Convert at your own risk: The Java Cast Expressions in the Wild*

Luis Mastrangelo
Università della Svizzera italiana
Lugano, Switzerland
luis.mastrangelo@usi.ch

Matthias Hauswirth
Università della Svizzera italiana
Lugano, Switzerland
matthias.hauswirth@usi.ch

Nathaniel Nystrom
Università della Svizzera italiana
Lugano, Switzerland
nathaniel.nystrom@usi.ch

ABSTRACT

In Java, type cast operators provide a way to fill the gap between compile time and runtime type safety. There is an increasing literature on how casting affects development productivity. This is done usually by doing empirical studies on development groups, which are given programming tasks they have to solve.

However, those programming tasks are usually artificial. And it is unclear whether or not they reflect the kind of code that it is actually written in the "real" world. To properly assess this kind of studies, it is needed to understand how the type cast operators are actually used.

Thus, we try to answer the question: How and why are casts being used in "real" Java code? This paper studies the casts operator in a large Java repository.

To study how are they used, and most importantly, why are they used, we have analyzed 88GB of compressed `.jar` files on a mainstream Java repository. We have discovered several *cast* patterns. We hope that our study gives support for more empirical studies to understand how a static type system impacts the development productivity.

CCS CONCEPTS

• **Software and its engineering** → **Patterns**; • **General and reference** → *Empirical studies*;

KEYWORDS

cast, pattern, mining, design, empirical, static analysis, bytecode, Java, Maven Central

ACM Reference format:
Luis Mastrangelo, Matthias Hauswirth, and Nathaniel Nystrom. 1997. Convert at your own risk: The Java Cast Expressions in the Wild. In *Proceedings of ACM Woodstock conference, El Paso, Texas USA, July 1997 (WOODSTOCK'97),* 5 pages.
https://doi.org/10.475/123_4

*Produces the permission block, and copyright information

WOODSTOCK'97, July 1997, El Paso, Texas USA

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

1 INTRODUCTION

In programming language design, the goal of a type system is to prevent certain kind of errors at runtime. Thus, a type system is formulated as a collections of constraints that gives any expression in the program a well defined type. Type systems can be characterized in many different ways. The most common being when it is either statically or dynamically checked.

In the context of object-oriented languages, there is usually a subtype mechanism that allows the interoperability of two different, but related types. In the particular case of Java (OO language with static type system), the cast expression[1] and the `instanceof` operator[2] provide a bridge between compile-time and runtime checking. This is due most to the subtyping mechanism found in most of these kind of languages.

But, there is a constant struggle between the advocates of these two categories. The ones for static type system claim that it help them to detect errors in advance. In the contrary, the ones for dynamic type system claim that the verbosity of a static system slows down the development progress; and any error detected by a static type system should be catched easily by a well defined test suite.

Unfortunately, there is no clear response to this dilemma. There are several studies that try to answer this question. Harlin et. al [7] test whether the use of a static type system improves development time. Stuchlik and Hanenberg [17] have done a empirical study about the relationship between type casts and development time. To properly assess these kind of studies, it is needed to understand what kind of casts are written, and more importantly, the rationale behind them.

Moreover, sometimes a cast indicates a design flaw in an object-oriented system. Can we detect when a cast is a sign of a flaw in an object-oriented design? Can we improve class design by studying the use of casts?

This paper tries to answer these questions. We have analyzed and studied a large Java repository looking for cast and related operators to see how and why are they used. We come up with cast patterns that provide the rationale behind them.

The rest of this paper is organized as follows. Section 2 presents an overview of casting in Java. Section 3 discusses our research questions and introduces our study. Section 4 presents an overview of how casts are used. Section 5 describes our methodology for finding casts usage patterns. Sections 6 and 7 introduce and discuss the patterns we found. Section 8 presents related work, and Section 9 concludes the paper.

[1] https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.16
[2] https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.20.2

2 CASTS

A *cast* in Java serves the purpose of convert two related types. As defined in the Java specification[3], there are several kinds of conversions. In this context we are interested in conversion of classes.

