Joshua Bloch

Updated for Java 9

Effective Java

Third Edition

1. Introduction

Used to help developers make effective use of the Java programming language and its fundamental
libraries: java.lang, java.util, and java.io, and sub-packages such as java.util.concurrent and java.util.function.

The items are loosely grouped into eleven chapters, each covering one broad aspect of software design.

Lambdas , Streams , Optionals , Default methods in interfaces , try-with-resources , @SafeVarargs , Modules are specified at this book.

For the most part, this book is not about performance. It is about writing programs that are clear, correct, usable, robust, flexible, and maintainable.

For the most part, this book uses technical terms as they are defined in *The Java Language Specification, Java SE 8 Edition* [JLS].

A few terms deserve special mention. The language supports four kinds of
types: interfaces (including annotations), classes (including enums), arrays, and primitives. The first three are known as reference types. Class instances and arrays are objects; primitive values are not. A class's members consist of its fields, methods, member classes, and member interfaces.

A method's signature consists of its name and the types of its formal parameters; the signature does not include the method's return type.

The term exported API, or simply API, refers to the classes, interfaces, constructors, members, and serialised forms by which a programmer accesses a class, interface, or package.

A programmer who writes a program that uses an API is referred to as a user of the API. A class whose implementation uses an API is a client of the API.

A class whose implementation uses an API is a client of the API.

Classes, interfaces, constructors, members, and serialised forms are collectively known as API elements.

An exported API consists of the API elements that are accessible outside of the package that defines the API. These are the API elements that any client can use and the author of the API commits to support.

Loosely speaking, the exported API of a package consists of the public and protected members and constructors of every public class or interface in the package.

In Java 9, a *module system* was added to the platform. If a library makes use of the module system, its exported API is the union of the exported APIs of all the packages exported by the library's module declaration.

2. Creating and Destroying Objects

ITEM 1: CONSIDER STATIC FACTORY METHODS INSTEAD OF CONSTRUCTORS

(https://www.geeksforgeeks.org/difference-between-constructor-and-static-factory-method-in-java/)

```
public final class GFG {
    public static GFG valueOf(float real, float imaginary)
    {
        return new GFG(real, imaginary);
    }
}
```

The traditional way for a class to allow a client to obtain an instance is to provide a public constructor. There is another technique that should be a part of every programmer’s toolkit. A class can provide a public static factory method, which is simply a static method that returns an instance of the class. Here’s a simple example from Boolean (the boxed primitive class for boolean). This method translates a boolean primitive value into a Boolean object reference:

```
public static Boolean valueOf(boolean b) {
 return b ? Boolean.TRUE : Boolean.FALSE;
}
```

A class can provide its clients with static factory methods instead of, or in addition to, public constructors.

One advantage of static factory methods is that, unlike constructors, they have names.

For example,
the constructor BigInteger(int, int, Random), which returns
a BigInteger that is probably prime, would have been better expressed as a static factory method
named BigInteger.probablePrime.

A second advantage of static factory methods is that, unlike constructors, they are not required to create a new object each time they're invoked.

The Boolean.valueOf(boolean)method illustrates this technique:
it *never* creates an object. This technique is similar to
the *Flyweight*pattern [Gamma95]. It can greatly improve performance if equivalent objects are requested often, especially if they are expensive to create.

A third advantage of static factory methods is that, unlike constructors, they can return an object of any subtype of their return type

Prior to Java 8, interfaces couldn't have static methods. By convention, static factory methods for an interface
named *Type* were put in a *noninstantiable companion class* (Item 4) named *Types*.

For example, the Java Collections Framework has forty-five utility implementations of its interfaces, providing unmodifiable collections, synchronized collections, and the like. Nearly all of these implementations are exported via static factory methods in one noninstantiable class (java.util.Collections).

A fourth advantage of static factories is that the class of the returned object can vary from call to call as a function of the input parameters.

A fifth advantage of static factories is that the class of the returned object need not exist when the class containing the method is written.

The main limitation of providing only static factory methods is that classes without public or protected constructors cannot be subclassed.

A second shortcoming of static factory methods is that they are hard for programmers to find.

Here are some common names for static factory methods.

• from—A type-conversion method that takes a single parameter and returns a corresponding instance of this type, for example:
Date d = Date.from(instant);

• of—An aggregation method that takes multiple parameters and returns an instance of this type that incorporates them, for example:

```
Set<Rank> faceCards = EnumSet.of(JACK, QUEEN, KING);
```

• valueOf—A more verbose alternative to from and of, for example:

```
BigInteger prime = BigInteger.valueOf(Integer.MAX_VALUE);
```

• instance or getInstance—Returns an instance that is described by its parameters (if any) but cannot be said to have the same value, for example:

```
StackWalker luke = StackWalker.getInstance(options);
```

• create or newInstance—Like instance or getInstance, except that the method guarantees that each call returns a new instance, for example:

```
Object newArray = Array.newInstance(classObject, arrayLen);
```

• getType—Like getInstance, but used if the factory method is in a different class. Type is the type of object returned by the factory method, for example:

```
FileStore fs = Files.getFileStore(path);
```

• newType—Like newInstance, but used if the factory method is in a different class. Type is the type of object returned by the factory method, for example:

BufferedReader br = Files.newBufferedReader(path);

type—A concise alternative to getType and newType, for example:

List<Complaint> litany = Collections.list(legacyLitany);

ITEM 2: CONSIDER A BUILDER WHEN FACED WITH MANY CONSTRUCTOR PARAMETERS

Static factories and constructors share a limitation: they do not scale well to large numbers of optional parameters

NutritionFacts cocaCola =
new NutritionFacts(240, 8, 100, 0, 35, 27);

Typically this constructor invocation will require many parameters that you don’t want to set, but you’re forced to pass a value for them anyway.

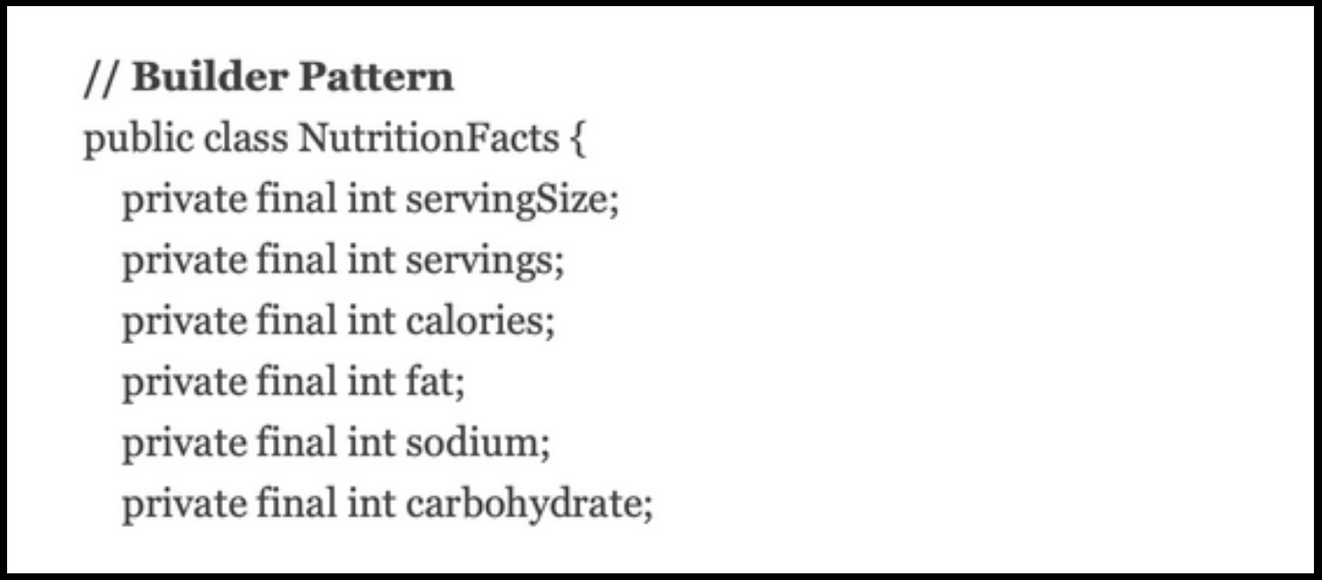
In short, the telescoping constructor pattern works, but it is hard to write client code when there are many parameters, and harder still to read it.

