Effective Java

Third Edition
To my family: Cindy, Tim, and Matt
This page intentionally left blank
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>Foreword</td>
<td>xi</td>
</tr>
<tr>
<td>Preface</td>
<td>Preface</td>
<td>xiii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>Acknowledgments</td>
<td>xvii</td>
</tr>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Creating and Destroying Objects</td>
<td>5</td>
</tr>
<tr>
<td>Item 1</td>
<td>Consider static factory methods instead of constructors</td>
<td>5</td>
</tr>
<tr>
<td>Item 2</td>
<td>Consider a builder when faced with many constructor parameters</td>
<td>10</td>
</tr>
<tr>
<td>Item 3</td>
<td>Enforce the singleton property with a private constructor or an enum type</td>
<td>17</td>
</tr>
<tr>
<td>Item 4</td>
<td>Enforce noninstantiability with a private constructor</td>
<td>19</td>
</tr>
<tr>
<td>Item 5</td>
<td>Prefer dependency injection to hardwiring resources</td>
<td>20</td>
</tr>
<tr>
<td>Item 6</td>
<td>Avoid creating unnecessary objects</td>
<td>22</td>
</tr>
<tr>
<td>Item 7</td>
<td>Eliminate obsolete object references</td>
<td>26</td>
</tr>
<tr>
<td>Item 8</td>
<td>Avoid finalizers and cleaners</td>
<td>29</td>
</tr>
<tr>
<td>Item 9</td>
<td>Prefer <code>try-with-resources</code> to <code>try-finally</code></td>
<td>34</td>
</tr>
<tr>
<td>3</td>
<td>Methods Common to All Objects</td>
<td>37</td>
</tr>
<tr>
<td>Item 10</td>
<td>Obey the general contract when overriding <code>equals</code></td>
<td>37</td>
</tr>
<tr>
<td>Item 11</td>
<td>Always override <code>hashCode</code> when you override <code>equals</code></td>
<td>50</td>
</tr>
<tr>
<td>Item 12</td>
<td>Always override <code>toString</code></td>
<td>55</td>
</tr>
<tr>
<td>Item 13</td>
<td>Override <code>clone</code> judiciously</td>
<td>58</td>
</tr>
<tr>
<td>Item 14</td>
<td>Consider implementing <code>Comparable</code></td>
<td>66</td>
</tr>
<tr>
<td>4</td>
<td>Classes and Interfaces</td>
<td>73</td>
</tr>
<tr>
<td>Item 15</td>
<td>Minimize the accessibility of classes and members</td>
<td>73</td>
</tr>
<tr>
<td>Item 16</td>
<td>In public classes, use accessor methods, not public fields</td>
<td>78</td>
</tr>
<tr>
<td>Item 17</td>
<td>Minimize mutability</td>
<td>80</td>
</tr>
<tr>
<td>Item 18</td>
<td>Favor composition over inheritance</td>
<td>87</td>
</tr>
</tbody>
</table>
Item 19: Design and document for inheritance or else prohibit it 93
Item 20: Prefer interfaces to abstract classes ....................... 99
Item 21: Design interfaces for posterity ......................... 104
Item 22: Use interfaces only to define types .................... 107
Item 23: Prefer class hierarchies to tagged classes .......... 109
Item 24: Favor static member classes over nonstatic .... 112
Item 25: Limit source files to a single top-level class .... 115

5 Generics .......................................................... 117
Item 26: Don’t use raw types ................................... 117
Item 27: Eliminate unchecked warnings ...................... 123
Item 28: Prefer lists to arrays .................................. 126
Item 29: Favor generic types ................................... 130
Item 30: Favor generic methods ................................ 135
Item 31: Use bounded wildcards to increase API flexibility ... 139
Item 32: Combine generics and varargs judiciously .......... 146
Item 33: Consider typesafe heterogeneous containers .... 151

6 Enums and Annotations ........................................ 157
Item 34: Use enums instead of int constants ............... 157
Item 35: Use instance fields instead of ordinals .......... 168
Item 36: Use EnumSet instead of bit fields ............... 169
Item 37: Use EnumMap instead of ordinal indexing .... 171
Item 38: Emulate extensible enums with interfaces .... 176
Item 39: Prefer annotations to naming patterns .......... 180
Item 40: Consistently use the Override annotation ...... 188
Item 41: Use marker interfaces to define types .......... 191

7 Lambdas and Streams ......................................... 193
Item 42: Prefer lambdas to anonymous classes .......... 193
Item 43: Prefer method references to lambdas .......... 197
Item 44: Favor the use of standard functional interfaces ... 199
Item 45: Use streams judiciously .......................... 203
Item 46: Prefer side-effect-free functions in streams ...... 210
Item 47: Prefer Collection to Stream as a return type ... 216
Item 48: Use caution when making streams parallel .. 222
8 Methods .......................................................... 227
  Item 49: Check parameters for validity ....................... 227
  Item 50: Make defensive copies when needed ............... 231
  Item 51: Design method signatures carefully ............ 236
  Item 52: Use overloading judiciously ..................... 238
  Item 53: Use varargs judiciously .......................... 245
  Item 54: Return empty collections or arrays, not nulls . 247
  Item 55: Return optionals judiciously .................... 249
  Item 56: Write doc comments for all exposed API elements . 254

9 General Programming ......................................... 261
  Item 57: Minimize the scope of local variables .......... 261
  Item 58: Prefer for-each loops to traditional for loops . 264
  Item 59: Know and use the libraries ...................... 267
  Item 60: Avoid float and double if exact answers are required . 270
  Item 61: Prefer primitive types to boxed primitives ...... 273
  Item 62: Avoid strings where other types are more appropriate . 276
  Item 63: Beware the performance of string concatenation . 279
  Item 64: Refer to objects by their interfaces .......... 280
  Item 65: Prefer interfaces to reflection .................. 282
  Item 66: Use native methods judiciously .................. 285
  Item 67: Optimize judiciously ............................ 286
  Item 68: Adhere to generally accepted naming conventions . 289

10 Exceptions ....................................................... 293
  Item 69: Use exceptions only for exceptional conditions ... 293
  Item 70: Use checked exceptions for recoverable conditions and runtime exceptions for programming errors . 296
  Item 71: Avoid unnecessary use of checked exceptions .... 298
  Item 72: Favor the use of standard exceptions .......... 300
  Item 73: Throw exceptions appropriate to the abstraction ... 302
  Item 74: Document all exceptions thrown by each method. 304
  Item 75: Include failure-capture information in detail messages. 306
  Item 76: Strive for failure atomicity ..................... 308
  Item 77: Don’t ignore exceptions ........................... 310
If a colleague were to say to you, “Spouse of me this night today manufactures the unusual meal in a home. You will join?” three things would likely cross your mind: third, that you had been invited to dinner; second, that English was not your colleague’s first language; and first, a good deal of puzzlement.

If you have ever studied a second language yourself and then tried to use it outside the classroom, you know that there are three things you must master: how the language is structured (grammar), how to name things you want to talk about (vocabulary), and the customary and effective ways to say everyday things (usage). Too often only the first two are covered in the classroom, and you find native speakers constantly suppressing their laughter as you try to make yourself understood.

It is much the same with a programming language. You need to understand the core language: is it algorithmic, functional, object-oriented? You need to know the vocabulary: what data structures, operations, and facilities are provided by the standard libraries? And you need to be familiar with the customary and effective ways to structure your code. Books about programming languages often cover only the first two, or discuss usage only spottily. Maybe that’s because the first two are in some ways easier to write about. Grammar and vocabulary are properties of the language alone, but usage is characteristic of a community that uses it.

The Java programming language, for example, is object-oriented with single inheritance and supports an imperative (statement-oriented) coding style within each method. The libraries address graphic display support, networking, distributed computing, and security. But how is the language best put to use in practice?

There is another point. Programs, unlike spoken sentences and unlike most books and magazines, are likely to be changed over time. It’s typically not enough to produce code that operates effectively and is readily understood by other persons; one must also organize the code so that it is easy to modify. There may be ten ways to write code for some task \( T \). Of those ten ways, seven will be awkward, inefficient, or puzzling. Of the other three, which is most likely to be similar to the code needed for the task \( T' \) in next year’s software release?
There are numerous books from which you can learn the grammar of the Java programming language, including *The Java™ Programming Language* by Arnold, Gosling, and Holmes, or *The Java™ Language Specification* by Gosling, Joy, yours truly, and Bracha. Likewise, there are dozens of books on the libraries and APIs associated with the Java programming language.

This book addresses your third need: customary and effective usage. Joshua Bloch has spent years extending, implementing, and using the Java programming language at Sun Microsystems; he has also read a lot of other people’s code, including mine. Here he offers good advice, systematically organized, on how to structure your code so that it works well, so that other people can understand it, so that future modifications and improvements are less likely to cause headaches—perhaps, even, so that your programs will be pleasant, elegant, and graceful.

Guy L. Steele Jr.
*Burlington, Massachusetts*
*April 2001*
Preface to the Third Edition

In 1997, when Java was new, James Gosling (the father of Java), described it as a “blue collar language” that was “pretty simple” [Gosling97]. At about the same time, Bjarne Stroustrup (the father of C++) described C++ as a “multi-paradigm language” that “deliberately differs from languages designed to support a single way of writing programs” [Stroustrup95]. Stroustrup warned:

Much of the relative simplicity of Java is—like for most new languages—partly an illusion and partly a function of its incompleteness. As time passes, Java will grow significantly in size and complexity. It will double or triple in size and grow implementation-dependent extensions or libraries. [Stroustrup]

Now, twenty years later, it’s fair to say that Gosling and Stroustrup were both right. Java is now large and complex, with multiple abstractions for many things, from parallel execution, to iteration, to the representation of dates and times.

I still like Java, though my ardor has cooled a bit as the platform has grown. Given its increased size and complexity, the need for an up-to-date best-practices guide is all the more critical. With this third edition of Effective Java, I did my best to provide you with one. I hope this edition continues to satisfy the need, while staying true to the spirit of the first two editions.

Small is beautiful, but simple ain’t easy.

San Jose, California
November 2017

P.S. I would be remiss if I failed to mention an industry-wide best practice that has occupied a fair amount of my time lately. Since the birth of our field in the 1950’s, we have freely reimplemented each others’ APIs. This practice was critical to the meteoric success of computer technology. I am active in the effort to preserve this freedom [CompSci17], and I encourage you to join me. It is crucial to the continued health of our profession that we retain the right to reimplement each others’ APIs.
Preface to the Second Edition

A lot has happened to the Java platform since I wrote the first edition of this book in 2001, and it’s high time for a second edition. The most significant set of changes was the addition of generics, enum types, annotations, autoboxing, and the for-each loop in Java 5. A close second was the addition of the new concurrency library, java.util.concurrent, also released in Java 5. With Gilad Bracha, I had the good fortune to lead the teams that designed the new language features. I also had the good fortune to serve on the team that designed and developed the concurrency library, which was led by Doug Lea.

The other big change in the platform is the widespread adoption of modern Integrated Development Environments (IDEs), such as Eclipse, IntelliJ IDEA, and NetBeans, and of static analysis tools, such as FindBugs. While I have not been involved in these efforts, I’ve benefited from them immensely and learned how they affect the Java development experience.

In 2004, I moved from Sun to Google, but I’ve continued my involvement in the development of the Java platform over the past four years, contributing to the concurrency and collections APIs through the good offices of Google and the Java Community Process. I’ve also had the pleasure of using the Java platform to develop libraries for use within Google. Now I know what it feels like to be a user.

As was the case in 2001 when I wrote the first edition, my primary goal is to share my experience with you so that you can imitate my successes while avoiding my failures. The new material continues to make liberal use of real-world examples from the Java platform libraries.

The first edition succeeded beyond my wildest expectations, and I’ve done my best to stay true to its spirit while covering all of the new material that was required to bring the book up to date. It was inevitable that the book would grow, and grow it did, from fifty-seven items to seventy-eight. Not only did I add twenty-three items, but I thoroughly revised all the original material and retired a few items whose better days had passed. In the Appendix, you can see how the material in this edition relates to the material in the first edition.

In the Preface to the First Edition, I wrote that the Java programming language and its libraries were immensely conducive to quality and productivity, and a joy to work with. The changes in releases 5 and 6 have taken a good thing and made it better. The platform is much bigger now than it was in 2001 and more complex, but once you learn the patterns and idioms for using the new features, they make your programs better and your life easier. I hope this edition captures my conti-
ued enthusiasm for the platform and helps make your use of the platform and its new features more effective and enjoyable.

San Jose, California
April 2008

Preface to the First Edition

In 1996 I pulled up stakes and headed west to work for JavaSoft, as it was then known, because it was clear that that was where the action was. In the intervening five years I’ve served as Java platform libraries architect. I’ve designed, implemented, and maintained many of the libraries and served as a consultant for many others. Presiding over these libraries as the Java platform matured was a once-in-a-lifetime opportunity. It is no exaggeration to say that I had the privilege to work with some of the great software engineers of our generation. In the process, I learned a lot about the Java programming language—what works, what doesn’t, and how to use the language and its libraries to best effect.

This book is my attempt to share my experience with you so that you can imitate my successes while avoiding my failures. I borrowed the format from Scott Meyers’s Effective C++, which consists of fifty items, each conveying one specific rule for improving your programs and designs. I found the format to be singularly effective, and I hope you do too.

In many cases, I took the liberty of illustrating the items with real-world examples from the Java platform libraries. When describing something that could have been done better, I tried to pick on code that I wrote myself, but occasionally I pick on something written by a colleague. I sincerely apologize if, despite my best efforts, I’ve offended anyone. Negative examples are cited not to cast blame but in the spirit of cooperation, so that all of us can benefit from the experience of those who’ve gone before.

While this book is not targeted solely at developers of reusable components, it is inevitably colored by my experience writing such components over the past two decades. I naturally think in terms of exported APIs (Application Programming Interfaces), and I encourage you to do likewise. Even if you aren’t developing reusable components, thinking in these terms tends to improve the quality of the software you write. Furthermore, it’s not uncommon to write a reusable compo-
nent without knowing it: You write something useful, share it with your buddy across the hall, and before long you have half a dozen users. At this point, you no longer have the flexibility to change the API at will and are thankful for all the effort that you put into designing the API when you first wrote the software.

My focus on API design may seem a bit unnatural to devotees of the new lightweight software development methodologies, such as *Extreme Programming*. These methodologies emphasize writing the simplest program that could possibly work. If you’re using one of these methodologies, you’ll find that a focus on API design serves you well in the *refactoring* process. The fundamental goals of refactoring are the improvement of system structure and the avoidance of code duplication. These goals are impossible to achieve in the absence of well-designed APIs for the components of the system.

No language is perfect, but some are excellent. I have found the Java programming language and its libraries to be immensely conducive to quality and productivity, and a joy to work with. I hope this book captures my enthusiasm and helps make your use of the language more effective and enjoyable.

*Cupertino, California*  
April 2001
Acknowledgments

Acknowledgments for the Third Edition

I thank the readers of the first two editions of this book for giving it such a kind and enthusiastic reception, for taking its ideas to heart, and for letting me know what a positive influence it had on them and their work. I thank the many professors who used the book in their courses, and the many engineering teams that adopted it.