Listing 1 shows how the cast operator is used to retrieve an element from a collection. In this case, the target of the cast expression is the get method (line 1), which its return type is `Object`. Therefore, in order to use it properly, a cast is needed.

```
1 String s = (String)list.get();
2 System.out.println(s);
```

Listing 1: Simple cast

Whenever a cast fails at runtime, a `ClassCastException`[4] is thrown. Listing 2 shows an example where a `ClassCastException` is thrown at runtime. In this example the exception is thrown because it is not possible to conversion from `Integer` to `String`.

```
1 Object x = new Integer(0);
2 System.out.println((String)x);
```

Listing 2: Throwing ClassCastException

As with any exception, the `ClassCastException` can be catched to detect whenever a cast failed. This is shown in listing 3.

```
1 try {
2   Object x = new Integer(0);
3   System.out.println((String)x);
4 } catch (ClassCastException e) {
5   System.out.println("");
6 }
```

Listing 3: Catching ClassCastException

Sometimes it is not desired to catch an exception to test whether a cast would fail otherwise. Thus, in addition to the cast operator, the `instanceof` operator tests whether an expression can be casted properly. Listing 4 shows a usage of the `instanceof` operator together with a cast expression.

```
1 if (x instanceof Foo) {
2   ((Foo)x).doFoo();
3 }
```

Listing 4: Example of instanceof operator

An alternative to using the `instanceof` operator is keeping track of the types at the application level, as shown in listing 5. This kind of cast is called *semi guarded* casts [18].

```
1 if (x.isFoo()) {
2   ((Foo)x).doFoo();
3 }
```

Listing 5: Example of instanceof operator

Doing an *upcast* is trivial and does not require an explicit casting.

[3]https://docs.oracle.com/javase/specs/jls/se8/html/jls-5.html
[4]https://docs.oracle.com/javase/8/docs/api/java/lang/ClassCastException.html

3 STUDY OVERVIEW

We believe we should care about how the casting operations are used in the wild if we want to properly support empirical studies related to static type systems. Therefore, we want to answer the following questions:

- $Q1$: **Are casting operations used in common application code?** We want to understand to what extent third-party code actually uses casting operations.
- $Q2$: **Which features of are used?** As provides many features, we want to understand which ones are actually used, and which ones can be ignored.
- $Q3$: **Why are features used?** We want to investigate what functionality third-party libraries require from . This could point out ways in which the Java language and/or the JVM need to be evolved to provide the same functionality, but in a safer way.

To answer the above questions, we need to determine whether and how casting operations are actually used in real-world third-party Java libraries. To achieve our goal, several elements are needed.

Code Repository. As a code base representative of the "real world", we have chosen the Maven Central[5] software repository. The rationale behind this decision is that a large number of well-known Java projects deploy to Maven Central using Apache Maven[6]. Besides code written in Java, projects written in Scala are also deployed to Maven Central using the Scala Build Tool (sbt)[7]. Moreover, Maven Central is the largest Java repository[8], and it contains projects from the most popular source code management repositories, like GitHub[9] and SourceForge[10].

Artifacts. In Maven terminology, an artifact is the output of the build procedure of a project. An artifact can be any type of file, ranging from a `.pdf` to a `.zip` file. However, artifacts are usually `.jar` files, which archive compiled Java bytecode stored in `.class` files.

Bytecode Analysis. We examine these kinds of artifacts to analyze how they use casting operations. We use a bytecode analysis library to search for method call sites and field accesses of the *sun.misc.Unsafe* class.

Usage Pattern Detection. After all call sites and field accesses are found, we analyze this information to discover usage patterns. It is common that an artifact exhibits more than one pattern. Our list of patterns is not exhaustive. We have manually investigated the source code of the 100 highest-impact artifacts using *sun.misc.Unsafe* to understand why and how they are using it.