JavaBeans Pattern

```
public class NutritionFacts {
private int servingSize = -1; // Required; no default value private int
private int calories = 0;
public void setServingSize(int val) { servingSize = val; }
public void setCalories(int val) { calories = val; }
}
```

```
NutritionFacts cocaCola = new NutritionFacts();
cocaCola.setServingSize(240); cocaCola.setServings(8);
cocaCola.setCalories(100); cocaCola.setSodium(35);
cocaCola.setCarbohydrate(27);
```

Unfortunately, the JavaBeans pattern has serious disadvantages of its own. Because construction is split across multiple calls, a JavaBean may be in an inconsistent state partway through its construction.



```
// Builder Pattern
public class NutritionFacts {
   private final int servingSize;
   private final int servings;
   private final int calories;
   private final int fat;
   private final int sodium;
   private final int carbohydrate;
```

```
        public static class Builder {
            // Required parameters
            private final int servingSize;
            private final int servings;

            // Optional parameters - initialized to default values
            private int calories      = 0;
            private int fat           = 0;
            private int sodium        = 0;
            private int carbohydrate  = 0;

            public Builder(int servingSize, int servings) {
                this.servingSize = servingSize;
                this.servings    = servings;

    }

    public Builder calories(int val)
        { calories = val;      return this; }
    public Builder fat(int val)
        { fat = val;           return this; }
    public Builder sodium(int val)
        { sodium = val;        return this; }
    public Builder carbohydrate(int val)
        { carbohydrate = val;  return this; }

    public NutritionFacts build() {
        return new NutritionFacts(this);
    }
}
```

```
    private NutritionFacts(Builder builder) {
        servingSize  = builder.servingSize;
        servings     = builder.servings;
        calories     = builder.calories;
        fat          = builder.fat;
        sodium       = builder.sodium;
        carbohydrate = builder.carbohydrate;
    }
}
```

NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8)
.calories(100).sodium(35).carbohydrate(27).build();

The Builder pattern simulates named optional parameters as found in Python and Scala.

The Builder pattern is well suited to class hierarchies.

the Builder pattern is a good choice when designing classes whose constructors or static factories would have more than a handful of parameter

```
// Builder pattern for class hierarchies
public abstract class Pizza {
  public enum Topping { HAM, MUSHROOM, ONION, PEPPER,
SAUSAGE }
  final Set<Topping> toppings;

  abstract static class Builder<T extends Builder<T>> {
    EnumSet<Topping> toppings =
EnumSet.noneOf(Topping.class);
    public T addTopping(Topping topping) {
      toppings.add(Objects.requireNonNull(topping));
```

```
      return self();
    }

    abstract Pizza build();

    // Subclasses must override this method to return
"this"
    protected abstract T self();
  }
  Pizza(Builder<?> builder) {
    toppings = builder.toppings.clone(); // See Item  50
  }
}
```

```
public class NyPizza extends Pizza {
  public enum Size { SMALL, MEDIUM, LARGE }
  private final Size size;

  public static class Builder extends Pizza.Builder<Builder> {
    private final Size size;

    public Builder(Size size) {
      this.size = Objects.requireNonNull(size);
    }

    @Override public NyPizza build() {
      return new NyPizza(this);
    }

    @Override protected Builder self() { return this; }
  }

  private NyPizza(Builder builder) {
    super(builder);
    size = builder.size;
  }
}
```

```
public class Calzone extends Pizza {
  private final boolean sauceInside;

  public static class Builder extends Pizza.Builder<Builder> {
    private boolean sauceInside = false; // Default

    public Builder sauceInside() {
      sauceInside = true;
      return this;
    }

    @Override public Calzone build() {
      return new Calzone(this);
    }

    @Override protected Builder self() { return this; }
  }
```

```
  private Calzone(Builder builder) {
    super(builder);
    sauceInside = builder.sauceInside;
  }
}
```

ITEM 3: ENFORCE THE SINGLETON PROPERTY WITH A PRIVATE CONSTRUCTOR OR AN ENUM TYPE

A singleton is simply a class that is instantiated exactly once

Making a class a singleton can make it difficult to test its clients

```
// Singleton with public final field
public class Elvis {
  public static final Elvis INSTANCE = new Elvis();
  private Elvis() { ... }

  public void leaveTheBuilding() { ... }
}
```

```
// Singleton with static factory
public class Elvis {
  private static final Elvis INSTANCE = new Elvis();
  private Elvis() { ... }
  public static Elvis getInstance() { return INSTANCE; }

  public void leaveTheBuilding() { ... }
}
```

Otherwise, each time a serialized instance is deserialized, a new instance will be created, leading, in the case of our example, to spurious Elvis sightings. To
prevent this from happening, add this readResolvemethod to the Elvis class:

```
// readResolve method to preserve singleton property
private Object readResolve() {
    // Return the one true Elvis and let the garbage collector
    // take care of the Elvis impersonator.
   return INSTANCE;
}
```

A third way to implement a singleton is to declare a single-element enum:

Click here to view code image

```
// Enum singleton - the preferred approach
public enum Elvis {
   INSTANCE;

   public void leaveTheBuilding() { ... }
}
```

a single-element enum type is often the best way to implement a singleton

ITEM 4: ENFORCE NONINSTANTIABILITY WITH A PRIVATE CONSTRUCTOR

```
public class UtilityClass {
// Suppress default constructor for noninstantiability
private UtilityClass() { throw new AssertionError(); }
.. // Remainder omitted }
```

The AssertionErrorisn’t strictly required, but it provides insurance in case the constructor is accidentally invoked from within the class.

ITEM 5: PREFER DEPENDENCY INJECTION TO HARDWIRING RESOURCES

public class SpellChecker {
private static final Lexicon dictionary = ...; private SpellChecker() {} // Noninstantiable
public static boolean isValid(String word) { ... }
public static List<String> suggestions(String typo) { ... } }

Click here to view code image

```
// Inappropriate use of singleton - inflexible & untestable!
public class SpellChecker {
   private final Lexicon dictionary = ...;

   private SpellChecker(...) {}
   public static INSTANCE = new SpellChecker(...);

   public boolean isValid(String word) { ... }
   public List<String> suggestions(String typo) { ... }
}
```

Neither of these approaches is satisfactory, because they assume that there is only one dictionary worth using. In practice, each language has its own dictionary, and special dictionaries are used

Static utility classes and singletons are inappropriate for classes whose behavior is parameterized by an underlying resource.

In summary, do not use a singleton or static utility class to implement a class that depends on one or more underlying
resources whose behavior affects that of the class, and do not have the class create these resources directly. Instead, pass the resources, or factories to create them, into the constructor (or static factory or builder). This practice, known as dependency injection, will greatly enhance the flexibility, reusability, and testability of a class.

```
class Employee{
Address address;
// Address add = new Address(); X
Employee(Address address){
this.address=address;
}
public void setAddress(Address address){
this.address=address;
}
}
```

ITEM 6: AVOID CREATING UNNECESSARY OBJECTS

It is often appropriate to reuse a single object instead of creating a new functionally equivalent object each time it is needed. Reuse can be both faster and more stylish. An object can always be reused if it is immutable

String s = new String("bikini"); // DON'T DO THIS!