I thank the whole team at Addison-Wesley and Pearson for their kindness, professionalism, patience, and grace under extreme pressure. Through it all, my editor Greg Doench remained unflappable: a fine editor and a perfect gentleman. I’m afraid his hair may have turned a bit gray as a result of this project, and I humbly apologize. My project manager, Julie Nahil, and my project editor, Dana Wilson, were all I could hope for: diligent, prompt, organized, and friendly. My copy editor, Kim Wimpsett, was meticulous and tasteful.

I have yet again been blessed with the best team of reviewers imaginable, and I give my sincerest thanks to each of them. The core team, who reviewed most every chapter, consisted of Cindy Bloch, Brian Kernighan, Kevin Bourrillion, Joe Bowbeer, William Chargin, Joe Darcy, Brian Goetz, Tim Halloran, Stuart Marks, Tim Peierls, and Yoshiki Shibata. Other reviewers included Marcus Biel, Dan Bloch, Beth Bottos, Martin Buchholz, Michael Diamond, Charlie Garrod, Tom Hawtin, Doug Lea, Aleksey Shipilëv, Lou Wasserman, and Peter Weinberger. These reviewers made numerous suggestions that led to great improvements in this book and saved me from many embarrassments.

I give special thanks to William Chargin, Doug Lea, and Tim Peierls, who served as sounding boards for many of the ideas in this book. William, Doug, and Tim were unfailingly generous with their time and knowledge.

Finally, I thank my wife, Cindy Bloch, for encouraging me to write, for reading each item in raw form, for writing the index, for helping me with all of the things that invariably come up when you take on a big project, and for putting up with me while I wrote.
Acknowledgments for the Second Edition

I thank the readers of the first edition of this book for giving it such a kind and enthusiastic reception, for taking its ideas to heart, and for letting me know what a positive influence it had on them and their work. I thank the many professors who used the book in their courses, and the many engineering teams that adopted it.

I thank the whole team at Addison-Wesley for their kindness, professionalism, patience, and grace under pressure. Through it all, my editor Greg Doench remained unflappable: a fine editor and a perfect gentleman. My production manager, Julie Nahil, was everything that a production manager should be: diligent, prompt, organized, and friendly. My copy editor, Barbara Wood, was meticulous and tasteful.

I have once again been blessed with the best team of reviewers imaginable, and I give my sincerest thanks to each of them. The core team, who reviewed every chapter, consisted of Lexi Baugher, Cindy Bloch, Beth Bottos, Joe Bowbeer, Brian Goetz, Tim Halloran, Brian Kernighan, Rob Konigsberg, Tim Peierls, Bill Pugh, Yoshiki Shibata, Peter Stout, Peter Weinberger, and Frank Yellin. Other reviewers included Pablo Bellver, Dan Bloch, Dan Bornstein, Kevin Bourrillion, Martin Buchholz, Joe Darcy, Neal Gafter, Laurence Gonsalves, Aaron Greenhouse, Barry Hayes, Peter Jones, Angelika Langer, Doug Lea, Bob Lee, Jeremy Manson, Tom May, Mike McCloskey, Andriy Tereshchenko, and Paul Tyma. Again, these reviewers made numerous suggestions that led to great improvements in this book and saved me from many embarrassments. And again, any remaining embarrassments are my responsibility.

I give special thanks to Doug Lea and Tim Peierls, who served as sounding boards for many of the ideas in this book. Doug and Tim were unfailingly generous with their time and knowledge.

I thank my manager at Google, Prabha Krishna, for her continued support and encouragement.

Finally, I thank my wife, Cindy Bloch, for encouraging me to write, for reading each item in raw form, for helping me with Framemaker, for writing the index, and for putting up with me while I wrote.
Acknowledgments for the First Edition

I thank Patrick Chan for suggesting that I write this book and for pitching the idea to Lisa Friendly, the series managing editor; Tim Lindholm, the series technical editor; and Mike Hendrickson, executive editor of Addison-Wesley. I thank Lisa, Tim, and Mike for encouraging me to pursue the project and for their superhuman patience and unyielding faith that I would someday write this book.

I thank James Gosling and his original team for giving me something great to write about, and I thank the many Java platform engineers who followed in James’s footsteps. In particular, I thank my colleagues in Sun’s Java Platform Tools and Libraries Group for their insights, their encouragement, and their support. The team consists of Andrew Bennett, Joe Darcy, Neal Gafter, Iris Garcia, Konstantin Kladko, Ian Little, Mike McCloskey, and Mark Reinhold. Former members include Zhenghua Li, Bill Maddox, and Naveen Sanjeeva.

I thank my manager, Andrew Bennett, and my director, Larry Abrahams, for lending their full and enthusiastic support to this project. I thank Rich Green, the VP of Engineering at Java Software, for providing an environment where engineers are free to think creatively and to publish their work.

I have been blessed with the best team of reviewers imaginable, and I give my sincerest thanks to each of them: Andrew Bennett, Cindy Bloch, Dan Bloch, Beth Bottos, Joe Bowbeer, Gilad Bracha, Mary Campione, Joe Darcy, David Eckhardt, Joe Fialli, Lisa Friendly, James Gosling, Peter Haggar, David Holmes, Brian Kernighan, Konstantin Kladko, Doug Lea, Zhenghua Li, Tim Lindholm, Mike McCloskey, Tim Peierls, Mark Reinhold, Ken Russell, Bill Shannon, Peter Stout, Phil Wadler, and two anonymous reviewers. They made numerous suggestions that led to great improvements in this book and saved me from many embarrassments. Any remaining embarrassments are my responsibility.

Numerous colleagues, inside and outside Sun, participated in technical discussions that improved the quality of this book. Among others, Ben Gomes, Steffen Grarup, Peter Kessler, Richard Roda, John Rose, and David Stoutamire contributed useful insights. A special thanks is due Doug Lea, who served as a sounding board for many of the ideas in this book. Doug has been unfailingly generous with his time and his knowledge.

I thank Julie Dinicola, Jacqui Doucette, Mike Hendrickson, Heather Olszyk, Tracy Russ, and the whole team at Addison-Wesley for their support and professionalism. Even under an impossibly tight schedule, they were always friendly and accommodating.
I thank Guy Steele for writing the Foreword. I am honored that he chose to participate in this project.

Finally, I thank my wife, Cindy Bloch, for encouraging and occasionally threatening me to write this book, for reading each item in its raw form, for helping me with Framemaker, for writing the index, and for putting up with me while I wrote.
CHAPTER  5

Generics

Since Java 5, generics have been a part of the language. Before generics, you had to cast every object you read from a collection. If someone accidentally inserted an object of the wrong type, casts could fail at runtime. With generics, you tell the compiler what types of objects are permitted in each collection. The compiler inserts casts for you automatically and tells you at compile time if you try to insert an object of the wrong type. This results in programs that are both safer and clearer, but these benefits, which are not limited to collections, come at a price. This chapter tells you how to maximize the benefits and minimize the complications.

Item 26: Don’t use raw types

First, a few terms. A class or interface whose declaration has one or more type parameters is a generic class or interface [JLS, 8.1.2, 9.1.2]. For example, the List interface has a single type parameter, E, representing its element type. The full name of the interface is List<E> (read “list of E”), but people often call it List for short. Generic classes and interfaces are collectively known as generic types.

Each generic type defines a set of parameterized types, which consist of the class or interface name followed by an angle-bracketed list of actual type parameters corresponding to the generic type’s formal type parameters [JLS, 4.4, 4.5]. For example, List<String> (read “list of string”) is a parameterized type representing a list whose elements are of type String. (String is the actual type parameter corresponding to the formal type parameter E.)

Finally, each generic type defines a raw type, which is the name of the generic type used without any accompanying type parameters [JLS, 4.8]. For example, the raw type corresponding to List<E> is List. Raw types behave as if all of the generic type information were erased from the type declaration. They exist primarily for compatibility with pre-generics code.
Before generics were added to Java, this would have been an exemplary collection declaration. As of Java 9, it is still legal, but far from exemplary:

```java
// Raw collection type - don't do this!
// My stamp collection. Contains only Stamp instances.
private final Collection stamps = ... ;
```

If you use this declaration today and then accidentally put a coin into your stamp collection, the erroneous insertion compiles and runs without error (though the compiler does emit a vague warning):

```java
// Erroneous insertion of coin into stamp collection
stamps.add(new Coin( ... )); // Emits "unchecked call" warning
```

You don’t get an error until you try to retrieve the coin from the stamp collection:

```java
// Raw iterator type - don't do this!
for (Iterator i = stamps.iterator(); i.hasNext(); )
   Stamp stamp = (Stamp) i.next(); // Throws ClassCastException
   stamp.cancel();
```

As mentioned throughout this book, it pays to discover errors as soon as possible after they are made, ideally at compile time. In this case, you don’t discover the error until runtime, long after it has happened, and in code that may be distant from the code containing the error. Once you see the ClassCastException, you have to search through the codebase looking for the method invocation that put the coin into the stamp collection. The compiler can’t help you, because it can’t understand the comment that says, “Contains only Stamp instances.”

With generics, the type declaration contains the information, not the comment:

```java
// Parameterized collection type - typesafe
private final Collection<Stamp> stamps = ... ;
```

From this declaration, the compiler knows that `stamps` should contain only `Stamp` instances and guarantees it to be true, assuming your entire codebase compiles without emitting (or suppressing; see Item 27) any warnings. When `stamps` is declared with a parameterized type declaration, the erroneous insertion generates a compile-time error message that tells you exactly what is wrong:

```java
Test.java:9: error: incompatible types: Coin cannot be converted to Stamp
   c.add(new Coin());
   ^
```
The compiler inserts invisible casts for you when retrieving elements from collections and guarantees that they won’t fail (assuming, again, that all of your code did not generate or suppress any compiler warnings). While the prospect of accidentally inserting a coin into a stamp collection may appear far-fetched, the problem is real. For example, it is easy to imagine putting a BigInteger into a collection that is supposed to contain only BigDecimal instances.

As noted earlier, it is legal to use raw types (generic types without their type parameters), but you should never do it. **If you use raw types, you lose all the safety and expressiveness benefits of generics.** Given that you shouldn’t use them, why did the language designers permit raw types in the first place? For compatibility. Java was about to enter its second decade when generics were added, and there was an enormous amount of code in existence that did not use generics. It was deemed critical that all of this code remain legal and interoperate with newer code that does use generics. It had to be legal to pass instances of parameterized types to methods that were designed for use with raw types, and vice versa. This requirement, known as migration compatibility, drove the decisions to support raw types and to implement generics using erasure (Item 28).

While you shouldn’t use raw types such as List, it is fine to use types that are parameterized to allow insertion of arbitrary objects, such as List<Object>. Just what is the difference between the raw type List and the parameterized type List<Object>? Loosely speaking, the former has opted out of the generic type system, while the latter has explicitly told the compiler that it is capable of holding objects of any type. While you can pass a List<String> to a parameter of type List, you can’t pass it to a parameter of type List<Object>. There are subtyping rules for generics, and List<String> is a subtype of the raw type List, but not of the parameterized type List<Object> (Item 28). As a consequence, you lose type safety if you use a raw type such as List, but not if you use a parameterized type such as List<Object>.

To make this concrete, consider the following program:

```java
// Fails at runtime - unsafeAdd method uses a raw type (List)!
public static void main(String[] args) {
    List<String> strings = new ArrayList<>();
    unsafeAdd(strings, Integer.valueOf(42));
    String s = strings.get(0); // Has compiler-generated cast
}

private static void unsafeAdd(List list, Object o) {
    list.add(o);
}
```
This program compiles, but because it uses the raw type `List`, you get a warning:

```
Test.java:10: warning: [unchecked] unchecked call to add(E) as a member of the raw type List
    list.add(o);
   ^
```

And indeed, if you run the program, you get a `ClassCastException` when the program tries to cast the result of the invocation `strings.get(0)`, which is an `Integer`, to a `String`. This is a compiler-generated cast, so it's normally guaranteed to succeed, but in this case we ignored a compiler warning and paid the price.

If you replace the raw type `List` with the parameterized type `List<Object>` in the `unsafeAdd` declaration and try to recompile the program, you'll find that it no longer compiles but emits the error message:

```
Test.java:5: error: incompatible types: List<String> cannot be converted to List<Object>
    unsafeAdd(strings, Integer.valueOf(42));
   ^
```

You might be tempted to use a raw type for a collection whose element type is unknown and doesn't matter. For example, suppose you want to write a method that takes two sets and returns the number of elements they have in common. Here's how you might write such a method if you were new to generics:

```
// Use of raw type for unknown element type - don't do this!
static int numElementsInCommon(Set s1, Set s2) {
    int result = 0;
    for (Object o1 : s1)
        if (s2.contains(o1))
            result++;
    return result;
}
```

This method works but it uses raw types, which are dangerous. The safe alternative is to use *unbounded wildcard types*. If you want to use a generic type but you don't know or care what the actual type parameter is, you can use a question mark instead. For example, the unbounded wildcard type for the generic type `Set<E>` is `Set<?>` (read “set of some type”). It is the most general parameterized `Set` type, capable of holding any set. Here is how the `numElementsInCommon` declaration looks with unbounded wildcard types:

```
// Uses unbounded wildcard type - typesafe and flexible
static int numElementsInCommon(Set<?> s1, Set<?> s2) { ... }
```
What is the difference between the unbounded wildcard type `Set<?>` and the raw type `Set`? Does the question mark really buy you anything? Not to belabor the point, but the wildcard type is safe and the raw type isn’t. You can put any element into a collection with a raw type, easily corrupting the collection’s type invariant (as demonstrated by the `unsafeAdd` method on page 119): **you can’t put any element (other than `null`) into a `Collection<?>`**. Attempting to do so will generate a compile-time error message like this:

```
WildCard.java:13: error: incompatible types: String cannot be converted to CAP#1
c.add("verboten");
  ^
where CAP#1 is a fresh type-variable:
  CAP#1 extends Object from capture of ?
```

Admittedly this error message leaves something to be desired, but the compiler has done its job, preventing you from corrupting the collection’s type invariant, whatever its element type may be. Not only can’t you put any element (other than `null`) into a `Collection<?>` , but you can’t assume anything about the type of the objects that you get out. If these restrictions are unacceptable, you can use **generic methods** (Item 30) or **bounded wildcard types** (Item 31).

There are a few minor exceptions to the rule that you should not use raw types. **You must use raw types in class literals.** The specification does not permit the use of parameterized types (though it does permit array types and primitive types) [JLS, 15.8.2]. In other words, `List.class`, `String[].class`, and `int.class` are all legal, but `List<String>.class` and `List<?>.class` are not.

A second exception to the rule concerns the `instanceof` operator. Because generic type information is erased at runtime, it is illegal to use the `instanceof` operator on parameterized types other than unbounded wildcard types. The use of unbounded wildcard types in place of raw types does not affect the behavior of the `instanceof` operator in any way. In this case, the angle brackets and question marks are just noise. **This is the preferred way to use the `instanceof` operator with generic types:**

```
// Legitimate use of raw type - instanceof operator
if (o instanceof Set) { // Raw type
    Set<?> s = (Set<?>) o; // Wildcard type
    ...
}
```
Note that once you’ve determined that o is a Set, you must cast it to the wildcard type Set<?>, not the raw type Set. This is a checked cast, so it will not cause a compiler warning.