4 ARE THEY *CASTS* OPERATOR USED?

Some repo and casts stats.

```
--- Size ---
Total uncompressed size: 176,925 MB
--- Structural ---
```

[5]http://central.sonatype.org/
[6]http://maven.apache.org/
[7]http://www.scala-sbt.org/
[8]http://www.modulecounts.com/
[9]https://github.com/
[10]http://sourceforge.net/

Table 1: Patterns and their alternatives. indicates that there is a proposed alternative for Java.

Number of	Count
Classes	24,116,635
Methods	222,525,678

```
Number of classes: 24,116,635
Number of methods: 222,525,678
Number of call sites: 661,713,609
Number of field uses: 334,462,791
Number of constants: 133,020,244
--- Instructions ---
Number of zeroOpCount: 833,070,650
Number of iincCount: 12,052,811
Number of multiANewArrayCount: 70,688
Number of intOpCount: 98,592,545
Number of jumpCount: 223,854,453
Number of varCount: 1,227,756,300
Number of invokeDynamicCount: 1,481,910
Number of lookupSwitchCount: 1,044,018
Number of tableSwitchCount: 1,377,260
--- Casts ---
Number of CHECKCAST: 47,947,250
Number of INSTANCEOF: 8,505,668
Number of ClassCastException: 114,049
Methods w/ CHECKCAST: 27,033,672
Methods w/ INSTANCEOF: 5,270,791
--- Error ---
Files not found: 150
```

5 FINDING *CASTS* USAGE PATTERNS

We have analyzed 88GB of jar files under the Maven Central Repository. We have used the last version of each artifact in the Maven Repository. This a representative of the artifact itself.

Then we have used ASM [2]

The *Bytecode* column refer to either an cast related instruction or exception. These are the cast related bytecodes:

checkcast as specified by: [11]

instanceof as specified by: [12]

ClassCastException as specified by: [13]

The following two columns indicates how many bytecode where found in: - *local* My local machine. This machine contains a *partial* download of a current snapshot of Maven Central. Redownload all the artifacts is in progress. - *fermat* fermat.inf.usi.ch machine. This machine contains an old snapshot of Maven Central (2015)

[11] https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html#jvms-6.5.checkcast

[12] https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html#jvms-6.5.instanceof

[13] https://docs.oracle.com/javase/7/docs/api/java/lang/ClassCastException.html

6 CASTS USAGE PATTERNS

This section presents the patterns we have found during our study. We present them sorted by how many artifacts depend on them, as computed from the Maven dependency graph described in nowwhere.

A summary of the patterns is shown in Table 2. The ***Pattern*** column indicates the name of the pattern. ***Found in*** indicates the number of artifacts in Maven Central that contain the pattern. ***Used by*** indicates the number of artifacts that transitively depend on the artifacts with the pattern. ***Most used artifacts*** presents the three most used artifacts containing the pattern, that is, the artifact with the most other artifacts that transitively depend upon it. Artifacts are shown using their Maven identifier, i.e., ⟨*groupId*⟩:⟨*artifactId*⟩.

6.1 Casting from Const Null

Description. This pattern consists of a null expression casted to a specific reference type.
Rationale. This pattern is useful for creating mock objects for testing and in deserializing serialized objects.
Implementation. The *allocateInstance* method takes as parameter a *java.lang.Class* object, and returns a new instance of that class. Unlike allocating an object directly, or through the reflection API, the object's constructor is not invoked.
Issues. The null expression has any reference type. In principle, there is no need to make a explicit cast to null.

6.2 Equals casting

Description. Casting used in a equals method.
Rationale. This pattern is useful for creating mock objects for testing and in deserializing serialized objects.
Implementation. The *allocateInstance* method takes as parameter a *java.lang.Class* object, and returns a new instance of that class. Unlike allocating an object directly, or through the reflection API, the object's constructor is not invoked.
Issues. The null expression has any reference type. In principle, there is no need to make a explicit cast to null.