The statement creates a new String instance each time it is executed, and none of those object creations is necessary

The improved version is simply the following:
String s = "bikini";

```
// Performance can be greatly improved!
static boolean isRomanNumeral(String s) {
    return s.matches("^(?=.)M*(C[MD]|D?C{0,3})"
            + "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");
}
```

The problem with this implementation is that it relies on the `String.matches` method. **While`String.matches` is the easiest way to check if a string matches a regular expression, it's not suitable for repeated use in performance-critical situations.** The problem is that it internally creates a `Pattern` instance for the regular expression and uses it only once,

after which it becomes eligible for garbage collection. Creating

```
// Reusing expensive object for improved performance
public class RomanNumerals {
    private static final Pattern ROMAN = Pattern.compile(
            "^(?=.)M*(C[MD]|D?C{0,3})"
            + "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");

    static boolean isRomanNumeral(String s) {
        return ROMAN.matcher(s).matches();
    }
}
```

ITEM 7: ELIMINATE OBSOLETE OBJECT REFERENCES

```
public Object pop() { if (size == 0)
throw new EmptyStackException(); return elements[--size];
}
```

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

ITEM 8: AVOID FINALIZERS AND CLEANERS

Finalizers are unpredictable, often dangerous, and generally unnecessary.

Cleaners are less dangerous than finalizers, but still unpredictable, slow, and generally unnecessary.

```
// An autocloseable class using a cleaner as a safety net
public class Room implements AutoCloseable {
  private static final Cleaner cleaner = Cleaner.create();

  // Resource that requires cleaning. Must not refer to Room!
  private static class State implements Runnable {
    int numJunkPiles; // Number of junk piles in this room

    State(int numJunkPiles) {
      this.numJunkPiles = numJunkPiles;
    }

    // Invoked by close method or cleaner
    @Override public void run() {
      System.out.println("Cleaning room");
      numJunkPiles = 0;
    }
  }

  // The state of this room, shared with our cleanable
  private final State state;

  // Our cleanable. Cleans the room when it's eligible for gc
  private final Cleaner.Cleanable cleanable;
```

```
  public Room(int numJunkPiles) {
    state = new State(numJunkPiles);
    cleanable = cleaner.register(this, state);
  }

  @Override public void close() {
    cleanable.clean();
  }
}
```

No guarantees are made relating to whether cleaning actions are invoked or not

```
public class Adult {
  public static void main(String[] args) {
    try (Room myRoom = new Room(7)) {
      System.out.println("Goodbye");
    }
  }
}
```

As you'd expect, running the `Adult` program prints `Goodbye`, followed by `Cleaning room`. But what about this ill-behaved program, which never cleans its room?

Click here to view code image

```
public class Teenager {
  public static void main(String[] args) {
    new Room(99);
    System.out.println("Peace out");
  }
}
```

ITEM 9: PREFER TRY-WITH-RESOURCES TO TRY-FINALLY

```
// try-finally - No longer the best way to close resources!
static String firstLineOfFile(String path) throws IOException {
  BufferedReader br = new BufferedReader(new
FileReader(path));
  try {
    return br.readLine();
  } finally {
    br.close();
  }
}
```

```
// try-finally is ugly when used with more than one
resource!
static void copy(String src, String dst) throws IOException {
  InputStream in = new FileInputStream(src);
  try {
    OutputStream out = new FileOutputStream(dst);
    try {
      byte[] buf = new byte[BUFFER_SIZE];
      int n;
      while ((n = in.read(buf)) >= 0)
        out.write(buf, 0, n);
    } finally {
      out.close();
```

```
    }
  } finally {
    in.close();
  }
}
```

Click here to view code image

```
// try-with-resources - the the best way to close resources!
static String firstLineOfFile(String path) throws IOException {
    try (BufferedReader br = new BufferedReader(
            new FileReader(path))) {
        return br.readLine();
    }
}
```

And here's how our second example looks using `try`-with-resources:

Click here to view code image

```
// try-with-resources on multiple resources - short and sweet
static void copy(String src, String dst) throws IOException {
    try (InputStream  in = new FileInputStream(src);
         OutputStream out = new FileOutputStream(dst)) {
        byte[] buf = new byte[BUFFER_SIZE];
        int n;
        while ((n = in.read(buf)) >= 0)
            out.write(buf, 0, n);
    }
}
```

```
// try-with-resources with a catch clause
static String firstLineOfFile(String path, String defaultVal) {
    try (BufferedReader br = new BufferedReader(
            new FileReader(path))) {
        return br.readLine();
    } catch (IOException e) {
        return defaultVal;
    }
}
```

The lesson is clear: Always use `try`-with-resources in preference to `try-finally` when working with resources that must be closed.

Chapter 3. Methods Common to All Objects

ALTHOUGH Object is a concrete class, it is designed primarily for extension. All of its nonfinal methods
(equals, hashCode, toString, clone, and finalize) have explicit general contracts because they are designed to be overridden.

ITEM 10: OBEY THE GENERAL CONTRACT WHEN OVERRIDING EQUALS

Each instance of the class is inherently unique.

There is no need for the class to provide a “logical equality” test. For example, java.util.regex.Pattern could have overridden equals to check whether two Pattern instances represented exactly the same regular expression, but the designers didn’t think that clients would need or want this functionality. Under these circumstances, the equals implementation inherited from Object is ideal.

A superclass has already overridden equals, and the superclass behavior is appropriate for this class.

The class is private or package-private, and you are certain that its equals method will never be invoked.

@Override public boolean equals(Object o) {
throw new AssertionError(); // Method is never called
}

The equals method implements an equivalence relation

```
public class Point {
   private final int x;
   private final int y;

   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }

   @Override public boolean equals(Object o) {
      if (!(o instanceof Point))
         return false;
      Point p = (Point)o;
      return p.x == x && p.y == y;
   }

   ... // Remainder omitted
}
```

```
public class CounterPoint extends Point {
   private static final AtomicInteger counter =
         new AtomicInteger();

   public CounterPoint(int x, int y) {
      super(x, y);
      counter.incrementAndGet();
   }
   public static int numberCreated() { return counter.get(); }
}
```

The Liskov substitution principle says that any important property of a type should also hold for all its subtypes so that any method written for the type should work equally well on its subtypes

ITEM 11: ALWAYS OVERRIDE HASHCODE WHEN YOU OVERRIDE EQUALS

```
// hashCode method with lazily initialized cached hash
code
private int hashCode; // Automatically initialized to 0

@Override public int hashCode() {
   int result = hashCode;
   if (result == 0) {
      result = Short.hashCode(areaCode);
      result = 31 * result + Short.hashCode(prefix);
      result = 31 * result + Short.hashCode(lineNum);
      hashCode = result;
   }
   return result;
}
```

ITEM 12: ALWAYS OVERRIDE TOSTRING

```
@Override public String toString() {
   return String.format("%03d-%03d-%04d",
         areaCode, prefix, lineNum);
}
```

ITEM 13: OVERRIDE CLONE JUDICIOUSLY

The Cloneable interface was intended as a mixin interface (Item 20) for classes to advertise that they permit cloning.