In summary, using raw types can lead to exceptions at runtime, so don’t use them. They are provided only for compatibility and interoperability with legacy code that predates the introduction of generics. As a quick review, Set<Object> is a parameterized type representing a set that can contain objects of any type, Set<?> is a wildcard type representing a set that can contain only objects of some unknown type, and Set is a raw type, which opts out of the generic type system. The first two are safe, and the last is not.

For quick reference, the terms introduced in this item (and a few introduced later in this chapter) are summarized in the following table:

<table>
<thead>
<tr>
<th>Term</th>
<th>Example</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameterized type</td>
<td>List&lt;String&gt;</td>
<td>Item 26</td>
</tr>
<tr>
<td>Actual type parameter</td>
<td>String</td>
<td>Item 26</td>
</tr>
<tr>
<td>Generic type</td>
<td>List&lt;E&gt;</td>
<td>Items 26, 29</td>
</tr>
<tr>
<td>Formal type parameter</td>
<td>E</td>
<td>Item 26</td>
</tr>
<tr>
<td>Unbounded wildcard type</td>
<td>List&lt;?&gt;</td>
<td>Item 26</td>
</tr>
<tr>
<td>Raw type</td>
<td>List</td>
<td>Item 26</td>
</tr>
<tr>
<td>Bounded type parameter</td>
<td>&lt;E extends Number&gt;</td>
<td>Item 29</td>
</tr>
<tr>
<td>Recursive type bound</td>
<td>&lt;&lt;T extends Comparable&lt;T&gt;&gt;</td>
<td>Item 30</td>
</tr>
<tr>
<td>Bounded wildcard type</td>
<td>List&lt;? extends Number&gt;</td>
<td>Item 31</td>
</tr>
<tr>
<td>Generic method</td>
<td>static &lt;E&gt; List&lt;E&gt; asList(E[] a)</td>
<td>Item 30</td>
</tr>
<tr>
<td>Type token</td>
<td>String.class</td>
<td>Item 33</td>
</tr>
</tbody>
</table>
Item 27: Eliminate unchecked warnings

When you program with generics, you will see many compiler warnings: unchecked cast warnings, unchecked method invocation warnings, unchecked parameterized vararg type warnings, and unchecked conversion warnings. The more experience you acquire with generics, the fewer warnings you’ll get, but don’t expect newly written code to compile cleanly.

Many unchecked warnings are easy to eliminate. For example, suppose you accidentally write this declaration:

Set<Lark> exaltation = new HashSet();

The compiler will gently remind you what you did wrong:

Venery.java:4: warning: [unchecked] unchecked conversion
  Set<Lark> exaltation = new HashSet();   ^
  required: Set<Lark>
  found:    HashSet

You can then make the indicated correction, causing the warning to disappear. Note that you don’t actually have to specify the type parameter, merely to indicate that it’s present with the *diamond operator* (`<>`), introduced in Java 7. The compiler will then *infer* the correct actual type parameter (in this case, Lark):

Set<Lark> exaltation = new HashSet<>();

Some warnings will be *much* more difficult to eliminate. This chapter is filled with examples of such warnings. When you get warnings that require some thought, persevere! **Eliminate every unchecked warning that you can.** If you eliminate all warnings, you are assured that your code is typesafe, which is a very good thing. It means that you won’t get a ClassCastException at runtime, and it increases your confidence that your program will behave as you intended.

If you can’t eliminate a warning, but you can prove that the code that provoked the warning is typesafe, then (and only then) suppress the warning with an @SuppressWarnings("unchecked") annotation. If you suppress warnings without first proving that the code is typesafe, you are giving yourself a false sense of security. The code may compile without emitting any warnings, but it can still throw a ClassCastException at runtime. If, however, you ignore unchecked warnings that you know to be safe (instead of suppressing them), you won’t notice when a new warning crops up that represents a real problem. The new warning will get lost amidst all the false alarms that you didn’t silence.
The SuppressWarnings annotation can be used on any declaration, from an individual local variable declaration to an entire class. Always use the SuppressWarnings annotation on the smallest scope possible. Typically this will be a variable declaration or a very short method or constructor. Never use SuppressWarnings on an entire class. Doing so could mask critical warnings.

If you find yourself using the SuppressWarnings annotation on a method or constructor that’s more than one line long, you may be able to move it onto a local variable declaration. You may have to declare a new local variable, but it’s worth it. For example, consider this toArray method, which comes from ArrayList:

```java
public <T> T[] toArray(T[] a) {
    if (a.length < size) {
        return (T[]) Arrays.copyOf(elements, size, a.getClass());
        System.arraycopy(elements, 0, a, 0, size);
    } else if (a.length > size) {
        a[size] = null;
    }
    return a;
}
```

If you compile ArrayList, the method generates this warning:

```
ArrayList.java:305: warning: [unchecked] unchecked cast
    return (T[]) Arrays.copyOf(elements, size, a.getClass());
                           ^
required: T[]
found:   Object[]
```

It is illegal to put a SuppressWarnings annotation on the return statement, because it isn’t a declaration [JLS, 9.7]. You might be tempted to put the annotation on the entire method, but don’t. Instead, declare a local variable to hold the return value and annotate its declaration, like so:

```java
// Adding local variable to reduce scope of @SuppressWarnings
public <T> T[] toArray(T[] a) {
    if (a.length < size) {
        // This cast is correct because the array we're creating
        // is of the same type as the one passed in, which is T[].
        @SuppressWarnings("unchecked") T[] result =
            (T[]) Arrays.copyOf(elements, size, a.getClass());
        return result;
    }
    System.arraycopy(elements, 0, a, 0, size);
    if (a.length > size) {
        a[size] = null;
    }
    return a;
}
```
The resulting method compiles cleanly and minimizes the scope in which unchecked warnings are suppressed.

\textbf{Every time you use a \texttt{@SuppressWarnings("unchecked")} annotation, add a comment saying why it is safe to do so.} This will help others understand the code, and more importantly, it will decrease the odds that someone will modify the code so as to make the computation unsafe. If you find it hard to write such a comment, keep thinking. You may end up figuring out that the unchecked operation isn’t safe after all.

In summary, unchecked warnings are important. Don’t ignore them. Every unchecked warning represents the potential for a \texttt{ClassCastException} at runtime. Do your best to eliminate these warnings. If you can’t eliminate an unchecked warning and you can prove that the code that provoked it is typesafe, suppress the warning with a \texttt{@SuppressWarnings("unchecked")} annotation in the narrowest possible scope. Record the rationale for your decision to suppress the warning in a comment.
Item 28: Prefer lists to arrays

Arrays differ from generic types in two important ways. First, arrays are *covariant*. This scary-sounding word means simply that if Sub is a subtype of Super, then the array type Sub[] is a subtype of the array type Super[]. Generics, by contrast, are *invariant*: for any two distinct types Type1 and Type2, List<Type1> is neither a subtype nor a supertype of List<Type2> [JLS, 4.10; Naftalin07, 2.5]. You might think this means that generics are deficient, but arguably it is arrays that are deficient. This code fragment is legal:

```java
// Fails at runtime!
Object[] objectArray = new Long[1];
objectArray[0] = "I don't fit in"; // Throws ArrayStoreException
```

but this one is not:

```java
// Won't compile!
List<Object> ol = new ArrayList<Long>(); // Incompatible types
ol.add("I don't fit in");
```

Either way you can't put a *String* into a *Long* container, but with an array you find out that you've made a mistake at runtime; with a list, you find out at compile time. Of course, you'd rather find out at compile time.

The second major difference between arrays and generics is that arrays are *reified* [JLS, 4.7]. This means that arrays know and enforce their element type at runtime. As noted earlier, if you try to put a *String* into an array of *Long*, you'll get an *ArrayStoreException*. Generics, by contrast, are implemented by *erasure* [JLS, 4.6]. This means that they enforce their type constraints only at compile time and discard (or *erase*) their element type information at runtime. Erasure is what allowed generic types to interoperate freely with legacy code that didn't use generics (Item 26), ensuring a smooth transition to generics in Java 5.

Because of these fundamental differences, arrays and generics do not mix well. For example, it is illegal to create an array of a generic type, a parameterized type, or a type parameter. Therefore, none of these array creation expressions are legal: `new List<E>[]`, `new List<String>[]`, `new E[]`. All will result in *generic array creation* errors at compile time.

Why is it illegal to create a generic array? Because it isn't typesafe. If it were legal, casts generated by the compiler in an otherwise correct program could fail at runtime with a *ClassCastException*. This would violate the fundamental guarantee provided by the generic type system.
ITEM 28: PREFER LISTS TO ARRAYS

To make this more concrete, consider the following code fragment:

```java
// Why generic array creation is illegal - won't compile!
List<String>[] stringLists = new List<String>[1];  // (1)
List<Integer> intList = List.of(42);         // (2)
Object[] objects = stringLists;            // (3)
objects[0] = intList;                       // (4)
String s = stringLists[0].get(0);          // (5)
```

Let's pretend that line 1, which creates a generic array, is legal. Line 2 creates and initializes a `List<Integer>` containing a single element. Line 3 stores the `List<String>` array into an `Object` array variable, which is legal because arrays are covariant. Line 4 stores the `List<Integer>` into the sole element of the `Object` array, which succeeds because generics are implemented by erasure: the runtime type of a `List<Integer>` instance is simply `List`, and the runtime type of a `List<String>[]` instance is `List[]`, so this assignment doesn’t generate an `ArrayStoreException`. Now we’re in trouble. We’ve stored a `List<Integer>` instance into an array that is declared to hold only `List<String>` instances. In line 5, we retrieve the sole element from the sole list in this array. The compiler automatically casts the retrieved element to `String`, but it’s an `Integer`, so we get a `ClassCastException` at runtime. In order to prevent this from happening, line 1 (which creates a generic array) must generate a compile-time error.

Types such as `E`, `List<E>`, and `List<String>` are technically known as \textit{non-reifiable} types [JLS, 4.7]. Intuitively speaking, a non-reifiable type is one whose runtime representation contains less information than its compile-time representation. Because of erasure, the only parameterized types that are reifiable are unbounded wildcard types such as `List<?>` and `Map<?,?>` (Item 26). It is legal, though rarely useful, to create arrays of unbounded wildcard types.

The prohibition on generic array creation can be annoying. It means, for example, that it’s not generally possible for a generic collection to return an array of its element type (but see Item 33 for a partial solution). It also means that you get confusing warnings when using varargs methods (Item 53) in combination with generic types. This is because every time you invoke a varargs method, an array is created to hold the varargs parameters. If the element type of this array is not reifiable, you get a warning. The `SafeVarargs` annotation can be used to address this issue (Item 32).

When you get a generic array creation error or an unchecked cast warning on a cast to an array type, the best solution is often to use the collection type `List<E>` in preference to the array type `E[]`. You might sacrifice some conciseness or performance, but in exchange you get better type safety and interoperability.
For example, suppose you want to write a `Chooser` class with a constructor that takes a collection, and a single method that returns an element of the collection chosen at random. Depending on what collection you pass to the constructor, you could use a chooser as a game die, a magic 8-ball, or a data source for a Monte Carlo simulation. Here’s a simplistic implementation without generics:

```java
// Chooser - a class badly in need of generics!
public class Chooser {
    private final Object[] choiceArray;
    public Chooser(Collection choices) {
        choiceArray = choices.toArray();
    }
    public Object choose() {
        Random rnd = ThreadLocalRandom.current();
        return choiceArray[rnd.nextInt(choiceArray.length)];
    }
}
```

To use this class, you have to cast the `choose` method’s return value from `Object` to the desired type every time you use invoke the method, and the cast will fail at runtime if you get the type wrong. Taking the advice of Item 29 to heart, we attempt to modify `Chooser` to make it generic. Changes are shown in boldface:

```java
// A first cut at making Chooser generic - won't compile
public class Chooser<T> {
    private final T[] choiceArray;
    public Chooser(Collection<T> choices) {
        choiceArray = choices.toArray();
    }
    // choose method unchanged
}
```

If you try to compile this class, you’ll get this error message:

```
Chooser.java:9: error: incompatible types: Object[] cannot be converted to T[]
    choiceArray = choices.toArray();
          ^
where T is a type-variable:
    T extends Object declared in class Chooser
```

No big deal, you say, I’ll cast the Object array to a T array:

```java
choiceArray = (T[]) choices.toArray();
```

This gets rid of the error, but instead you get a warning:

```
Chooser.java:9: warning: [unchecked] unchecked cast
choiceArray = (T[]) choices.toArray();
```

The compiler is telling you that it can’t vouch for the safety of the cast at runtime because the program won’t know what type T represents—remember, element type information is erased from generics at runtime. Will the program work? Yes, but the compiler can’t prove it. You could prove it to yourself, put the proof in a comment and suppress the warning with an annotation, but you’re better off eliminating the cause of warning (Item 27).

To eliminate the unchecked cast warning, use a list instead of an array. Here is a version of the Chooser class that compiles without error or warning:

```java
// List-based Chooser - typesafe
public class Chooser<T> {
    private final List<T> choiceList;

    public Chooser(Collection<T> choices) {
        choiceList = new ArrayList<>(choices);
    }

    public T choose() {
        Random rnd = ThreadLocalRandom.current();
        return choiceList.get(rnd.nextInt(choiceList.size()));
    }
}
```

This version is a tad more verbose, and perhaps a tad slower, but it's worth it for the peace of mind that you won’t get a ClassCastException at runtime.

In summary, arrays and generics have very different type rules. Arrays are covariant and reified; generics are invariant and erased. As a consequence, arrays provide runtime type safety but not compile-time type safety, and vice versa for generics. As a rule, arrays and generics don’t mix well. If you find yourself mixing them and getting compile-time errors or warnings, your first impulse should be to replace the arrays with lists.
**Item 29: Favor generic types**

It is generally not too difficult to parameterize your declarations and make use of the generic types and methods provided by the JDK. Writing your own generic types is a bit more difficult, but it’s worth the effort to learn how.

