%
trove	2000189	0	2000094	95	400121		100.00%	0.00%
velocity	607	0	68	539	695		11.20%	88.80%
weka	605473	0	3094	602379	868		0.51%	99.49%
xalan	869	1	266	602	2181	0.12%	30.61%	69.28%
xerces	1128	11	604	513	1211	0.98%	53.55%	45.48%
xmojo	88	0	60	28	1205		68.18%	31.82%
Sum	5403518	3990	3360967	2038561	0	0.07%	62.20%	37.73%
Mean	180117.2666667	133	112032.2333333	67952.03333333	91224.03333333	1.81%	50.18%	48.02%
Deviation	478415.7161729	432.8727614279	392685.9751906	231305.6539103	234568.495139	5.20%	35.95%	35.62%

Objects in not in Collections