Click here to view code image

x.clone().getClass() == x.getClass()

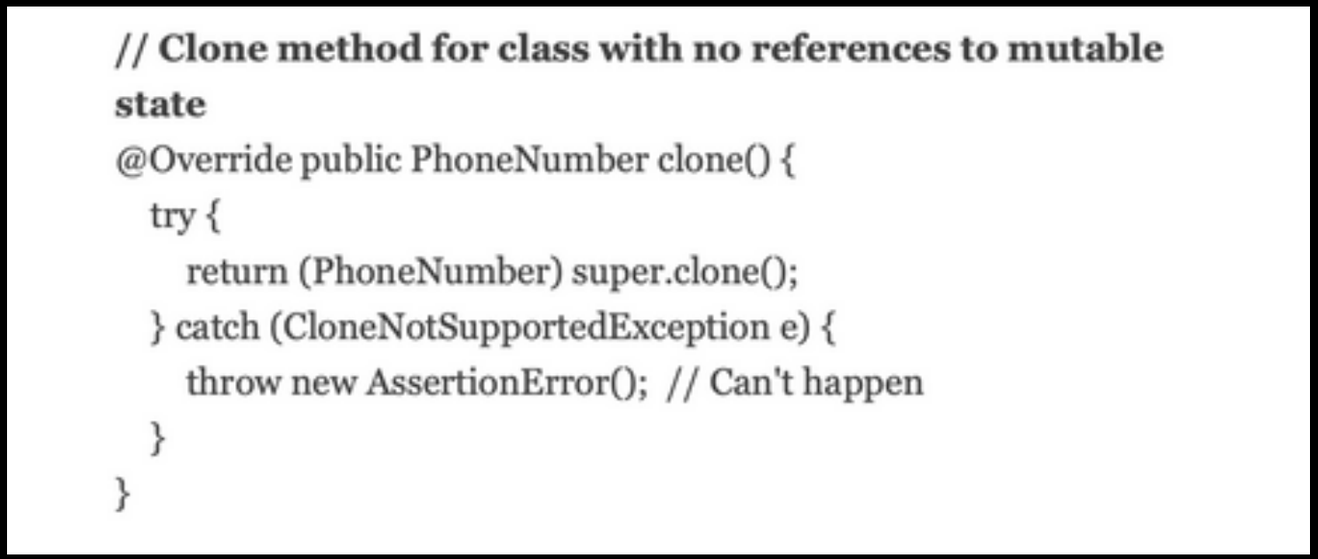
will be `true`, but these are not absolute requirements. While it is typically the case that

x.clone().equals(x)

will be `true`, this is not an absolute requirement.
By convention, the object returned by this method should be obtained by calling `super.clone`. If a class and all of its superclasses (except `Object`) obey this convention, it will be the case that

Click here to view code image

x.clone().getClass() == x.getClass().



```
// Clone method for class with no references to mutable
state
@Override public PhoneNumber clone() {
  try {
    return (PhoneNumber) super.clone();
  } catch (CloneNotSupportedException e) {
    throw new AssertionError();  // Can't happen
  }
}
```

// Iteratively copy the linked list headed by this Entry
Entry deepCopy() {
Entry result = new Entry(key, value, next);
for (Entry p = result; p.next != null; p = p.next)
p.next = new Entry(p.next.key, p.next.value, p.next.next); return result;
}

// clone method for extendable class not supporting Cloneable
@Override
protected final Object clone() throws CloneNotSupportedException {
throw new CloneNotSupportedException(); }
There is one

ITEM 14: CONSIDER IMPLEMENTING COMPARABLE

```
public class WordList {
   public static void main(String[] args) {
      Set<String> s = new TreeSet<>();
      Collections.addAll(s, args);
      System.out.println(s);
   }
}
```

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

```
// Single-field Comparable with object reference field
public final class CaseInsensitiveString
      implements Comparable<CaseInsensitiveString> {
   public int compareTo(CaseInsensitiveString cis) {
      return String.CASE_INSENSITIVE_ORDER.compare(s, cis.s);
   }
   ... // Remainder omitted
}
```

Use of the relational
operators < and > in compareTo methods is verbose and error-prone and no longer recommended.

```
private static final Comparator<PhoneNumber> COMPARATOR =
        comparingInt((PhoneNumber pn) -> pn.areaCode)
         .thenComparingInt(pn -> pn.prefix)
```

```
         .thenComparingInt(pn -> pn.lineNum);

public int compareTo(PhoneNumber pn) {
   return COMPARATOR.compare(this, pn);
}
```

// Comparator based on static compare method

```
static Comparator<Object> hashCodeOrder = new Comparator<>()
{
   public int compare(Object o1, Object o2) {
      return Integer.compare(o1.hashCode(), o2.hashCode());
   }
};
```

```
static Comparator<Object> hashCodeOrder =
      Comparator.comparingInt(o -> o.hashCode());
```

Chapter 4. Classes and Interfaces

This chapter contains guidelines to help you make the best use of these elements so that your classes and interfaces are usable, robust, and flexible.

ITEM 15: MINIMIZE THE ACCESSIBILITY OF CLASSES AND MEMBERS

- **private**—The member is accessible only from the top-level class where it is declared.
- **package-private**—The member is accessible from any class in the package where it is declared. Technically known as *default* access, this is the access level you get if no access modifier is specified (except for interface members, which are public by default).
- **protected**—The member is accessible from subclasses of the class where it is declared (subject to a few restrictions [JLS, 6.6.2]) and from any class in the package where it is declared.
- **public**—The member is accessible from anywhere.

it is wrong for a class to have a public static final array field, or an accessor that returns such a field.

```
// Potential security hole!
public static final Thing[] VALUES = { ... };
```

```
private static final Thing[] PRIVATE_VALUES = { ... };
public static final List<Thing> VALUES =
  Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES))
;
```

Click here to view code image

```
private static final Thing[] PRIVATE_VALUES = { ... };
public static final Thing[] values() {
   return PRIVATE_VALUES.clone();
}
```

As of Java 9, there are two additional, implicit access levels introduced as part of the module system. A module is a grouping of packages, like a package is a grouping of classes. A module may explicitly export some of its packages via export declarations in
its module declaration (which is by convention contained in a source file named module-info.java). Public and protected members of unexported packages in a module are inaccessible outside the module; within the module, accessibility is unaffected by export declarations. Using the module system allows you to share classes among packages within a module without making them visible to the entire world.

Ensure that objects referenced by public static final fields are immutable.

ITEM 16: IN PUBLIC CLASSES, USE ACCESSOR METHODS, NOT PUBLIC FIELDS

```
// Encapsulation of data by accessor methods and
mutators
class Point {
   private double x;
   private double y;

   public Point(double x, double y) {
      this.x = x;
      this.y = y;
   }

   public double getX() { return x; }
   public double getY() { return y; }

   public void setX(double x) { this.x = x; }
   public void setY(double y) { this.y = y; }
}
```

if a class is accessible outside its package, provide accessor methods

if a class is package-private or is a private nested class, there is nothing inherently wrong with exposing its data fields

```
// Public class with exposed immutable fields - questionable
public final class Time {
   private static final int HOURS_PER_DAY    = 24;
   private static final int MINUTES_PER_HOUR = 60;

   public final int hour;
   public final int minute;

   public Time(int hour, int minute) {
      if (hour < 0 || hour >= HOURS_PER_DAY)
         throw new IllegalArgumentException("Hour: " + hour);
      if (minute < 0 || minute >= MINUTES_PER_HOUR)
         throw new IllegalArgumentException("Min: " + minute);
      this.hour = hour;
      this.minute = minute;
   }
   ... // Remainder omitted
}
```

In summary, public classes should never expose mutable fields. It is less harmful, though still questionable, for public classes to expose immutable fields. It is, however, sometimes desirable for package-private or private nested classes to expose fields, whether mutable or immutable.