Consider the simple (toy) stack implementation from Item 7:

```java
// Object-based collection - a prime candidate for generics
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        Object result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference
        return result;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

This class should have been parameterized to begin with, but since it wasn’t, we can generify it after the fact. In other words, we can parameterize it without harming clients of the original non-parameterized version. As it stands, the client has to cast objects that are popped off the stack, and those casts might fail at runtime.

The first step in generifying a class is to add one or more type parameters to its
declaration. In this case there is one type parameter, representing the element type of the stack, and the conventional name for this type parameter is $E$ (Item 68).

The next step is to replace all the uses of the type $Object$ with the appropriate type parameter and then try to compile the resulting program:

```java
// Initial attempt to generify Stack - won't compile!
public class Stack<E> {
    private E[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new E[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(E e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public E pop() {
        if (size == 0)
            throw new EmptyStackException();
        E result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference
        return result;
    }

    ... // no changes in isEmpty or ensureCapacity
}
```

You'll generally get at least one error or warning, and this class is no exception. Luckily, this class generates only one error:

```
Stack.java:8: generic array creation
    elements = new E[DEFAULT_INITIAL_CAPACITY];
∧
```

As explained in Item 28, you can’t create an array of a non-reifiable type, such as $E$. This problem arises every time you write a generic type that is backed by an array. There are two reasonable ways to solve it. The first solution directly circumvents the prohibition on generic array creation: create an array of $Object$ and cast
it to the generic array type. Now in place of an error, the compiler will emit a warning. This usage is legal, but it's not (in general) typesafe:

```java
Stack.java:8: warning: [unchecked] unchecked cast
    elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
        ^
```

The compiler may not be able to prove that your program is typesafe, but you can. You must convince yourself that the unchecked cast will not compromise the type safety of the program. The array in question (elements) is stored in a private field and never returned to the client or passed to any other method. The only elements stored in the array are those passed to the `push` method, which are of type `E`, so the unchecked cast can do no harm.

Once you've proved that an unchecked cast is safe, suppress the warning in as narrow a scope as possible (Item 27). In this case, the constructor contains only the unchecked array creation, so it's appropriate to suppress the warning in the entire constructor. With the addition of an annotation to do this, `Stack` compiles cleanly, and you can use it without explicit casts or fear of a `ClassCastException`:

```java
// The elements array will contain only E instances from push(E).
// This is sufficient to ensure type safety, but the runtime
// type of the array won't be E[]; it will always be Object[]!
@SuppressWarnings("unchecked")
public Stack() {
    elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
}
```

The second way to eliminate the generic array creation error in `Stack` is to change the type of the field `elements` from `E[]` to `Object[]`. If you do this, you'll get a different error:

```java
Stack.java:19: incompatible types
    E result = elements[--size];
        ^
```

You can change this error into a warning by casting the element retrieved from the array to `E`, but you will get a warning:

```java
Stack.java:19: warning: [unchecked] unchecked cast
    E result = (E) elements[--size];
        ^
```
Because E is a non-reifiable type, there's no way the compiler can check the cast at runtime. Again, you can easily prove to yourself that the unchecked cast is safe, so it's appropriate to suppress the warning. In line with the advice of Item 27, we suppress the warning only on the assignment that contains the unchecked cast, not on the entire pop method:

```java
// Appropriate suppression of unchecked warning
public E pop() {
    if (size == 0)
        throw new EmptyStackException();

    // push requires elements to be of type E, so cast is correct
    @SuppressWarnings("unchecked") E result =
        (E) elements[--size];

    elements[size] = null; // Eliminate obsolete reference
    return result;
}
```

Both techniques for eliminating the generic array creation have their adherents. The first is more readable: the array is declared to be of type E[], clearly indicating that it contains only E instances. It is also more concise: in a typical generic class, you read from the array at many points in the code; the first technique requires only a single cast (where the array is created), while the second requires a separate cast each time an array element is read. Thus, the first technique is preferable and more commonly used in practice. It does, however, cause heap pollution (Item 32): the runtime type of the array does not match its compile-time type (unless E happens to be Object). This makes some programmers sufficiently queasy that they opt for the second technique, though the heap pollution is harmless in this situation.

The following program demonstrates the use of our generic Stack class. The program prints its command line arguments in reverse order and converted to uppercase. No explicit cast is necessary to invoke String's toUpperCase method on the elements popped from the stack, and the automatically generated cast is guaranteed to succeed:

```java
// Little program to exercise our generic Stack
public static void main(String[] args) {
    Stack<String> stack = new Stack<>();
    for (String arg : args)
        stack.push(arg);
    while (!stack.isEmpty())
        System.out.println(stack.pop().toUpperCase());
    }
```
The foregoing example may appear to contradict Item 28, which encourages the use of lists in preference to arrays. It is not always possible or desirable to use lists inside your generic types. Java doesn't support lists natively, so some generic types, such as `ArrayList`, must be implemented atop arrays. Other generic types, such as `HashMap`, are implemented atop arrays for performance.

The great majority of generic types are like our `Stack` example in that their type parameters have no restrictions: you can create a `Stack<Object>`, `Stack<int[]>`, `Stack<List<String>>`, or `Stack` of any other object reference type. Note that you can't create a `Stack` of a primitive type: trying to create a `Stack<int>` or `Stack<double>` will result in a compile-time error. This is a fundamental limitation of Java's generic type system. You can work around this restriction by using boxed primitive types (Item 61).

There are some generic types that restrict the permissible values of their type parameters. For example, consider `java.util.concurrent.DelayQueue`, whose declaration looks like this:

```java
class DelayQueue<E extends Delayed> implements BlockingQueue<E>
```

The type parameter list (`<E extends Delayed>`) requires that the actual type parameter `E` be a subtype of `java.util.concurrent.Delayed`. This allows the `DelayQueue` implementation and its clients to take advantage of `Delayed` methods on the elements of a `DelayQueue`, without the need for explicit casting or the risk of a `ClassCastException`. The type parameter `E` is known as a banded type parameter. Note that the subtype relation is defined so that every type is a subtype of itself [JLS, 4.10], so it is legal to create a `DelayQueue<Delayed>`.

In summary, generic types are safer and easier to use than types that require casts in client code. When you design new types, make sure that they can be used without such casts. This will often mean making the types generic. If you have any existing types that should be generic but aren't, generify them. This will make life easier for new users of these types without breaking existing clients (Item 26).
Item 30: Favor generic methods

Just as classes can be generic, so can methods. Static utility methods that operate on parameterized types are usually generic. All of the “algorithm” methods in Collections (such as binarySearch and sort) are generic.

Writing generic methods is similar to writing generic types. Consider this deficient method, which returns the union of two sets:

```java
// Uses raw types - unacceptable! (Item 26)
public static Set union(Set s1, Set s2) {
    Set result = new HashSet(s1);
    result.addAll(s2);
    return result;
}
```

This method compiles but with two warnings:

```
Union.java:5: warning: [unchecked] unchecked call to
HashSet(Collection<? extends E>) as a member of raw type HashSet
Set result = new HashSet(s1);
   ^
Union.java:6: warning: [unchecked] unchecked call to
addAll(Collection<? extends E>) as a member of raw type Set
result.addAll(s2);
   ^
```

To fix these warnings and make the method typesafe, modify its declaration to declare a type parameter representing the element type for the three sets (the two arguments and the return value) and use this type parameter throughout the method. The type parameter list, which declares the type parameters, goes between a method’s modifiers and its return type. In this example, the type parameter list is `<E>`, and the return type is `Set<E>`. The naming conventions for type parameters are the same for generic methods and generic types (Items 29, 68):

```java
// Generic method
public static <E> Set<E> union(Set<E> s1, Set<E> s2) {
    Set<E> result = new HashSet<>(s1);
    result.addAll(s2);
    return result;
}
```

At least for simple generic methods, that’s all there is to it. This method compiles without generating any warnings and provides type safety as well as ease of
use. Here's a simple program to exercise the method. This program contains no casts and compiles without errors or warnings:

```java
// Simple program to exercise generic method
public static void main(String[] args) {
    Set<String> guys = Set.of("Tom", "Dick", "Harry");
    Set<String> stooges = Set.of("Larry", "Moe", "Curly");
    Set<String> aflCio = union(guys, stooges);
    System.out.println(aflCio);
}
```

When you run the program, it prints [Moe, Tom, Harry, Larry, Curly, Dick]. (The order of the elements in the output is implementation-dependent.)

A limitation of the union method is that the types of all three sets (both input parameters and the return value) have to be exactly the same. You can make the method more flexible by using bounded wildcard types (Item 31).

On occasion, you will need to create an object that is immutable but applicable to many different types. Because generics are implemented by erasure (Item 28), you can use a single object for all required type parameterizations, but you need to write a static factory method to repeatedly dole out the object for each requested type parameterization. This pattern, called the generic singleton factory, is used for function objects (Item 42) such as Collections.reverseOrder, and occasionally for collections such as Collections.emptySet.

Suppose that you want to write an id entity function dispenser. The libraries provide Function.identity, so there's no reason to write your own (Item 59), but it is instructive. It would be wasteful to create a new identity function object time one is requested, because it's stateless. If Java's generics were reified, you would need one identity function per type, but since they're erased a generic singleton will suffice. Here's how it looks:

```java
// Generic singleton factory pattern
private static UnaryOperator<Object> IDENTITY_FN = (t) -> t;

@SuppressWarnings("unchecked")
public static <T> UnaryOperator<T> identityFunction() {
    return (UnaryOperator<T>) IDENTITY_FN;
}
```

The cast of IDENTITY_FN to (UnaryFunction<T>) generates an unchecked cast warning, as UnaryOperator<Object> is not a UnaryOperator<T> for every T. But the identity function is special: it returns its argument unmodified, so we know that it is typesafe to use it as a UnaryFunction<T>, whatever the value of T.
Therefore, we can confidently suppress the unchecked cast warning generated by this cast. Once we’ve done this, the code compiles without error or warning.

Here is a sample program that uses our generic singleton as a UnaryOperator<String> and a UnaryOperator<Number>. As usual, it contains no casts and compiles without errors or warnings:

```java
// Sample program to exercise generic singleton
public static void main(String[] args) {
    String[] strings = {"jute", "hemp", "nylon"};
    UnaryOperator<String> sameString = identityFunction();
    for (String s : strings)
        System.out.println(sameString.apply(s));

    Number[] numbers = {1, 2.0, 3L};
    UnaryOperator<Number> sameNumber = identityFunction();
    for (Number n : numbers)
        System.out.println(sameNumber.apply(n));
}
```

It is permissible, though relatively rare, for a type parameter to be bounded by some expression involving that type parameter itself. This is what’s known as a recursive type bound. A common use of recursive type bounds is in connection with the Comparable interface, which defines a type’s natural ordering (Item 14). This interface is shown here:

```java
public interface Comparable<T> {
    int compareTo(T o);
}
```

The type parameter T defines the type to which elements of the type implementing Comparable<T> can be compared. In practice, nearly all types can be compared only to elements of their own type. So, for example, String implements Comparable<String>, Integer implements Comparable<Integer>, and so on.

Many methods take a collection of elements implementing Comparable to sort it, search within it, calculate its minimum or maximum, and the like. To do these things, it is required that every element in the collection be comparable to every other element in it, in other words, that the elements of the list be mutually comparable. Here is how to express that constraint:

```java
// Using a recursive type bound to express mutual comparability
public static <E extends Comparable<E>> E max(Collection<E> c);
```
The type bound `<E extends Comparable<E>>` may be read as "any type E that can be compared to itself," which corresponds more or less precisely to the notion of mutual comparability.

Here is a method to go with the previous declaration. It calculates the maximum value in a collection according to its elements’ natural order, and it compiles without errors or warnings:

```java
// Returns max value in a collection - uses recursive type bound
public static <E extends Comparable<E>> E max(Collection<E> c) {
    if (c.isEmpty())
        throw new IllegalArgumentException("Empty collection");
    E result = null;
    for (E e : c)
        if (result == null || e.compareTo(result) > 0)
            result = Objects.requireNonNull(e);
    return result;
}
```

Note that this method throws `IllegalArgumentException` if the list is empty. A better alternative would be to return an `Optional<E>` (Item 55).

Recursive type bounds can get much more complex, but luckily they rarely do. If you understand this idiom, its wildcard variant (Item 31), and the simulated self-type idiom (Item 2), you'll be able to deal with most of the recursive type bounds you encounter in practice.

In summary, generic methods, like generic types, are safer and easier to use than methods requiring their clients to put explicit casts on input parameters and return values. Like types, you should make sure that your methods can be used without casts, which often means making them generic. And like types, you should generify existing methods whose use requires casts. This makes life easier for new users without breaking existing clients (Item 26).
Item 31: Use bounded wildcards to increase API flexibility

As noted in Item 28, parameterized types are *invariant*. In other words, for any two distinct types Type1 and Type2, List<Type1> is neither a subtype nor a supertype of List<Type2>. Although it is counterintuitive that List<String> is not a subtype of List<Object>, it really does make sense. You can put any object into a List<Object>, but you can put only strings into a List<String>. Since a List<String> can't do everything a List<Object> can, it isn't a subtype (by the Liskov substitution principal, Item 10).

Sometimes you need more flexibility than invariant typing can provide. Consider the Stack class from Item 29. To refresh your memory, here is its public API:

```java
public class Stack<E> {
    public Stack();
    public void push(E e);
    public E pop();
    public boolean isEmpty();
}
```

Suppose we want to add a method that takes a sequence of elements and pushes them all onto the stack. Here's a first attempt:

```java
// pushAll method without wildcard type - deficient!
public void pushAll(Iterable<E> src) {
    for (E e : src)
        push(e);
}
```

This method compiles cleanly, but it isn't entirely satisfactory. If the element type of the Iterable src exactly matches that of the stack, it works fine. But suppose you have a Stack<Number> and you invoke `push(intVal)`, where `intVal` is of type Integer. This works because Integer is a subtype of Number. So logically, it seems that this should work, too:

```java
Stack<Number> numberStack = new Stack<>();
Iterable<Integer> integers = ... ;
numberStack.pushAll(integers);
```

If you try it, however, you'll get this error message because parameterized types are invariant:

```
StackTest.java:7: error: incompatible types: Iterable<Integer>
    numberStack.pushAll(integers);
```
Luckily, there’s a way out. The language provides a special kind of parameterized type call a *bounded wildcard type* to deal with situations like this. The type of the input parameter to *pushAll* should not be “Iterable of E” but “Iterable of some subtype of E,” and there is a wildcard type that means precisely that: `Iterable<? extends E>`. (The use of the keyword *extends* is slightly misleading: recall from Item 29 that *subtype* is defined so that every type is a subtype of itself, even though it does not extend itself.) Let’s modify *pushAll* to use this type:

```java
// Wildcard type for a parameter that serves as an E producer
public void pushAll(Iterable<? extends E> src) {
    for (E e : src)
        push(e);
}
```

With this change, not only does *Stack* compile cleanly, but so does the client code that wouldn’t compile with the original *pushAll* declaration. Because *Stack* and its client compile cleanly, you know that everything is typesafe.

Now suppose you want to write a *popAll* method to go with *pushAll*. The *popAll* method pops each element off the stack and adds the elements to the given collection. Here’s how a first attempt at writing the *popAll* method might look:

```java
// popAll method without wildcard type - deficient!
public void popAll(Collection<E> dst) {
    while (!isEmpty())
        dst.add(pop());
}
```

Again, this compiles cleanly and works fine if the element type of the destination collection exactly matches that of the stack. But again, it isn’t entirely satisfactory. Suppose you have a *Stack<Number>* and variable of type *Object*. If you pop an element from the stack and store it in the variable, it compiles and runs without error. So shouldn’t you be able to do this, too?