6.3 Casting after Clone

Description. Casting used in a equals method.
Rationale. This pattern is useful for creating mock objects for testing and in deserializing serialized objects.
Implementation. The *allocateInstance* method takes as parameter a *java.lang.Class* object, and returns a new instance of that class. Unlike allocating an object directly, or through the reflection API, the object's constructor is not invoked.
Issues. The null expression has any reference type. In principle, there is no need to make a explicit cast to null.

6.4 Collections element casting

Description. Casting after getting an element from a collection.
Rationale. This pattern is useful for creating mock objects for testing and in deserializing serialized objects.
Implementation. The *allocateInstance* method takes as parameter a *java.lang.Class* object, and returns a new instance of that class. Unlike allocating an object directly, or through the reflection API, the object's constructor is not invoked.

Table 2: Patterns and their occurrences in the Maven Central repository.

	Pattern	Found In	Used by	Most used artifacts
1	Casting from Const Null	88	14794	*org.springframework:spring-core*, *org.objenesis:objenesis*, *org.mockito:mockito-all*
2	Equals casting	88	14794	*org.springframework:spring-core*, *org.objenesis:objenesis*, *org.mockito:mockito-all*

Issues. The null expression has any reference type. In principle, there is no need to make a explicit cast to null.

6.5 Collections element casting

Description. Casting after getting an element from a generics.
Rationale. This pattern is useful for creating mock objects for testing and in deserializing serialized objects.
Implementation. The *allocateInstance* method takes as parameter a *java.lang.Class* object, and returns a new instance of that class. Unlike allocating an object directly, or through the reflection API, the object's constructor is not invoked.
Issues. The null expression has any reference type. In principle, there is no need to make a explicit cast to null.

7 DISCUSSION

Here we discuss about the results.

8 RELATED WORK

The manifest [10]. In the article, *In Defense of Soundiness: A Manifesto*, https://cacm.acm.org/magazines/2015/2/182650-in-defense-of-soundiness the authors claim that in order to understand how the limit of analysis tools impact software, first we need to understand the what kind of code is being written in the real world. This fact opens the door for empirical studies about language features and their use in software repositories, e.g., Maven Central.

- Guarded Type Promotion – Eliminating Redundant Casts in Java [18] Study of type casts in several project. Quite similar to what we want to do. Focus on Guarded Type casts.

- Contracts in the Wild: A Study of Java Programs [6] Investigate 25 fix contract patterns. Section 2.3: Come up with new Contract Patterns.

- Challenges for Static Analysis of Java Reflection – Literature Review and Empirical Study [8]. They also have done a study on Casts. But only for a small curated sets of projects. They analyze the relevance of static analysis tools w.r.t reflection. We want to study Reflection in the Wild. Empirical Studies on subjects need to be correlated with real world use cases, e.g. Maven Repository.

- Static vs. Dynamic Type Systems: An Empirical Study About the Relationship between Type Casts and Development Time [17] Studied the type casts in relation of development time. Group study. We want to Study Casts in the Wild.

- An empirical study of the influence of static type systems on the usability of undocumented software [12] Similar to Challenges ...

- Impact of Using a Static-Type System in Computer Programming [7] Test whether the use of a Static-Type System improves productivity. Productivity in this case is measured by development time. Two languages, a statically and dynamically-typed. Two programming tasks, Code a program from scratch and Debug a faulty program. Two program kinds, Simple program and Encryption program. A static-type system does not impact coding a program form scratch. Nevertheless, a static-type system does make software productivity improve when debugging a program.

- Empirical Study of Usage and Performance of Java Collections [4]. Mining GitHub corpus to study the use of collections, and how these usages can be improved.

Mining metapatterns in Java [15]
Adoption of Java Generics3 [14]

8.1 Exception Handling

Android [3]
How developer use exception handling in java [1]
Libraries java exception [16]
bdd [9]
java generics championed [13]
code smell [5]

8.2 Evidence Languages

Similar to our work related to **Unsafe** [11]

9 CONCLUSIONS

Here we conclude.