ITEM 17: MINIMIZE MUTABILITY

An immutable class is simply a class whose instances cannot be modified. All of the information contained in each instance is fixed for the lifetime of the object, so no changes can ever be observed. The Java platform libraries contain many immutable classes, including String, the boxed primitive classes,
and BigInteger and BigDecimal

```
// Immutable complex number class
public final class Complex {
  private final double re;
  private final double im;

  public Complex(double re, double im) {
    this.re = re;
    this.im = im;
  }

  public double realPart()      { return re; }
  public double imaginaryPart() { return im; }

  public Complex plus(Complex c) {
    return new Complex(re + c.re, im + c.im);
  }

  public Complex minus(Complex c) {
    return new Complex(re - c.re, im - c.im);
  }

  public Complex times(Complex c) {
    return new Complex(re * c.re - im * c.im,
                       re * c.im + im * c.re);
  }
```

```
    double tmp = c.re * c.re + c.im * c.im;
    return new Complex((re * c.re + im * c.im) / tmp,
                       (im * c.re - re * c.im) / tmp);
  }

  @Override public boolean equals(Object o) {
    if (o == this)
      return true;
    if (!(o instanceof Complex))
      return false;
    Complex c = (Complex) o;

    // See page 47 to find out why we use compare instead of ==
    return Double.compare(c.re, re) == 0
      && Double.compare(c.im, im) == 0;
  }
  @Override public int hashCode() {
    return 31 * Double.hashCode(re) + Double.hashCode(im);
  }

  @Override public String toString() {
    return "(" + re + " + " + im + "i)";
  }
}
```

Immutable objects are inherently thread-safe; they require no synchronization.

immutable objects can be shared freely

To make a class immutable, follow these five rules:

1. **Don't provide methods that modify the object's state** (known as *mutators*).

2. **Ensure that the class can't be extended.** This prevents careless or malicious subclasses from compromising the immutable behavior of the class by behaving as if the object's state has changed. Preventing subclassing is generally accomplished by making the class final, but there is an alternative that we'll discuss later.

3. **Make all fields final.** This clearly expresses your intent in a manner that is enforced by the system. Also, it is necessary to ensure correct behavior if a reference to a newly created instance is passed from one thread to another without synchronization, as spelled out in the *memory model* [JLS, 17.5; Goetz06, 16].

4. **Make all fields private.** This prevents clients from obtaining access to mutable objects referred to by fields and modifying these objects directly. While it is technically permissible for immutable classes to have public final fields containing primitive values or references to immutable objects, it is not recommended because it precludes changing the internal representation in a later release (Items 15 and 16).

5. **Ensure exclusive access to any mutable components.** If your class has any fields that refer to mutable objects, ensure that clients of the class cannot obtain references to these objects. Never initialize such a field to a client-provided object reference or return the field from an accessor. Make *defensive copies* (Item 50) in constructors, accessors, and `readObject` methods (Item 88).

```
// Immutable class with static factories instead of
constructors
public class Complex {
   private final double re;
   private final double im;

   private Complex(double re, double im) {
      this.re = re;
      this.im = im;
   }

   public static Complex valueOf(double re, double im) {
      return new Complex(re, im);
   }

   ... // Remainder unchanged
}
```

This approach is often the best alternative. It is the most flexible because it allows the use of multiple package-private implementation classes. To its clients that reside outside its package, the immutable class is effectively final because it is impossible to extend a class that comes from another package and that lacks a public or protected constructor. Besides allowing the flexibility of multiple implementation classes, this approach makes it possible to tune the performance of the class in subsequent releases by improving the object-caching capabilities of the static factories.

ITEM 18: FAVOR COMPOSITION OVER INHERITANCE

Unlike method invocation, inheritance violates encapsulation

```
public class InstrumentedHashSet<E> extends HashSet<E> {
  // The number of attempted element insertions
  private int addCount = 0;

  public InstrumentedHashSet() {
  }

  public InstrumentedHashSet(int initCap, float loadFactor) {
    super(initCap, loadFactor);
  }
  @Override public boolean add(E e) {
    addCount++;
    return super.add(e);
  }
  @Override public boolean addAll(Collection<? extends E> c) {
    addCount += c.size();
    return super.addAll(c);
  }
  public int getAddCount() {
```

```
    return addCount;
  }
}
```

```
public class InstrumentedSet<E> extends ForwardingSet<E> {
  private int addCount = 0;

  public InstrumentedSet(Set<E> s) {
    super(s);
  }
```

```
  @Override public boolean add(E e) {
    addCount++;
    return super.add(e);
  }
  @Override public boolean addAll(Collection<? extends E> c) {
    addCount += c.size();
    return super.addAll(c);
  }
  public int getAddCount() {
    return addCount;
  }
}
```

ITEM 19: DESIGN AND DOCUMENT FOR INHERITANCE OR ELSE PROHIBIT IT

The class must document its self-use of overridable methods.

Designing a class for inheritance requires great effort and places substantial limitations on the class

The best solution to this problem is to prohibit subclassing in classes that are not designed and documented to be safely subclassed. There are two ways to prohibit subclassing. The easier of the two is to declare the class final. The alternative is to make all the constructors private or package-private and to add public static factories in place of the constructors.

ITEM 20: PREFER INTERFACES TO ABSTRACT CLASSES

Existing classes can easily be retrofitted to implement a new interface.

Interfaces are ideal for defining mixins.

Interfaces allow for the construction of nonhierarchical type frameworks

```
public interface Singer {
   AudioClip sing(Song s);
}
public interface Songwriter {
   Song compose(int chartPosition);
}
```

In real life, some singers are also songwriters. Because we used interfaces rather than abstract classes to define these types, it is perfectly permissible for a single class to implement both `Singer` and `Songwriter`. In fact, we can define a third interface that extends both `Singer` and `Songwriter` and adds new methods that are appropriate to the combination:

Click here to view code image

```
public interface SingerSongwriter extends Singer, Songwriter {
   AudioClip strum();
   void actSensitive();
}
```

Interfaces enable safe, powerful functionality enhancements

```
public abstract class AbstractMapEntry<K,V>
    implements Map.Entry<K,V> {
  // Entries in a modifiable map must override this method
  @Override public V setValue(V value) {
    throw new UnsupportedOperationException();
  }
  // Implements the general contract of Map.Entry.equals
  @Override public boolean equals(Object o) {
    if (o == this)
      return true;
    if (!(o instanceof Map.Entry))
      return false;
    Map.Entry<?,?> e = (Map.Entry) o;
```

```
    return Objects.equals(e.getKey(),  getKey())
      && Objects.equals(e.getValue(), getValue());
  }

  // Implements the general contract of Map.Entry.hashCode
  @Override public int hashCode() {
    return Objects.hashCode(getKey())
        ^ Objects.hashCode(getValue());
  }

  @Override public String toString() {
    return getKey() + "=" + getValue();
  }
}
```

ITEM 21: DESIGN INTERFACES FOR POSTERITY

```
default boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);
    boolean result = false;
    for (Iterator<E> it = iterator(); it.hasNext(); ) {
        if (filter.test(it.next())) {
            it.remove();
            result = true;
        }
    }
    return result;
}
```

ITEM 22: USE INTERFACES ONLY TO DEFINE TYPES

```
// Constant interface antipattern - do not use!
public interface PhysicalConstants {
    // Avogadro's number (1/mol)
    static final double AVOGADROS_NUMBER    =
6.022_140_857e23;

    // Boltzmann constant (J/K)
    static final double BOLTZMANN_CONSTANT =
1.380_648_52e-23;

    // Mass of the electron (kg)
    static final double ELECTRON_MASS      = 9.109_383_56e-31;
}
```

```
public class PhysicalConstants {
  private PhysicalConstants() { }  // Prevents instantiation

  public static final double AVOGADROS_NUMBER =
6.022_140_857e23;
  public static final double BOLTZMANN_CONST  =
1.380_648_52e-23;
  public static final double ELECTRON_MASS    =
9.109_383_56e-31;
}
```

```
import static com.effectivejava.science.PhysicalConstants.*;

public class Test {
   double  atoms(double mols) {
      return AVOGADROS_NUMBER * mols;
   }
   ...
   // Many more uses of PhysicalConstants justify static import
}
```