```java
Stack<Number> numberStack = new Stack<Number>();
Collection<Object> objects = ...;
numberStack.popAll(objects);
```

If you try to compile this client code against the version of *popAll* shown earlier, you’ll get an error very similar to the one that we got with our first version of *pushAll*: `Collection<Object>` is not a subtype of `Collection<Number>`. Once again, wildcard types provide a way out. The type of the input parameter to
popAll should not be "collection of E" but "collection of some supertype of E" (where supertype is defined such that E is a supertype of itself [JLS, 4.10]). Again, there is a wildcard type that means precisely that: Collection<? super E>. Let's modify popAll to use it:

```java
// Wildcard type for parameter that serves as an E consumer
public void popAll(Collection<? super E> dst) {
    while (!isEmpty())
        dst.add(pop());
}
```

With this change, both Stack and the client code compile cleanly.

The lesson is clear. **For maximum flexibility, use wildcard types on input parameters that represent producers or consumers.** If an input parameter is both a producer and a consumer, then wildcard types will do you no good: you need an exact type match, which is what you get without any wildcards.

Here is a mnemonic to help you remember which wildcard type to use:

**PECS stands for producer-extends, consumer-super.**

In other words, if a parameterized type represents a T producer, use <? extends T>; if it represents a T consumer, use <? super T>. In our Stack example, pushAll's `src` parameter produces E instances for use by the Stack, so the appropriate type for `src` is `Iterable<? extends E>`. popAll's `dst` parameter consumes E instances from the Stack, so the appropriate type for `dst` is `Collection<? super E>`. The PECS mnemonic captures the fundamental principle that guides the use of wildcard types. Naftalin and Wadler call it the *Get and Put Principle* [Naftalin07, 2.4].

With this mnemonic in mind, let's take a look at some method and constructor declarations from previous items in this chapter. The Chooser constructor in Item 28 has this declaration:

```java
public Chooser(Collection<T> choices)
```

This constructor uses the collection choices only to produce values of type T (and stores them for later use), so its declaration should use a wildcard type that extends T. Here's the resulting constructor declaration:

```java
// Wildcard type for parameter that serves as an T producer
public Chooser(Collection<? extends T> choices)
```

And would this change make any difference in practice? Yes, it would. Suppose you have a `List<Integer>`, and you want to pass it in to the constructor
for a `Chooser<Number>`.

This would not compile with the original declaration, but it does once you add the bounded wildcard type to the declaration.

Now let's look at the `union` method from Item 30. Here is the declaration:

```java
public static <E> Set<E> union(Set<E> s1, Set<E> s2)
```

Both parameters, `s1` and `s2`, are `E` producers, so the PECS mnemonic tells us that the declaration should be as follows:

```java
public static <E> Set<E> union(Set<? extends E> s1,
                Set<? extends E> s2)
```

Note that the return type is still `Set<E>`. **Do not use bounded wildcard types as return types.** Rather than providing additional flexibility for your users, it would force them to use wildcard types in client code. With the revised declaration, this code will compile cleanly:

```java
Set<Integer> integers = Set.of(1, 3, 5);
Set<Double> doubles = Set.of(2.0, 4.0, 6.0);
Set<Number> numbers = union(integers, doubles);
```

Properly used, wildcard types are nearly invisible to the users of a class. They cause methods to accept the parameters they should accept and reject those they should reject. **If the user of a class has to think about wildcard types, there is probably something wrong with its API.**

Prior to Java 8, the type inference rules were not clever enough to handle the previous code fragment, which requires the compiler to use the contextually specified return type (or target type) to infer the type of `E`. The target type of the `union` invocation shown earlier is `Set<Number>`. If you try to compile the fragment in an earlier version of Java (with an appropriate replacement for the `Set.of` factory), you’ll get a long, convoluted error message like this:

```java
Union.java:14: error: incompatible types
  ^
    required: Set<Number>
    found:    Set<INT#1>
where INT#1,INT#2 are intersection types:
  INT#1 extends Number,Comparable<? extends INT#2>
  INT#2 extends Number,Comparable<?>
```

Luckily there is a way to deal with this sort of error. If the compiler doesn’t infer the correct type, you can always tell it what type to use with an **explicit type**
ITEM 31: USE BOUNDED WILDCARDS TO INCREASE API FLEXIBILITY

argument [JLS, 15.12]. Even prior to the introduction of target typing in Java 8,
this isn’t something that you had to do often, which is good because explicit type
arguments aren’t very pretty. With the addition of an explicit type argument, as
shown here, the code fragment compiles cleanly in versions prior to Java 8:

```java
// Explicit type parameter - required prior to Java 8
Set<Number> numbers = Union.<Number>union(integers, doubles);
```

Next let’s turn our attention to the `max` method in Item 30. Here is the original
declaration:

```java
public static <T extends Comparable<T>> T max(List<T> list)
```

Here is a revised declaration that uses wildcard types:

```java
public static <T extends Comparable<? super T>> T max(List<? extends T> list)
```

To get the revised declaration from the original, we applied the PECS heuristic
twice. The straightforward application is to the parameter `list`. It produces `T`
instances, so we change the type from `List<T>` to `List<? extends T>`. The tricky
application is to the type parameter `T`. This is the first time we’ve seen a wildcard
applied to a type parameter. Originally, `T` was specified to extend `Comparable<T>`,
but a comparable of `T` consumes `T` instances (and produces integers indicating
order relations). Therefore, the parameterized type `Comparable<T>` is replaced by
the bounded wildcard type `Comparable<? super T>`. Comparables are always
consumers, so you should generally use `Comparable<? super T> in preference to
Comparable<T>`. The same is true of comparators; therefore, you should generally
use `Comparator<? super T> in preference to Comparator<T>`.

The revised max declaration is probably the most complex method declaration
in this book. Does the added complexity really buy you anything? Again, it does.
Here is a simple example of a list that would be excluded by the original declara-
dation but is permitted by the revised one:

```java
List<ScheduledFuture<?>> scheduledFutures = ... ;
```

The reason that you can’t apply the original method declaration to this list is
that `ScheduledFuture` does not implement `Comparable<ScheduledFuture>`. Instead, it is a subinterface of `Delayed`, which extends `Comparable<Delayed>`. In
other words, a `ScheduledFuture` instance isn't merely comparable to other
ScheduledFuture instances; it is comparable to any Delayed instance, and that’s enough to cause the original declaration to reject it. More generally, the wildcard is required to support types that do not implement Comparable (or Comparator) directly but extend a type that does.

There is one more wildcard-related topic that bears discussing. There is a duality between type parameters and wildcards, and many methods can be declared using one or the other. For example, here are two possible declarations for a static method to swap two indexed items in a list. The first uses an unbounded type parameter (Item 30) and the second an unbounded wildcard:

// Two possible declarations for the swap method
public static <E> void swap(List<E> list, int i, int j);
public static void swap(List<?> list, int i, int j);

Which of these two declarations is preferable, and why? In a public API, the second is better because it’s simpler. You pass in a list—any list—and the method swaps the indexed elements. There is no type parameter to worry about. As a rule, if a type parameter appears only once in a method declaration, replace it with a wildcard. If it’s an unbounded type parameter, replace it with an unbounded wildcard; if it’s a bounded type parameter, replace it with a bounded wildcard.

There’s one problem with the second declaration for swap. The straightforward implementation won’t compile:

    public static void swap(List<?> list, int i, int j) {
        list.set(i, list.set(j, list.get(i)));
    }

Trying to compile it produces this less-than-helpful error message:

    Swap.java:5: error: incompatible types: Object cannot be converted to CAP#1
        list.set(i, list.set(j, list.get(i)));
        ^
    where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?

It doesn’t seem right that we can’t put an element back into the list that we just took it out of. The problem is that the type of list is List<?>, and you can’t put any value except null into a List<?>. Fortunately, there is a way to implement this method without resorting to an unsafe cast or a raw type. The idea is to write a
private helper method to capture the wildcard type. The helper method must be a
generic method in order to capture the type. Here’s how it looks:

```java
public static void swap(List<?> list, int i, int j) {
    swapHelper(list, i, j);
}

// Private helper method for wildcard capture
private static <E> void swapHelper(List<E> list, int i, int j) {
    list.set(i, list.set(j, list.get(i)));
}
```

The `swapHelper` method knows that `list` is a `List<E>`. Therefore, it knows
that any value it gets out of this list is of type `E` and that it’s safe to put any value of
type `E` into the list. This slightly convoluted implementation of `swap` compiles cleanly. It allows us to export the nice wildcard-based declaration, while taking
advantage of the more complex generic method internally. Clients of the `swap`
method don’t have to confront the more complex `swapHelper` declaration, but
they do benefit from it. It is worth noting that the helper method has precisely the
signature that we dismissed as too complex for the public method.

In summary, using wildcard types in your APIs, while tricky, makes the APIs
far more flexible. If you write a library that will be widely used, the proper use of
wildcard types should be considered mandatory. Remember the basic rule:
producer-extends, consumer-super (PECS). Also remember that all comparables
and comparators are consumers.
**Item 32: Combine generics and varargs judiciously**

Varargs methods (Item 53) and generics were both added to the platform in Java 5, so you might expect them to interact gracefully; sadly, they do not. The purpose of varargs is to allow clients to pass a variable number of arguments to a method, but it is a *leaky abstraction*: when you invoke a varargs method, an array is created to hold the varargs parameters; that array, which should be an implementation detail, is visible. As a consequence, you get confusing compiler warnings when varargs parameters have generic or parameterized types.

Recall from Item 28 that a non-reifiable type is one whose runtime representation has less information than its compile-time representation, and that nearly all generic and parameterized types are non-reifiable. If a method declares its varargs parameter to be of a non-reifiable type, the compiler generates a warning on the declaration. If the method is invoked on varargs parameters whose inferred type is non-reifiable, the compiler generates a warning on the invocation too. The warnings look something like this:

```
warning: [unchecked] Possible heap pollution from parameterized vararg type List<String>
```

*Heap pollution* occurs when a variable of a parameterized type refers to an object that is not of that type [JLS, 4.12.2]. It can cause the compiler’s automatically generated casts to fail, violating the fundamental guarantee of the generic type system.

For example, consider this method, which is a thinly disguised variant of the code fragment on page 127:

```java
// Mixing generics and varargs can violate type safety!
static void dangerous(List<String>... stringLists) {
    List<Integer> intList = List.of(42);
    Object[] objects = stringLists;
    objects[0] = intList;  // Heap pollution
    String s = stringLists[0].get(0);  // ClassCastException
}
```

This method has no visible casts yet throws a `ClassCastException` when invoked with one or more arguments. Its last line has an invisible cast that is generated by the compiler. This cast fails, demonstrating that type safety has been compromised, and it is unsafe to store a value in a generic varargs array parameter.

This example raises an interesting question: Why is it even legal to declare a method with a generic varargs parameter, when it is illegal to create a generic array explicitly? In other words, why does the method shown previously generate only a warning, while the code fragment on page 127 generates an error? The
answer is that methods with varargs parameters of generic or parameterized types can be very useful in practice, so the language designers opted to live with this inconsistency. In fact, the Java libraries export several such methods, including Arrays.asList(T... a), Collections.addAll(Collection<? super T> c, T... elements), and EnumSet.of(E first, E... rest). Unlike the dangerous method shown earlier, these library methods are typesafe.

Prior to Java 7, there was nothing the author of a method with a generic varargs parameter could do about the warnings at the call sites. This made these APIs unpleasant to use. Users had to put up with the warnings or, preferably, to eliminate them with @SuppressWarnings("unchecked") annotations at every call site (Item 27). This was tedious, harmed readability, and hid warnings that flagged real issues.

In Java 7, the SafeVarargs annotation was added to the platform, to allow the author of a method with a generic varargs parameter to suppress client warnings automatically. In essence, the SafeVarargs annotation constitutes a promise by the author of a method that it is typesafe. In exchange for this promise, the compiler agrees not to warn the users of the method that calls may be unsafe.

It is critical that you do not annotate a method with @SafeVarargs unless it actually is safe. So what does it take to ensure this? Recall that a generic array is created when the method is invoked, to hold the varargs parameters. If the method doesn’t store anything into the array (which would overwrite the parameters) and doesn’t allow a reference to the array to escape (which would enable untrusted code to access the array), then it’s safe. In other words, if the varargs parameter array is used only to transmit a variable number of arguments from the caller to the method—which is, after all, the purpose of varargs—then the method is safe.

It is worth noting that you can violate type safety without ever storing anything in the varargs parameter array. Consider the following generic varargs method, which returns an array containing its parameters. At first glance, it may look like a handy little utility:

// UNSAFE - Exposes a reference to its generic parameter array!
static <T> T[] toArray(T... args) {
    return args;
}

This method simply returns its varargs parameter array. The method may not look dangerous, but it is! The type of this array is determined by the compile-time types of the arguments passed in to the method, and the compiler may not have enough information to make an accurate determination. Because this method returns its varargs parameter array, it can propagate heap pollution up the call stack.
To make this concrete, consider the following generic method, which takes three arguments of type T and returns an array containing two of the arguments, chosen at random:

```java
static <T> T[] pickTwo(T a, T b, T c) {
    switch(ThreadLocalRandom.current().nextInt(3)) {
        case 0: return toArray(a, b);
        case 1: return toArray(a, c);
        case 2: return toArray(b, c);
    }
    throw new AssertionError(); // Can't get here
}
```

This method is not, in and of itself, dangerous and would not generate a warning except that it invokes the toArray method, which has a generic varargs parameter.

When compiling this method, the compiler generates code to create a varargs parameter array in which to pass two T instances to toArray. This code allocates an array of type Object[], which is the most specific type that is guaranteed to hold these instances, no matter what types of objects are passed to pickTwo at the call site. The toArray method simply returns this array to pickTwo, which in turn returns it to its caller, so pickTwo will always return an array of type Object[].

Now consider this main method, which exercises pickTwo:

```java
public static void main(String[] args) {
    String[] attributes = pickTwo("Good", "Fast", "Cheap");
}
```

There is nothing at all wrong with this method, so it compiles without generating any warnings. But when you run it, it throws a ClassCastException, though it contains no visible casts. What you don’t see is that the compiler has generated a hidden cast to String[] on the value returned by pickTwo so that it can be stored in attributes. The cast fails, because Object[] is not a subtype of String[]. This failure is quite disconcerting because it is two levels removed from the method that actually causes the heap pollution (toArray), and the varargs parameter array is not modified after the actual parameters are stored in it.

This example is meant to drive home the point that it is unsafe to give another method access to a generic varargs parameter array, with two exceptions: it is safe to pass the array to another varargs method that is correctly annotated with @SafeVarargs, and it is safe to pass the array to a non-varargs method that merely computes some function of the contents of the array.
Here is a typical example of a safe use of a generic varargs parameter. This method takes an arbitrary number of lists as arguments and returns a single list containing the elements of all of the input lists in sequence. Because the method is annotated with @SafeVarargs, it doesn’t generate any warnings, on the declaration or at its call sites:

```java
// Safe method with a generic varargs parameter
@SafeVarargs
static <T> List<T> flatten(List<? extends T>... lists) {
    List<T> result = new ArrayList<>();
    for (List<? extends T> list : lists)
        result.addAll(list);
    return result;
}
```

The rule for deciding when to use the SafeVarargs annotation is simple: Use @SafeVarargs on every method with a varargs parameter of a generic or parameterized type, so its users won’t be burdened by needless and confusing compiler warnings. This implies that you should never write unsafe varargs methods like dangerous or toArray. Every time the compiler warns you of possible heap pollution from a generic varargs parameter in a method you control, check that the method is safe. As a reminder, a generic varargs methods is safe if:

1. it doesn’t store anything in the varargs parameter array, and
2. it doesn’t make the array (or a clone) visible to untrusted code.

If either of these prohibitions is violated, fix it.

Note that the SafeVarargs annotation is legal only on methods that can’t be overridden, because it is impossible to guarantee that every possible overriding method will be safe. In Java 8, the annotation was legal only on static methods and final instance methods; in Java 9, it became legal on private instance methods as well.

An alternative to using the SafeVarargs annotation is to take the advice of Item 28 and replace the varargs parameter (which is an array in disguise) with a List parameter. Here’s how this approach looks when applied to our flatten method. Note that only the parameter declaration has changed:

```java
// List as a typesafe alternative to a generic varargs parameter
static <T> List<T> flatten(List<List<? extends T>> lists) {
    List<T> result = new ArrayList<>();
    for (List<? extends T> list : lists)
        result.addAll(list);
    return result;
}
```
This method can then be used in conjunction with the static factory method `List.of` to allow for a variable number of arguments. Note that this approach relies on the fact that the `List.of` declaration is annotated with `@SafeVarargs`:

```java
audience = flatten(List.of(friends, romans, countrymen));
```

The advantage of this approach is that the compiler can prove that the method is typesafe. You don’t have to vouch for its safety with a `SafeVarargs` annotation, and you don’t have worry that you might have erred in determining that it was safe. The main disadvantage is that the client code is a bit more verbose and may be a bit slower.

This trick can also be used in situations where it is impossible to write a safe varargs method, as is the case with the `toArray` method on page 147. Its `List` analogue is the `List.of` method, so we don’t even have to write it; the Java libraries authors have done the work for us. The `pickTwo` method then becomes this:

```java
static <T> List<T> pickTwo(T a, T b, T c) {
    switch(rnd.nextInt(3)) {
        case 0: return List.of(a, b);
        case 1: return List.of(a, c);
        case 2: return List.of(b, c);
    }
    throw new AssertionError();
}
```

and the main method becomes this:

```java
public static void main(String[] args) {
    List<String> attributes = pickTwo("Good", "Fast", "Cheap");
}
```

The resulting code is typesafe because it uses only generics, and not arrays.

In summary, varargs and generics do not interact well because the varargs facility is a leaky abstraction built atop arrays, and arrays have different type rules from generics. Though generic varargs parameters are not typesafe, they are legal. If you choose to write a method with a generic (or parameterized) varargs parameter, first ensure that the method is typesafe, and then annotate it with `@SafeVarargs` so it is not unpleasant to use.
Item 33: Consider typesafe heterogeneous containers

Common uses of generics include collections, such as Set<E> and Map<K,V>, and single-element containers, such as ThreadLocal<T> and AtomicReference<T>. In all of these uses, it is the container that is parameterized. This limits you to a fixed number of type parameters per container. Normally that is exactly what you want. A Set has a single type parameter, representing its element type; a Map has two, representing its key and value types; and so forth.

Sometimes, however, you need more flexibility. For example, a database row can have arbitrarily many columns, and it would be nice to be able to access all of them in a typesafe manner. Luckily, there is an easy way to achieve this effect. The idea is to parameterize the key instead of the container. Then present the parameterized key to the container to insert or retrieve a value. The generic type system is used to guarantee that the type of the value agrees with its key.

As a simple example of this approach, consider a Favorites class that allows its clients to store and retrieve a favorite instance of arbitrarily many types. The Class object for the type will play the part of the parameterized key. The reason this works is that class Class is generic. The type of a class literal is not simply Class, but Class<T>. For example, String.class is of type Class<String>, and Integer.class is of type Class<Integer>. When a class literal is passed among methods to communicate both compile-time and runtime type information, it is called a type token [Bracha04].

The API for the Favorites class is simple. It looks just like a simple map, except that the key is parameterized instead of the map. The client presents a Class object when setting and getting favorites. Here is the API:

```
// Typesafe heterogeneous container pattern - API
public class Favorites {
    public <T> void putFavorite(Class<T> type, T instance);
    public <T> T getFavorite(Class<T> type);
}
```

Here is a sample program that exercises the Favorites class, storing, retrieving, and printing a favorite String, Integer, and Class instance:

```
// Typesafe heterogeneous container pattern - client
public static void main(String[] args) {
    Favorites f = new Favorites();
    f.putFavorite(String.class, "Java");
    f.putFavorite(Integer.class, 0xcafebabe);
    f.putFavorite(Class.class, Favorites.class);
}
```
CHAPTER 5 GENERICS

String favoriteString = f.getFavorite(String.class);
int favoriteInteger = f.getFavorite(Integer.class);
Class<?> favoriteClass = f.getFavorite(Class.class);
System.out.printf("%s %x %s\n", favoriteString,
favoriteInteger, favoriteClass.getName());
}

As you would expect, this program prints Java cafebabe Favorites. Note, incidentally, that Java’s printf method differs from C’s in that you should use %n where you’d use \n in C. The %n generates the applicable platform-specific line separator, which is \n on many but not all platforms.

A Favorites instance is typesafe: it will never return an Integer when you ask it for a String. It is also heterogeneous: unlike an ordinary map, all the keys are of different types. Therefore, we call Favorites a typesafe heterogeneous container.

The implementation of Favorites is surprisingly tiny. Here it is, in its entirety:

// Typesafe heterogeneous container pattern - implementation
public class Favorites {
    private Map<Class<?>, Object> favorites = new HashMap<>();

    public <T> void putFavorite(Class<T> type, T instance) {
        favorites.put(Objects.requireNonNull(type), instance);
    }

    public <T> T getFavorite(Class<T> type) {
        return type.cast(favorites.get(type));
    }
}

There are a few subtle things going on here. Each Favorites instance is backed by a private Map<Class<?>, Object> called favorites. You might think that you couldn’t put anything into this Map because of the unbounded wildcard type, but the truth is quite the opposite. The thing to notice is that the wildcard type is nested: it’s not the type of the map that’s a wildcard type but the type of its key. This means that every key can have a different parameterized type: one can be Class<String>, the next Class<Integer>, and so on. That’s where the heterogeneity comes from.

The next thing to notice is that the value type of the favorites Map is simply Object. In other words, the Map does not guarantee the type relationship between keys and values, which is that every value is of the type represented by its key. In
fact, Java’s type system is not powerful enough to express this. But we know that it’s true, and we take advantage of it when the time comes to retrieve a favorite.

The putFavorite implementation is trivial: it simply puts into favorites a mapping from the given Class object to the given favorite instance. As noted, this discards the “type linkage” between the key and the value; it loses the knowledge that the value is an instance of the key. But that’s OK, because the getFavorites method can and does reestablish this linkage.

The implementation of getFavorite is trickier than that of putFavorite. First, it gets from the favorites map the value corresponding to the given Class object. This is the correct object reference to return, but it has the wrong compile-time type: it is Object (the value type of the favorites map) and we need to return a T. So, the getFavorite implementation dynamically casts the object reference to the type represented by the Class object, using Class’s cast method.

The cast method is the dynamic analogue of Java’s cast operator. It simply checks that its argument is an instance of the type represented by the Class object. If so, it returns the argument; otherwise it throws a ClassCastException. We know that the cast invocation in getFavorite won’t throw ClassCastException, assuming the client code compiled cleanly. That is to say, we know that the values in the favorites map always match the types of their keys.

So what does the cast method do for us, given that it simply returns its argument? The signature of the cast method takes full advantage of the fact that class Class is generic. Its return type is the type parameter of the Class object:

```java
public class Class<T> {
    T cast(Object obj);
}
```

This is precisely what’s needed by the getFavorite method. It is what allows us to make Favorites typesafe without resorting to an unchecked cast to T.

There are two limitations to the Favorites class that are worth noting. First, a malicious client could easily corrupt the type safety of a Favorites instance, by using a Class object in its raw form. But the resulting client code would generate an unchecked warning when it was compiled. This is no different from a normal collection implementations such as HashSet and HashMap. You can easily put a String into a HashSet<Integer> by using the raw type HashSet (Item 26). That said, you can have runtime type safety if you’re willing to pay for it. The way to ensure that Favorites never violates its type invariant is to have the putFavorite
method check that instance is actually an instance of the type represented by type, and we already know how to do this. Just use a dynamic cast:

```java
// Achieving runtime type safety with a dynamic cast
public <T> void putFavorite(Class<T> type, T instance) {
    favorites.put(type, type.cast(instance));
}
```

There are collection wrappers in `java.util.Collections` that play the same trick. They are called `checkedSet`, `checkedList`, `checkedMap`, and so forth. Their static factories take a `Class` object (or two) in addition to a collection (or map). The static factories are generic methods, ensuring that the compile-time types of the `Class` object and the collection match. The wrappers add reification to the collections they wrap. For example, the wrapper throws a `ClassCastException` at runtime if someone tries to put a `Coin` into your `Collection<Stamp>`. These wrappers are useful for tracking down client code that adds an incorrectly typed element to a collection, in an application that mixes generic and raw types.

The second limitation of the `Favorites` class is that it cannot be used on a non-reifiable type (Item 28). In other words, you can store your favorite `String` or `String[]`, but not your favorite `List<String>`. If you try to store your favorite `List<String>`, your program won’t compile. The reason is that you can’t get a `Class` object for `List<String>`. The class literal `List<String>.class` is a syntax error, and it’s a good thing, too. `List<String>` and `List<Integer>` share a single `Class` object, which is `List.class`. It would wreak havoc with the internals of a `Favorites` object if the “type literals” `List<String>.class` and `List<Integer>.class` were legal and returned the same object reference. There is no entirely satisfactory workaround for this limitation.

The type tokens used by `Favorites` are unbounded: `getFavorite` and `putFavorite` accept any `Class` object. Sometimes you may need to limit the types that can be passed to a method. This can be achieved with a bounded type token, which is simply a type token that places a bound on what type can be represented, using a bounded type parameter (Item 30) or a bounded wildcard (Item 31).

The annotations API (Item 39) makes extensive use of bounded type tokens. For example, here is the method to read an annotation at runtime. This method comes from the `AnnotatedElement` interface, which is implemented by the reflective types that represent classes, methods, fields, and other program elements:

```java
public <T extends Annotation>
    T getAnnotation(Class<T> annotationType);
```
The argument, annotationType, is a bounded type token representing an annotation type. The method returns the element's annotation of that type, if it has one, or null, if it doesn’t. In essence, an annotated element is a typesafe heterogeneous container whose keys are annotation types.

Suppose you have an object of type Class<?> and you want to pass it to a method that requires a bounded type token, such as getAnnotation. You could cast the object to Class<? extends Annotation>, but this cast is unchecked, so it would generate a compile-time warning (Item 27). Luckily, class Class provides an instance method that performs this sort of cast safely (and dynamically). The method is called asSubclass, and it casts the Class object on which it is called to represent a subclass of the class represented by its argument. If the cast succeeds, the method returns its argument; if it fails, it throws a ClassCastException.

Here’s how you use the asSubclass method to read an annotation whose type is unknown at compile time. This method compiles without error or warning:

```java
// Use of asSubclass to safely cast to a bounded type token
static Annotation getAnnotation(AnnotatedElement element,
                                  String annotationTypeName) {
    Class<?> annotationType = null; // Unbounded type token
    try {
        annotationType = Class.forName(annotationTypeName);
    } catch (Exception ex) {
        throw new IllegalArgumentException(ex);
    }
    return element.getAnnotation(
                                  annotationType.asSubclass(Annotation.class));
}
```

In summary, the normal use of generics, exemplified by the collections APIs, restricts you to a fixed number of type parameters per container. You can get around this restriction by placing the type parameter on the key rather than the container. You can use Class objects as keys for such typesafe heterogeneous containers. A Class object used in this fashion is called a type token. You can also use a custom key type. For example, you could have a DatabaseRow type representing a database row (the container), and a generic type Column<T> as its key.
Symbols
_ in numeric literals, 108
... variable arity argument
See varargs
<> type parameter delimiters, 117, 123
<?> unbounded wildcard type, 120
A
abstract classes
designing for inheritance, 97
vs. interfaces, 99–103
noninstantiability and, 19
subclassing, and equals, 45
access control mechanisms, 74
access levels, 73–77
module-level, 76–77
of readResolve, 362
rules of thumb for, 74–75
of static member classes, 112
access modifiers, 74
accessor methods, 78
defensive copies and, 80, 233
for failure-capture data, 297, 307
immutability and, 80
naming conventions for, 291–292
vs. public fields, 78–79
for toString data, 57
actual type parameters, 117
Adapter pattern, 23, 101, 113
aggregate types vs. strings, 276
alien methods, 317
annotations, 157, 180–192
defining, 180
detecting repeated and non-repeated, 186
documenting, 259
functional interfaces and, 202
vs. naming patterns, 180–187
synthetic, 186
See also marker annotations
anonymous classes, 112, 114, 193
in adapters, 101
prefer lambdas to, 193–196
serialization and, 196
antipatterns, 2
bit field enum pattern, 169
bounded wildcard types as return types, 142
breaking from naming conventions, 82
busy waits, 336
constant as a hashCode, 51
constant interface, 107–108
copy constructor of immutable object, 83
difference-based comparators, 71
empty catch blocks, 310
equals with unreliable resources, 45
exceptions for flow control, 293
excessive string concatenation, 279
floating point for monetary calculations, 270
hardwiring resources, 20
hashCode missing significant fields, 54
immutable classes not effectively final, 85
inappropriate subclassing, 92
int enum pattern, 157
naming patterns, 180
null returns for collections, arrays, 247–248
ordinal abuse, 168, 171
overriding equals but not hashCode, 50
serializable inner classes, 345
signatures differ by parameter type order, 6
String enum pattern, 158
string overuse, 276
tagged classes, 109–111
value-component subclasses and equals, 44
API design
access levels and, 74
bounded wildcard types and, 139–145
callbacks, 28
constant interface pattern and, 107
dependencies and, 286
inheritance and, 93–98
interfaces as parameter types, 170
member classes and, 114
performance and, 286–287
serialization and, 343–345
singletons, 18
API elements, 4
documenting, 254–260
API, toString return values as defacto, 57
arrays
ci() and, 65
covariant typing, 126
defensive copying of, 76, 234
dependency injection and, 21
PECS mnemonic for, 141
as return types, 142
vs. unbounded wildcard types, 121
boxed primitives
== operator and, 274
appropriate uses of, 275
generics and, 134
prefer primitive types to, 24, 273–275
Bridge pattern, 8
Builder pattern, 10–16
adapted for method invocation, 237
busy waits, 336
C
avoiding memory leaks from, 28
of expensive objects, 22–23
of hash codes, 53
immutable objects and, 82, 85
callback frameworks, wrapper classes and, 91
callbacks, avoiding memory leaks from, 28
canonical forms, 47
capabilities vs. strings, 276–277
casts
dynamic, 153, 155
invisible (see compiler-generated casts)
unchecked, warnings of, 127, 129, 137
char values, and streams, 206
checked exceptions
avoiding overuse of, 298–299
declaring, 304
failure atomicity and, 308
purpose of, 296
refactoring to unchecked, 299
vs. unchecked, 296–297
circularities
  in cleaners, 33
  initialization, 333, 366
  serialization attacks and, 360
Class class, as parameterized key, 151
class hierarchies, 110
  Builder pattern and, 14
  combinatorial explosions in, 100
class literals
  as annotation parameter values, 183
  as generics, 151
  raw types in, 121
class-based frameworks, 281
classes, 73–114
  access levels of, 74
  anonymous (see anonymous classes)
  base, 281
  composition, 87–92
  designing for inheritance, 93–98
  documenting
    for inheritance, 93–94
    thread safety of, 330–332
  generic, 117
  helper, for shortening parameter lists, 237
  hierarchy of (see class hierarchies)
  immutable (see immutable objects)
  implementation inheritance, 87–92
  instances of, 3
  levels of thread safety, 330
  members, 3
  minimizing accessibility of, 73–77
  mutable, and thread-safety, 322
  naming conventions for, 289–291
  nested (see nested classes)
  noninstantiable companions, 7
  reusable forwarding, 89–91
  singletons (see singletons)
  summary descriptions of, 257
SuppressWarnings annotation and, 124
tagged, vs. class hierarchies, 109–111
unintentionally instantiable, 19
unrelated, 243
utility (see utility classes)
wrapper (see wrapper classes)
See also individual class names
classifier functions, 213
cleaners, 29–33
c l o n e m e t h o d ,  58–65
  arrays and, 65
  as a constructor, 61, 96
  vs. copy or conversion constructor, 65
defensive copies and, 76, 233
  final fields and, 61
general contract, 58–59
  immutable objects and, 83
  nonfinal classes and, 233
  nonfinal methods and, 64
  overridable methods and, 96
  thread-safety and, 64
Cloneable interface, 58–65
  alternatives to, 65
  behavior of, 58
  designing for inheritance and, 64, 96
  implementing, 64
collections
  concurrent, 321, 325–326
  empty, vs. null as return value, 247
  optionals and, 252
  as return types, vs. streams, 216–221
  collectors, 211–215
  downstream, 213
Collectors API, organization of, 211–215
combinatorial explosions, 100
comparator construction methods, 70, 194
compilers, 69
  as consumers in PECS, 143
See also compareTo method
comparator construction methods, 70, 194
compilers, 69
  as consumers in PECS, 143
See also Comparable interface
compatibility
  backward, 350
  binary, 107, 305
  forward, 350
  migration, 119
  source, 305
unchecked exceptions and, 305
compiler warnings, types of, 123
compiler-generated casts, 117, 119, 127
components, 2
composition, 8, 89
equals and, 44
vs. inheritance, 87–92
conceptual weight, 7
concurrency, 311
documenting method behavior for, 330–332
improving via internal synchronization, 322
concurrency utilities, 323–329
vs. wait and notify, 325–329
concurrent collections, 321, 325–326
conditionally thread-safe classes,
documenting, 331
consistency requirements
equals, 38, 45
hashCode, 50
consistent with equals, 68
unreliable resources and, 45
constant fields, naming conventions for, 290
constant interfaces, 107
constant utility classes, 108
constants, 76
in anonymous classes, 114
naming conventions for, 290
constant-specific behaviors, 162–166
lambdas for, 195
constant-specific class bodies, 162
constant-specific method implementations
See constant-specific behaviors
constructors, 4
calling overridable methods in, 95
checking parameters of, 353
cloning as a, 61
copy and conversion, 65
default, 19
defensive copying of parameters, 232
deserialization as, 344
establishing invariants, 82, 86
noninstantiability and, 19
private (see private constructors)
readObject as a, 353
reflection and, 282
replacing with static factories, 5–9
safely overloading, 240–241
for singletons, 17–18
summary descriptions of, 257
SuppressWarnings annotation and, 124
validity checking parameters of, 229
contention, synchronization and, 321
contracts
cloning, 58
cmpareTo, 66
documentation as, 304
equals, 38–46
hashCode, 50
toString, 55
corrupted objects, 12, 30, 227, 309
countdown latches, 326
covariant arrays, 126
covariant return typing, 16, 60
creating objects, 5–33
cross-platform structured-data representations,
341
custom serialized forms, 346–352
D
data consistency
maintaining in face of failure, 308–309
synchronization for, 311–316
data corruption, 285, 312
date, replacements for, 232
deadlocks
resource ordering, 320, 351
thread starvation, 328
Decorator pattern, 91
default access
See package-private access level
default constructors, 19
default implementations, 104
default methods on interfaces, 99, 104–105
default serialized forms, 346–352
disadvantages of, 348
defensive copies, 231–235
of arrays, 234
builders and, 14
cloning and, 233
deserialization and, 353, 357
documenting, 234
immutable objects and, 83
of mutable internal fields, 233
double-check idiom, 334–335
double-check idiom, 334–335
double-precision floating-point arithmetic, 270
downstream collectors, 213
dynamic casts, 153, 155

doing.

ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
ed.
equals method (continued)
return values of, and compareTo, 68
subclassing and, 42, 45
unreliable resources and, 45
when to override, 37–38
equivalence classes, 39
equivalence relations, 38
erasure, 119, 126
errors
generic array creation, 126–127, 133
purpose of, 297
runtime, default methods and, 105
exact results, types for obtaining, 270
exception chaining, 302–303
exception translation idiom, 230, 302
exceptions, 293–310
accessor methods for, 297, 307
checked vs. unchecked, 296–297
choosing among, 301
commonly reused, 301
control flow and, 294
detail messages for, 306–307
documenting, 227, 304–305
failure-capture data, 307
for invalid method parameters, 228
ignoring, 310
logging of, 303
multi-catch facility, 320
vs. optionals or special return values, 295
prefer standard existing, 300–301
preventing, 303
vs. state testing methods, 294
suppression of, 36
unchaught, and finalizers, 30
using appropriately, 293–295
See also individual exception names
Executor Framework, 323
executor service, 323–324
explicit type arguments, 142
export declarations, 76
exported APIs
See API design; APIs
extending classes
See inheritance; subclassing
extending interfaces, 4
extensible enums, 176–179
extra-linguistic mechanisms
cloning, 58, 65
native methods, 285
reflection, 282
serialization, 344, 363
See also hidden constructors
F
Factory Method pattern, 5, 21
failure atomicity, 230, 308–309
fields
access levels of, 73–77
class invariants and, 75
constant, naming conventions for, 290
derived, 47, 52
exposing, vs. accessor methods, 78–79
final (see final fields)
initialization techniques for, 335
mutable, defensive copies of, 233
naming conventions for, 290, 292
public static final, for singletons, 17
reflection and, 282
summary descriptions of, 257
tags, 109
thread safety and, 75
final fields
for defining constants, 290
incompatible with cloning, 61
incompatible with serialization, 357
finalizer attacks, and prevention, 30–31
finalizers, 29–33
alternative to, 31
Float
for binary floating-point arithmetic, 270
when to avoid, 270–272
fluent APIs, 14, 203
Flyweight pattern, 6
footprint
See space consumption
for loops
dual variable idiom, 263
prefer for-each loops to, 264–266
vs. while loops, 262
for-each loops
limitations of, 266
prefer over for loops, 264–266
fork-join tasks and pools, 324
formal type parameters, 117
forwarding methods, 89, 102
frameworks
callback, 91
class-based, 281
curator, 323
interface-based, 6
nonhierarchical type, 99
service provider, 8
function objects, 114
vs. code blocks, 207
functional interfaces, 193
method overloading and, 243
organization of standard, 200–201
using standard, 199–202
functional programming, 82

G
gadgets, 340
garbage collection, 27, 29–30, 113
general contracts
See contracts
generic array creation errors, 126–127, 133
generic classes and interfaces, 117
generic methods, 135–138
vs. unbounded wildcard types, 121
generic singleton factories, 18, 136
generic type parameters
See type parameters
generic types, 14, 117, 130–134
documenting, 258
immutability and, 136
generic varargs parameter arrays
heap pollution from, 147–148
replacing with lists, 149
unsafe as storage, 146
unsafe to expose, 147–148
generics, 117–155
boxed primitives and, 134
compiler-generated casts and, 117
erasure and, 126
implementing atop arrays, 131–133
incompatibility with primitive types, 134
invariant typing, 126
varargs and, 127, 146–150
generifying existing code, 130
Get and Put Principle, 141

H
hashCode method
equal and, 48, 50–54
general contract for, 50
how to write, 51
immutable objects and, 53
heap pollution, 133, 146–148
heap profilers, 28
helper classes, 112
for shortening parameter lists, 237
hidden constructors, 61, 96, 339, 344, 353
See also extralinguistic mechanisms
hierarchical builder pattern, 16

I
immutable objects
canonical forms and, 47
cclone and, 59
dependency injection and, 21
empty arrays and, 248
data types and, 160
typeSets and, 170
failure atomicity and, 308
functional approach and, 82
hashCode and, 53
JavaBeans and, 12
mutable companion classes for, 84
object reuse and, 22
rules for, 80
serialization and, 85, 353–358
static factory methods and, 84
subclasseing and, 97
thread safety and, 82
imperative programming, 82
implementation details
documenting for inheritance, 94
exposing, 92
implementation inheritance, 87
implementing interfaces, 4
default methods and, 105
inconsistent with equal and, 67–68
unreliable resources and, 45
information hiding
See encapsulation
inheritance, 3
  vs. composition, 87–92
  constructors and, 95
  designing for, 93–98
  documenting for, 93–94
  encapsulation and, 87
  fragility of, 89
  hooks to facilitate, 94
  implementation vs. interface, 3, 87
  of method doc comments, 259
  multiple, simulated, 102
  self-use of overridable methods and, 98
  uses of, 92
  See also subclassing
initialization
circularities, 333, 366
defensive copying and, 80
  of fields on deserialization, 351
  incomplete, 96
  lazy (see lazy initialization)
  of local variables, 261
  normal vs. lazy, 333
  at object creation, 86
inner classes, 112
  Serializable and, 345
to extend skeletal implementations, 102
instance fields
  access levels of, 75
  initializing, 333
  lazy initialization of, 334
  vs. ordinals, 168
instance-controlled classes, 6, 158
  singletons, 17–18
  static factory methods and, 6
  utility classes, 19
  See also enum types
instanceof operator, parameter types, 121
int constants vs. enum types, 157–167
int, for monetary calculations, 270
interface-based frameworks, 6, 99–103
interface inheritance, 87
interfaces, 73–114
  vs. abstract classes, 99–103
  access levels of, 74
  accessibility of static members, 7
  default methods on, 99, 104–105
  design of, 104–106
emulating extensible enums with, 176–179
  enabling functionality enhancements, 100
generic, 117
marker (see marker interfaces)
mixin, 58, 99
naming conventions for, 289–291
for nonhierarchical type frameworks, 99
noninstantiable companion classes and, 7
  as parameter types, 170, 237
  prefer to reflection, 282
  purpose of, 58, 107–108
  for referring to objects, 280–281
  reflective instantiation of, 283–284
  serialization and, 344
  skeletal implementations and, 100–103
  static methods and, 7
  summary descriptions of, 257
  See also individual interface names
internal field theft attacks, 360–362
invariant types, 126, 139
invariants
  clone and, 61
  concurrency and, 328
  constructors and, 82, 86
  corruption of, 92
  enum types and, 362
  maintaining, 229, 234, 308
  of objects and members, 75, 78
J
JavaBeans
  immutability and, 12
  method-naming conventions, 291
  pattern, 11–12
Javadoc, 254
  architecture documents and, 260
  class-level comments, 228, 331
  client-side indexes in, 258
  comment inheritance, 259
  formatting, 255–256
  module- and package-level comments, 259
  summary descriptions, 257
K
  key extractor functions, 70
L

lambdas, 70, 193–225
  cleaners and, 33
  for constant-specific behaviors, 195
  prefer method references to, 197–198
  prefer to anonymous classes, 193–196
  serialization and, 196
lazy initialization, 23, 53, 85, 333–335
lazy initialization holder class idiom, 334–335
lazy initialization with a synchronized accessor idiom, 333–334
leaky abstractions, 146
libraries, 267–269
Liskov substitution principle, 43, 75
listeners, avoiding memory leaks from, 28
lists
  vs. arrays, 126–129
  for generic varargs parameter arrays, 149
  mutual comparability in, 138
liveness
  ensuring, 317–322, 328
  failures of, 224, 313
local classes, 112, 114
local variables
  minimizing scope of, 261–263
  naming conventions for, 290, 292
locality of reference, 223
locks
  fields containing, 332
  finalizers or cleaners and, 30
  private, 332
  reentrant, 320
logical equality, 37–38
long, for monetary calculations, 270
loops
  nested, 264–266
  See also for loops; for-each loops

M

maps
  member classes and, 113
  nested, 173–175
  vs. streams, behavior of, 173
marker annotations, 181
  vs. marker interfaces, 191
marker interfaces, 191–192
member classes, 112–114