REFERENCES

[1] Muhammad Asaduzzaman, Muhammad Ahasanuzzaman, Chanchal K. Roy, and Kevin A. Schneider. 2016. How Developers Use Exception Handling in Java?. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 516–519. https://doi.org/10.1145/2901739.2903500

[2] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*.

[3] R. Coelho, L. Almeida, G. Gousios, and A. v. Deursen. 2015. Unveiling Exception Handling Bug Hazards in Android Based on GitHub and Google Code Issues. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 134–145. https://doi.org/10.1109/MSR.2015.20

[4] Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. 2017. Empirical Study of Usage and Performance of Java Collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17)*. ACM, New York, NY, USA, 389–400. https://doi.org/10.1145/3030207.3030221

[5] S. Counsell, R. M. Hierons, H. Hamza, S. Black, and M. Durrand. 2010. Is a Strategy for Code Smell Assessment Long Overdue?. In *Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics (WETSoM '10)*. ACM, New York, NY, USA, 32–38. https://doi.org/10.1145/1809223.1809228

[6] Jens Dietrich, David J. Pearce, Kamil Jezek, Premek Brada, and Jens Dietrich. 2017. Contracts in the Wild: A Study of Java Programs. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings*

in Informatics (LIPIcs)), Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 9:1–9:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.9

[7] I. R. Harlin, H. Washizaki, and Y. Fukazawa. 2017. Impact of Using a Static-Type System in Computer Programming. In *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*. 116–119. https://doi.org/10.1109/HASE.2017.17

[8] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for Static Analysis of Java Reflection: Literature Review and Empirical Study. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 507–518. https://doi.org/10.1109/ICSE.2017.53

[9] Ondřej Lhoták and Laurie Hendren. 2008. Evaluating the Benefits of Context-sensitive Points-to Analysis Using a BDD-based Implementation. *ACM Trans. Softw. Eng. Methodol.* 18, 1, Article 3 (Oct. 2008), 53 pages. https://doi.org/10.1145/1391984.1391987

[10] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. https://doi.org/10.1145/2644805

[11] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. 2015. Use at Your Own Risk: The Java Unsafe API in the Wild. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 695–710. https://doi.org/10.1145/2814270.2814313

[12] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. 2012. An Empirical Study of the Influence of Static Type Systems on the Usability of Undocumented Software. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA, 683–702. https://doi.org/10.1145/2384616.2384666

[13] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. 2011. Java Generics Adoption: How New Features Are Introduced, Championed, or Ignored. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11)*. ACM, New York, NY, USA, 3–12. https://doi.org/10.1145/1985441.1985446

[14] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. 2013. Adoption and Use of Java Generics. *Empirical Softw. Engg.* 18, 6 (Dec. 2013), 1047–1089. https://doi.org/10.1007/s10664-012-9236-6

[15] Daryl Posnett, Christian Bird, and Premkumar T. Devanbu. 2010. THEX: Mining metapatterns from java. In *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings*. 122–125. https://doi.org/10.1109/MSR.2010.5463349

[16] Demóstenes Sena, Roberta Coelho, Uirá Kulesza, and Rodrigo Bonifácio. 2016. Understanding the Exception Handling Strategies of Java Libraries: An Empirical Study. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 212–222. https://doi.org/10.1145/2901739.2901757

[17] Andreas Stuchlik and Stefan Hanenberg. 2011. Static vs. Dynamic Type Systems: An Empirical Study About the Relationship Between Type Casts and Development Time. In *Proceedings of the 7th Symposium on Dynamic Languages (DLS '11)*. ACM, New York, NY, USA, 97–106. https://doi.org/10.1145/2047849.2047861

[18] Johnni Winther. 2011. Guarded Type Promotion: Eliminating Redundant Casts in Java. In *Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs (FTfJP '11)*. ACM, New York, NY, USA, Article 6, 8 pages. https://doi.org/10.1145/2076674.2076680