ITEM 23: PREFER CLASS HIERARCHIES TO TAGGED CLASSES

```
class Figure {
   enum Shape { RECTANGLE, CIRCLE };

   // Tag field - the shape of this figure
   final Shape shape;

   // These fields are used only if shape is RECTANGLE
   double length;
   double width;

   // This field is used only if shape is CIRCLE
   double radius;

   // Constructor for circle
   Figure(double radius) {
      shape = Shape.CIRCLE;
```

```
      this.radius = radius;
   }

   // Constructor for rectangle
   Figure(double length, double width) {
      shape = Shape.RECTANGLE;
      this.length = length;
      this.width = width;
   }
```

Occasionally you may run across a class whose instances come in two or more flavors and contain a tag field indicating the flavor of the instance.

tagged classes are verbose, error-prone, and inefficient.

A tagged class is just a pallid imitation of a class hierarchy

```
// Class hierarchy replacement for a tagged class
abstract class Figure {
   abstract double area();
}
```

```
class Circle extends Figure {
   final double radius;

   Circle(double radius) { this.radius = radius; }

   @Override double area() { return Math.PI * (radius * radius); }
}
class Rectangle extends Figure {
   final double length;
   final double width;

   Rectangle(double length, double width) {
      this.length = length;
      this.width  = width;
   }
   @Override double area() { return length * width; }
}
```

```
class Square extends Rectangle {
   Square(double side) {
      super(side, side);
   }
```

ITEM 24: FAVOR STATIC MEMBER CLASSES OVER NONSTATIC

A nested class is a class defined within another class. A nested class should exist only to serve its enclosing class. If a nested class would be useful in some other context, then it should be a top-level class. There are four kinds of nested classes: static member classes, nonstatic member classes, anonymous classes, and local
classes. All but the first kind are known as inner classes.

```
// Typical use of a nonstatic member class
public class MySet<E> extends AbstractSet<E> {
  ... // Bulk of the class omitted

  @Override public Iterator<E> iterator() {
    return new MyIterator();
  }

  private class MyIterator implements Iterator<E> {
    ...
  }
}
```

If you declare a member class that does not require access to an enclosing instance, always put
the static modifier in its declaration

(
https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html)

ITEM 25: LIMIT SOURCE FILES TO A SINGLE TOP-LEVEL CLASS

While the Java compiler lets you define multiple top-level classes in a single source file, there are no benefits associated with doing so, and there are significant risks

```
// Two classes defined in one file. Don't ever do this!
class Utensil {
    static final String NAME = "pot";
}

class Dessert {
    static final String NAME = "pie";
}
```

```
// Static member classes instead of multiple top-level
classes
public class Test {
   public static void main(String[] args) {
      System.out.println(Utensil.NAME + Dessert.NAME);
   }

   private static class Utensil {
      static final String NAME = "pan";
   }

   private static class Dessert {
      static final String NAME = "cake";
   }
}
```

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.

ITEM 26: DON’T USE RAW TYPES

```
// Raw collection type - don't do this!

// My stamp collection. Contains only Stamp instances.
private final Collection stamps = ... ;

// 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:

Click here to view code image

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

```
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);
}
```

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:**

Click here to view code image

```
// Legitimate use of raw type - instanceof operator
if (o instanceof Set) {       // Raw type
    Set<?> s = (Set<?>) o;    // Wildcard type
    ...
}
```

Term	Example
Parameterized type	`List<String>`
Actual type parameter	`String`
Generic type	`List<E>`
Formal type parameter	`E`
Unbounded wildcard type	`List<?>`
Raw type	`List`
Bounded type parameter	`<E extends Number>`
Recursive type bound	`<T extends Comparable<T>>`
Bounded wildcard type	`List<? extends Number>`
Generic method	`static <E> List<E> asList(E[] a)`
Type token	`String.class`

ITEM 27: ELIMINATE UNCHECKED WARNINGS

Eiminate every unchecked warning that you can

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

Always use the SuppressWarnings annotation on the smallest scope possible

```
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);
   if (a.length > size)
      a[size] = null;
   return a;
}
```

```
@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;
}
```

Every time you use a @SuppressWarnings("unchecked") annotation, add a comment saying why it is safe to do so.

(https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/)

ITEM 28: PREFER LISTS TO ARRAYS

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

but this one is not:

Click here to view code image

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

```
// 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)
```

```
// 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)];
  }
}

// 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()));
  }
}
```

ITEM 29: FAVOR GENERIC TYPES

Writing your own generic types is a bit more difficult, but it’s worth the effort to learn how.

```
public class Stack<E> {
private E[] elements;
if (size == 0)
throw new EmptyStackException();
private int size = 0;
private static final int DEFAULT_INITIAL_CAPACITY = 16;

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

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

public E pop() { if (size == 0)
throw new EmptyStackException();
@SuppressWarnings("unchecked")
E result = (E) elements[--size];
elements[size] = null;
}
}
```

ITEM 30: FAVOR GENERIC METHODS

```
// 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:

Click here to view code image

```
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);
                   ^
```

The type parameter list, which declares the type parameters, goes between a method's modifiers and its return type.

```
// 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;
}
```

```
// 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);
}

// 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;
}
```

ITEM 31: USE BOUNDED WILDCARDS TO INCREASE API FLEXIBILITY

If the user of a class has to think about wildcard types, there is probably something wrong with its API.

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

Click here to view code image

```
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)));
}
```

ITEM 32: COMBINE GENERICS AND VARARGS JUDICIOUSLY

```
// 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
}
```

it is unsafe to store a value in a generic varargs array parameter.

the SafeVarargs annotation constitutes a promise by the author of a method that it is typesafe

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

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
}
```

```
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();
}
```

ITEM 33: CONSIDER TYPESAFE HETEROGENEOUS CONTAINERS

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

Click here to view code image

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

```
// 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));
   }
}
```

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

```
// 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));
}
```

Chapter 6. Enums and Annotations

JAVA supports two special-purpose families of reference types: a kind of class called an enum type, and a kind of interface called an annotation type

ITEM 34: USE ENUMS INSTEAD OF INT CONSTANTS

An enumerated type is a type whose legal values consist of a fixed set of constants

```
// The int enum pattern - severely deficient!
public static final int APPLE_FUJI         = 0;
public static final int APPLE_PIPPIN       = 1;
public static final int APPLE_GRANNY_SMITH = 2;
public static final int ORANGE_NAVEL  = 0;
public static final int ORANGE_TEMPLE = 1;
public static final int ORANGE_BLOOD  = 2;
```

```
// Tasty citrus flavored applesauce!
int i = (APPLE_FUJI - ORANGE_TEMPLE) / APPLE_PIPPIN;

public enum Apple  { FUJI, PIPPIN, GRANNY_SMITH }
public enum Orange { NAVEL, TEMPLE, BLOOD }


// Enum type with data and behavior
public enum Planet {
    MERCURY(3.302e+23, 2.439e6),
    VENUS  (4.869e+24, 6.052e6),
    EARTH  (5.975e+24, 6.378e6),
    MARS   (6.419e+23, 3.393e6),
    JUPITER(1.899e+27, 7.149e7),
    SATURN (5.685e+26, 6.027e7),
    URANUS (8.683e+25, 2.556e7),
    NEPTUNE(1.024e+26, 2.477e7);

    private final double mass;          // In kilograms
    private final double radius;        // In meters
```