  See also static member classes
members, 3
  minimizing accessibility of, 73–77
memory footprint
  See space consumption
memory leaks, 26–27
  nonstatic member classes and, 113
  self-management of memory, 28
memory model, 80
merge functions, 212
meta-annotations, 181
method chaining, 14
method overloading, 238–244
  accidental, of equals, 49
  effects of autoboxing and generics, 241–242
  functional interfaces and, 202, 243
  parameters and, 240
  static selection among methods, 238
method overriding, 49, 238–239
  access levels and, 75
cClone, 58
dynamic selection among methods, 238
equals, 37–49
hashCode, 50–54
self-use and, 98
toString, 55–57
unintentional, 190
method references, 18
  kinds of, 198
  prefer to lambdas, 197–198
methods, 3, 227–260
  access levels of, 74
  accessor (see accessor methods)
    alien, 317
    common to all objects, 37–72
    constant-specific, for enum-types, 162
    documenting, 254–255
    exceptions thrown by, 304–305
    overridable, 93
    summary descriptions of, 257
  thread safety of, 330–332
  failure atomicity and, 308–309
  forwarding (see forwarding methods)
generic, 121, 135–138
invocation, reflection and, 282
legal for SafeVarargs, 149
INDEX

methods (continued)
minimizing accessibility of, 73
naming conventions for, 9, 290–291
native, 31, 285
nonfinal, and clone, 64
overloading (see method overloading)
overriding (see method overriding)
parameter lists for, 236
private, to capture wildcard types, 145
shortening parameter lists of, 236
signatures of, 3, 236–237
size of, 263
state-testing, vs. special return value, 295
static factory (see static factory methods)
SuppressWarnings annotation and, 124
validity checking parameters, 227–230
varargs, 245–246
See also individual method names
mixin interfaces, 58, 99
mixing primitives, boxed primitives, 24, 274
modules, 76–77
monetary calculations, types for, 270–271
Monty Python reference, subtle, 247
multi-catch facility, 320
multiple inheritance, simulated, 102
mutability
JavaBeans pattern and, 11
minimizing, 80–86
mutable companion classes, 84
mutable reductions, 223
mutators, 78
mutual comparability, 138
mutual exclusion, 311
N
named optional parameters, 14
naming conventions, 236, 289–292
of generic type parameters, 131
grammatical, 291–292
of skeletal implementation classes, 101
of static factory methods, 9
streams and, 208
of type parameters, 135
naming patterns vs. annotations, 180–187
native methods, 31, 285
native peers, 31
natural ordering, 66
nested classes, 112
access levels of, 74
decreasing accessibility with, 74
in serialization proxy pattern, 363
types of, 112
nested interfaces, access levels of, 74
nested maps, 173–175
nonhierarchical type frameworks, 99
noninstantiable classes, 19
noninstantiable companion classes, 7
non-nullity of equals, 38, 45
non-reifiable types, 127, 131, 146
nonstatic member classes, 112–114
notify vs. notifyAll, 328–329
null checking, 228
nulling out obsolete object references, 27
NullPointerException, equals contract and, 46
O
object pools, 24
object reference fields, equals and, 47
objects, 3
avoiding reflective access, 282–284
base classes and, 281
creating and destroying, 5–33
creation and performance, 6, 22–23
deserialization filtering of, 342
effectively immutable, 316
eliminating obsolete references to, 26–28, 60, 308
expense of creating, 24
favor referring to by interfaces, 280–281
function, 114
immutable (see immutable objects)
in inconsistent states, 11–12, 96, 309
(see also corrupted objects)
methods common to all, 37–72
nulling out obsolete references to, 27
process, 114
reuse, 22–25
safe publication of, 316
string representations of, 55–57
when to refer to by class, 281
Observer pattern, 317
obsolete object references, 26–28, 60, 308
open calls, 321
optimizations, 286–288
  caching hash codes, 53
  lazy initialization, 333–335
  notify instead of notifyAll, 329
  object reuse, 22–25
  order of comparisons in equals, 47
  parallelizing streams, 224
  static initialization, 23
  StringBuffer and, 279
  using == in equals, 46
  optionals, 249
  exceptions and, 295, 298
  as return values, 249–253
  ordinals
    vs. enum maps, 171–175
    vs. instance fields, 168
  overloading
    See method overloading
  Override annotations, 49, 188–190
  overriding
    See method overriding

P
  package-private access level, 4, 74, 84
  packages, naming conventions for, 289–290
  parallelizing streams, 222–225
  parameter lists
    of builders, 14
    shortening, 236–237
    varargs and, 245–246
  parameterized types, 117–122
    reifiable, 127
  parameterless constructors, 19
  parameters
    defensive copies of mutable, 232
    type (see type parameters)
    validity checking of, 227–230, 353–355
  PECS mnemonic, 141
  performance, 286–288
    autoboxing and, 24, 201, 275
    BigDecimal and, 271
    builder pattern, 16
    cleaners, 29–30
    defensive copying and, 234
    of enums, 167, 170
    of equals, 46–47
    of excessive synchronization, 321
  finalizers, 29–30
    for-each loops and, 264
    of hashCode, 50, 53
    immutable classes and, 83–85
    libraries and, 268
    measuring, 287
    memory leaks and, 27
    native methods and, 285
    object creation and, 6, 22–23
    of reflection, 282
    parallelizing streams and, 222–225
    of serialization, 348–350
    software architecture and, 286–287
    state-testing vs. special return value, 295
    static factories and, 6
    of string concatenation, 279
    toString and, 57
    varargs and, 246
    wrapper classes and, 91
    See also optimizations
  performance model, 288
  portability
    cleaners and, 29
    finalizers and, 29
    native methods and, 285
    thread priorities and, 337
    thread scheduler and, 336
    predicates, 104
    primitive fields
      compareTo and, 69
      equals and, 47
    primitive types, 273
    incompatibility with generic types, 134
    optionals and, 253
    prefer over boxed primitives, 24, 273–275
    See also individual primitive types
  private access level, 74
  private constructors, 84
    for noninstantiability, 19
    for singletons, 17–18
  private lock object idiom, 332
  private lock objects, 332
  procedural programming, 82
  process objects, 114
  producer-consumer queues, 326
  programming principles, 2
  promptness of finalization, 29
protected access level, 74–75
public access level, 74
public fields vs. accessor methods, 78–79
publicly accessible locks, 331

Q
qualified this construct, 112

R
racy single check idiom, 335
range checking, 229
raw types, 117–122
readObject method, 353–358
defensive copies and, 80
how to write, 358
incompatible with instance-controlled objects, 359
overridable methods and, 96, 358
readResolve method
access levels of, 97
choosing access levels of, 362
prefer enum types to, 359–362
using for instance-controlled classes, 359
recipes
adding behaviors to individual enum constants, 162
adding data to enum types, 160
builder pattern, 12
checking significant fields in equals, 47
clone, 64
cомpareTo, 68
eliminating self-use, 98
equals, 46
generifying a class, 130–133
hashCode, 51
implementing generics atop arrays, 131–133
method chaining, 14
noninstantiable classes, 19
readObject, 358
serialization proxies, 363–364
serialized singletons, 18
singletons as single-element enums, 18
singletons with private constructors, 17
skeletal implementations, 102
tagged classes to class hierarchies, 110–111
See also rules
recursive type bounds, 137
recursive type parameters, 14
reduction strategy, 211
reductions, 223
reentrant locks, 320
reference types, 3, 273
reflection, 282–284
AccessibleObject.setAccessible attacks, 17
clone and, 58
drawbacks of, 282
reflective interface instantiation, 283
uses for, 282, 284
reflexivity requirements
compareTo, 68
equals, 38–39
refined types, 126
resource factories, 21
resource-ordering deadlocks, 351
resources
locked, and finalizers, 30
releasing, 31
restricted marker interfaces, 191
return classes, varied
serialization proxy pattern and, 365
static factory methods and, 7–8
return statements, SuppressWarnings annotation and, 124
return types
bounded wildcard types as, 142
collections vs. streams, 216–221
static factory methods and, 6
reusable forwarding classes, 89–91
rules
accessibility, 74–75
appropriateness of checked exceptions, 298
choosing bounded wildcard types, 141
choosing exception types, 296–297
decreasing serialization dangers, 341–342
for immutable objects, 80
mapping domains to package names, 289
marker interfaces vs. annotations, 192
optimization, 286
for performance of parallel streams, 223
replacing type parameters with wildcards, 144
for SafeVarargs annotations, 149
static members accessibility, 112
writing doc comments, 254–260
runtime exceptions
  See unchecked exceptions

S
  safe array accesses, 76
  safe languages, 231
  safe publication, 316
  safety failures, 315
  parallel streams and, 224
  wait and, 328
  SafeVarargs annotations, 147
  legal uses of, 149
  scope
    local variables, 261–263
    SuppressWarnings annotations, 124
      of variables, obsolete references and, 27
    security, defensive copying for, 25, 231
    security issues
      accessible nonzero-length arrays, 76
      AccessibleObject.setAccessible
        attacks, 17
      denial of service attacks, 331
      deserialization bombs, 339–340
      ElvisStealer attacks, 362
      finalizer attacks, 30–31
      gadgets, 340
      internal field theft attacks, 355–357
      ransomware attacks, 339
      reflection, 17
      remote code execution, 340
      rogue object reference attacks, 355–357
      serialization, 339, 344, 353, 360
      stealer attacks, 362
      strings as keys for granting data access, 277
      subclassing and, 89
      time-of-check/time-of-use (TOCTOU)
        attacks, 233
  SELF problem, 91
  self-use
    documenting, for inheritance, 93
    eliminating, for inheritance, 98
    serial version UIDs, 343, 351–352
  Serializable, 343–345
  serialization, 339–366
    anonymous classes and, 196
    costs of, 343
    decreasing the dangers of, 341–342
    designing for inheritance and, 96–97
    documenting for, 347, 350
    effect on exported APIs, 343
    flexible return classes for, 365
    immutability and, 85, 353
    internal field theft attacks and, 360–362
    lambdas and, 196
    object deserialization filtering, 342
    prefer alternatives to, 339–342
    singletons and, 18
    synchronization for, 351
    transient fields for, 348
    validity checking in, 357
    when to use, 345
    serialization proxy pattern, 363–366
    serialized forms, as part of exported APIs, 343
    serialized instances vs. serialization proxy
      pattern, 363–366
  service provider frameworks, 8
  short-circuiting operations, 223
  signatures of methods, 3, 236–237
  signum function, 67
  simple implementations, 103
  simulated multiple inheritance, 102
  simulated self-type idiom, 14
  single-check idiom, 335
  singletons, 17–18
    vs. dependency injection, 20
  skeletal implementations, 100–101
  source files, 115–116
  space consumption
    enum types, 175
    immutable objects and, 83
    memory leaks and, 27
    nonstatic member classes and, 113
  spliterator, 223
  spurious wake-ups, 329
  state-dependent modify operations, 325
  state-testing methods, 294–295, 299
static factory methods, 5
advantages of, 5–8
anonymous classes within, 114
in API documentation, 8
vs. cloning, 65
copy and conversion factories, 65
flexibility in returned classes, 7–8
for generic singletons, 18, 136
immutable objects and, 22, 82, 84
instance-controlled classes and, 6
limitations of, 8–9
naming conventions for, 9, 292
replacing constructors with, 5–9, 22, 240
return types of, 6–8
for service provider frameworks, 8
for singletons, 17
subcassing and, 8
static fields
for defining constants, 290
lazy initialization of, 334
synchronization of mutable, 322
static import facility, 108
static imports, 70
static member classes, 112
cleaners and, 33
common uses of, 112–113
for enum types, 161
vs. nonstatic, 112, 114
for representing aggregates, 276
for shortening parameter lists, 237
static members, accessibility in interfaces, 7
storage pools, 28
strategy enum pattern, 166
Strategy pattern, 193
stream pipelines, 203
side-effect free, 210–215
stream unique identifiers
See serial version UIDs
streams, 193, 203–225
char values and, 206
collectors for, 211–215
for functional programming, 210–215
vs. maps, behavior of, 173
parallelizing, 222–225
preserving order from parallel, 224
as return types, vs. collections, 216–221
specifying collectors for, 173, 214
strengths of, 207
vs. threads, 323–324
using, 203–209
See also collectors
String constants vs. enum types, 158
string representations, 55–57, 306
strings
concatenating, 279
as substitutes for other types, 276–278
subcassing, 3, 87
abstract classes, and equals, 45
access levels and, 75
appropriateness of, 92
Cloneable and, 96
compareTo and, 68
equals and, 40, 42
finalizer attacks and, 31
fragility, 89
invariant corruption and, 92
method access levels and, 75
prohibiting, 8, 18–19, 85, 97
serialization and, 344
skeletal implementations, 102
static factory methods and, 8
as test of design for inheritance, 95
See also inheritance
subtype relations, 134, 140
summary descriptions in Javadoc, 257
supertype relations, 141
SuppressWarnings annotation, 123–125
switch statements, and enum types, 164, 167
symmetry requirements
compareTo, 68
equals, 38–39
synchronization
of atomic data, 312–314
excessive, 317–322
and performance, 321
ramifications of, 317
internal, 322
for mutual exclusion, 311
serialization and, 351
for shared mutable data, 311–316
techniques for, 314–316
for thread communication, 312–314
synchronized regions
  alien methods and, 317–321
  minimizing work in, 321
synchronizers, 326
synthetic annotations, 186

T
  tag fields, 109
tardy finalization, 29
tasks, 324
  vs. threads, 323–324
telescoping constructor pattern, 10–11
Template Method pattern, 101, 199
this, in doc comments, 256
this, in lambdas vs. anonymous classes, 196
thread pools, 323
  sizing of, 336
thread priorities, 337
thread safety
  documenting, 322, 330–332
  immutable objects and, 82
  levels of, 330–331
  mutability and, 75, 322
thread schedulers, 336–337
thread starvation deadlocks, 328
Thread.yield method, avoiding, 337
threads, busy-waiting, 336
throwables, types of, 296
time-of-check/time-of-use attacks, 233
TOCTOU attacks, 233
toString method, 55–57
  enum types and, 160
  general contract for, 55
  when to override, 57
transient fields, 348–351
  with readResolve, 360
  when to use, 351
transitivity requirements
compareTo, 68
equals, 38, 40–45
try-finally
  prefer try-with-resources to, 34
try-with-resources
  prefer to try-finally, 34–36
type bounds, recursive, 137
type inference, 70, 123, 142, 194
type parameter lists, 135
type parameters, 117, 135
  bounded, 134, 154
  naming conventions for, 135, 290
  recursively bound, 14, 137
  vs. wildcards, 144
  type safety
    dynamic casts and, 154
    from enum types, 158
    heap pollution and, 146
    parameterized types and, 119
    raw types and, 119
  type tokens, 151
types
  conversion of, 65, 291
  generic, 117, 130–134
  non-reifiable, 127, 131
  parameterized, 117–122
  primitive, 273
  raw, 117
  reference, 273
  See also bounded wildcard types; unbounded wildcard types
typesafe heterogeneous container pattern, 151–155
  incompatibility with nonreifiable types, 154
U
unbounded type parameters
  vs. bounded wildcard types, 144
unbounded wildcard types, 120
  vs. bounded wildcard types, 121
  nested, 152
  vs. raw types, 121
  reifiable, 127
  vs. unbounded type parameters, 144
unchecked exceptions
  vs. checked exceptions, 296–297
  compatibility and, 305
  excluding from method declarations, 304
  purpose of, 296
unchecked warnings, 123–125
  of casts, 127, 129, 137
underscores, in numeric literals, 108
unintentional object retentions
  See memory leaks
unintentionally instantiable classes, 19
users of APIs, 4
utility classes, 19
vs. constant interfaces, 108
vs. dependency injection, 20

V
validity checking
builders and, 14
constructor parameters, 229
defensive copying and, 232
doctor serialized objects, 355
explicit, 230
parameters, 227–230
failure atomicity and, 308
readObject parameters, 353–355
value classes, 38
toString and, 56
value types vs. strings, 276
varargs, 245–246
builders and, 16
with generics, 146–150
generics, and compiler warnings, 127
performance, 246
variable arity methods, 245
variable return classes
serialization proxy pattern and, 365
static factory methods and, 7–8
variables
atomic operations on, 311
local (see local variables)
naming conventions for, 290
scope of, and obsolete references, 27
to avoid subclassing, 44, 68
to maintain invariants, 234
naming conventions for, 291
object reuse and, 23
volatile modifier, 314–315

W
wait loop idiom, 328–329
warnings, unchecked
See unchecked warnings
weak references, 28
while loops vs. for loops, 262
wildcard types
capturing, 145
vs. type parameters, 144
See also bounded wildcard types; unbounded wildcard types
window of vulnerability, 233
work queues, 326
wrapper class idiom, 100
wrapper classes, 89–91
defensive copying and, 235
incompatible with callback frameworks, 91
vs. subclassing, 97
writeReplace method, access levels of, 97

Z
zero-length arrays, immutability of, 248