```
    private final double surfaceGravity; // In m / s^2

    // Universal gravitational constant in m^3 / kg s^2
    private static final double G = 6.67300E-11;

    // Constructor
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
        surfaceGravity = G * mass / (radius * radius);
    }

    public double mass()          { return mass; }
    public double radius()        { return radius; }
    public double surfaceGravity() { return surfaceGravity; }

    public double surfaceWeight(double mass) {
        return mass * surfaceGravity;  // F = ma
    }
}
```

To associate data with enum constants, declare instance fields and write a constructor that takes the data and stores it in the fields

```
public class WeightTable {
  public static void main(String[] args) {
    double earthWeight = Double.parseDouble(args[0]);
    double mass = earthWeight / Planet.EARTH.surfaceGravity();
    for (Planet p : Planet.values())
       System.out.printf("Weight on %s is %f%n",
                         p, p.surfaceWeight(mass));
    }
}
```

// Enum type that switches on its own value - questionable

```
public enum Operation {
   PLUS, MINUS, TIMES, DIVIDE;

   // Do the arithmetic operation represented by this constant
   public double apply(double x, double y) {
      switch(this) {
         case PLUS:    return x + y;
         case MINUS:  return x - y;
         case TIMES:  return x * y;
         case DIVIDE: return x / y;
      }
      throw new AssertionError("Unknown op: " + this);
   }
}
```

```
// Enum type with constant-specific method implementations
public enum Operation {
  PLUS  {public double apply(double x, double y){return x + y;}},
  MINUS {public double apply(double x, double y){return x - y;}},
  TIMES {public double apply(double x, double y){return x * y;}},
  DIVIDE{public double apply(double x, double y){return x / y;}};

  public abstract double apply(double x, double y);
}
  // Enum type with constant-specific class bodies and data
  public enum Operation {
     PLUS("+") {
        public double apply(double x, double y) { return x + y; }
     },
     MINUS("-") {
        public double apply(double x, double y) { return x - y; }
     },
     TIMES("*") {
        public double apply(double x, double y) { return x * y; }
     },
     DIVIDE("/") {
        public double apply(double x, double y) { return x / y; }
     };

     private final String symbol;

     Operation(String symbol) { this.symbol = symbol; }

     @Override public String toString() { return symbol; }

     public abstract double apply(double x, double y);
  }
```

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    for (Operation op : Operation.values())
        System.out.printf("%f %s %f = %f%n",
                          x, op, y, op.apply(x, y));
}
```

```
// Implementing a fromString method on an enum type
private static final Map<String, Operation> stringToEnum =
        Stream.of(values()).collect(
            toMap(Object::toString, e -> e));

// Returns Operation for string, if any
public static Optional<Operation> fromString(String symbol) {
    return Optional.ofNullable(stringToEnum.get(symbol));
}
```

```
// Switch on an enum to simulate a missing method
public static Operation inverse(Operation op) {
  switch(op) {
    case PLUS:   return Operation.MINUS;
    case MINUS:  return Operation.PLUS;
    case TIMES:  return Operation.DIVIDE;
    case DIVIDE: return Operation.TIMES;

    default:  throw new AssertionError("Unknown op: " + op);
  }
}
```

Use enums any time you need a set of constants whose members are known at compile time.

It is not necessary that the set of constants in an enum type stay fixed for all time.

ITEM 35: USE INSTANCE FIELDS INSTEAD OF ORDINALS

All enums have an ordinal method, which returns the numerical position of each enum constant in its type.

Never derive a value associated with an enum from its ordinal; store it in an instance field instead:

```
public enum Ensemble {
   SOLO,   DUET,   TRIO, QUARTET, QUINTET,
   SEXTET, SEPTET, OCTET, NONET,  DECTET;

   public int numberOfMusicians() { return ordinal() + 1; }
}
```

```
public enum Ensemble {
   SOLO(1), DUET(2), TRIO(3), QUARTET(4), QUINTET(5),
   SEXTET(6), SEPTET(7), OCTET(8), DOUBLE_QUARTET(8),
   NONET(9), DECTET(10), TRIPLE_QUARTET(12);

   private final int numberOfMusicians;
   Ensemble(int size) { this.numberOfMusicians = size; }
   public int numberOfMusicians() { return numberOfMusicians; }
}
```

ITEM 36: USE ENUMSET INSTEAD OF BIT FIELDS

```
// Bit field enumeration constants - OBSOLETE!
public class Text {
   public static final int STYLE_BOLD          = 1 << 0;  // 1
   public static final int STYLE_ITALIC        = 1 << 1;  // 2
   public static final int STYLE_UNDERLINE     = 1 << 2;  // 4
   public static final int STYLE_STRIKETHROUGH = 1 << 3;  // 8

   // Parameter is bitwise OR of zero or more STYLE_ constants
   public void applyStyles(int styles) { ... }
}

   // EnumSet - a modern replacement for bit fields
   public class Text {
      public enum Style { BOLD, ITALIC, UNDERLINE,
```

```
   STRIKETHROUGH }

      // Any Set could be passed in, but EnumSet is clearly best
      public void applyStyles(Set<Style> styles) { ... }
   }
```

ITEM 37: USE ENUMMAP INSTEAD OF ORDINAL INDEXING

```
class Plant {
   enum LifeCycle { ANNUAL, PERENNIAL, BIENNIAL }

   final String name;
   final LifeCycle lifeCycle;

   Plant(String name, LifeCycle lifeCycle) {
      this.name = name;
      this.lifeCycle = lifeCycle;
   }

   @Override public String toString() {
      return name;
   }
}
```

```
System.out.println(Arrays.stream(garden)
      .collect(groupingBy(p -> p.lifeCycle,
         () -> new EnumMap<>(LifeCycle.class), toSet())));
```

```
// Naive stream-based approach - unlikely to produce an
EnumMap!
System.out.println(Arrays.stream(garden)
      .collect(groupingBy(p -> p.lifeCycle)));
```

```
// Using ordinal() to index into an array - DON'T DO THIS!
Set<Plant>[] plantsByLifeCycle =
    (Set<Plant>[]) new Set[Plant.LifeCycle.values().length];
for (int i = 0; i < plantsByLifeCycle.length; i++)
```

```
    plantsByLifeCycle[i] = new HashSet<>();

for (Plant p : garden)
    plantsByLifeCycle[p.lifeCycle.ordinal()].add(p);

// Print the results
for (int i = 0; i < plantsByLifeCycle.length; i++) {
    System.out.printf("%s: %s%n",
        Plant.LifeCycle.values()[i], plantsByLifeCycle[i]);
}
```

```
    // Using an EnumMap to associate data with an enum
    Map<Plant.LifeCycle, Set<Plant>>  plantsByLifeCycle =
        new EnumMap<>(Plant.LifeCycle.class);
    for (Plant.LifeCycle lc : Plant.LifeCycle.values())
```

```
        plantsByLifeCycle.put(lc, new HashSet<>());
    for (Plant p : garden)
        plantsByLifeCycle.get(p.lifeCycle).add(p);
    System.out.println(plantsByLifeCycle);
```

// Using ordinal() to index array of arrays - DON'T DO THIS!

```
public enum Phase {
   SOLID, LIQUID, GAS;
```

```
   public enum Transition {
      MELT, FREEZE, BOIL, CONDENSE, SUBLIME, DEPOSIT;

      // Rows indexed by from-ordinal, cols by to-ordinal
      private static final Transition[][] TRANSITIONS = {
         { null,    MELT,     SUBLIME },
         { FREEZE,  null,      BOIL    },
         { DEPOSIT, CONDENSE, null    }
      };

      // Returns the phase transition from one phase to another
      public static Transition from(Phase from, Phase to) {
         return TRANSITIONS[from.ordinal()][to.ordinal()];
      }
   }
}
```

```
public enum Phase {
  SOLID, LIQUID, GAS;

  public enum Transition {
    MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),
    BOIL(LIQUID, GAS),    CONDENSE(GAS, LIQUID),
    SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID);

    private final Phase from;
    private final Phase to;

    Transition(Phase from, Phase to) {
      this.from = from;
      this.to = to;
    }

    // Initialize the phase transition map
    private static final Map<Phase, Map<Phase, Transition>>
     m = Stream.of(values()).collect(groupingBy(t -> t.from,
      () -> new EnumMap<>(Phase.class),
      toMap(t -> t.to, t -> t,
       (x, y) -> y, () -> new EnumMap<>(Phase.class))));

    public static Transition from(Phase from, Phase to) {
      return m.get(from).get(to);
    }
```

```
  }
}
```

```
// Adding a new phase using the nested EnumMap implementation
public enum Phase {
```

```
    SOLID, LIQUID, GAS, PLASMA;

    public enum Transition {
        MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),
        BOIL(LIQUID, GAS),   CONDENSE(GAS, LIQUID),
        SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID),
        IONIZE(GAS, PLASMA), DEIONIZE(PLASMA, GAS);
        ... // Remainder unchanged
    }
}
```

it is rarely appropriate to use ordinals to index into arrays: use EnumMap instead.

ITEM 38: EMULATE EXTENSIBLE ENUMS WITH INTERFACES

```
// Emulated extensible enum using an interface
public interface Operation {
    double apply(double x, double y);
}


 public enum BasicOperation implements Operation {
    PLUS("+") {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        public double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        public double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        public double apply(double x, double y) { return x / y; }
    };
    private final String symbol;

    BasicOperation(String symbol) {
        this.symbol = symbol;
    }

    @Override public String toString() {
        return symbol;
    }
}
```

```
// Emulated extension enum
public enum ExtendedOperation implements Operation {
  EXP("^") {
    public double apply(double x, double y) {
      return Math.pow(x, y);
    }
  },
  REMAINDER("%") {
    public double apply(double x, double y) {
      return x % y;
    }
  };

  private final String symbol;

  ExtendedOperation(String symbol) {
    this.symbol = symbol;
  }

  @Override public String toString() {
    return symbol;
  }
}
```

```
    public static void main(String[] args) {
        double x = Double.parseDouble(args[0]);
        double y = Double.parseDouble(args[1]);
        test(ExtendedOperation.class, x, y);
    }

    private static <T extends Enum<T> & Operation> void test(
            Class<T> opEnumType, double x, double y) {
        for (Operation op : opEnumType.getEnumConstants())
            System.out.printf("%f %s %f = %f%n",
                              x, op, y, op.apply(x, y));
    }

public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    test(Arrays.asList(ExtendedOperation.values()), x, y);
}

private static void test(Collection<? extends Operation>
opSet,
        double x, double y) {
    for (Operation op : opSet)
        System.out.printf("%f %s %f = %f%n",
                          x, op, y, op.apply(x, y));
}
```

ITEM 39: PREFER ANNOTATIONS TO NAMING PATTERNS

```
// Marker annotation type declaration
import java.lang.annotation.*;

/**
 * Indicates that the annotated method is a test method.
 * Use only on parameterless static methods.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
}
```

```
// Program containing marker annotations
public class Sample {
    @Test public static void m1() { }  // Test should pass
    public static void m2() { }
    @Test public static void m3() {      // Test should fail
        throw new RuntimeException("Boom");
    }

    public static void m4() { }
    @Test public void m5() { } // INVALID USE: nonstatic
method
    public static void m6() { }
    @Test public static void m7() {    // Test should fail
        throw new RuntimeException("Crash");
    }
    public static void m8() { }
}
```

```
// Program containing annotations with a parameter
public class Sample2 {
   @ExceptionTest(ArithmeticException.class)
   public static void m1() {  // Test should pass
      int i = 0;
      i = i / i;
   }
   @ExceptionTest(ArithmeticException.class)
   public static void m2() {  // Should fail (wrong exception)
      int[] a = new int[0];
      int i = a[1];
   }
   @ExceptionTest(ArithmeticException.class)
   public static void m3() { }  // Should fail (no exception)
}

  // Repeatable annotation type
  @Retention(RetentionPolicy.RUNTIME)
  @Target(ElementType.METHOD)
  @Repeatable(ExceptionTestContainer.class)
  public @interface ExceptionTest {
     Class<? extends Exception> value();
  }

  @Retention(RetentionPolicy.RUNTIME)
  @Target(ElementType.METHOD)
  public @interface ExceptionTestContainer {
     ExceptionTest[] value();
  }
```

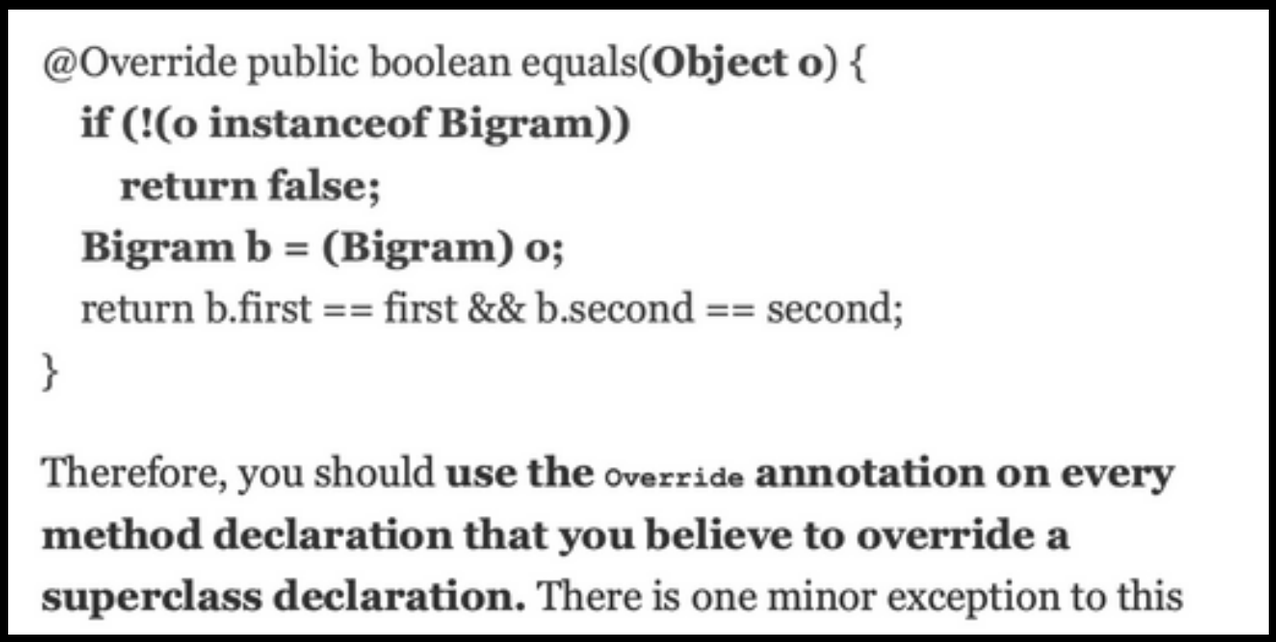
ITEM 40: CONSISTENTLY USE THE OVERRIDE ANNOTATION

```
// Can you spot the bug?
public class Bigram {
    private final char first;
    private final char second;

    public Bigram(char first, char second) {
        this.first  = first;
        this.second = second;
    }
    public boolean equals(Bigram b) {
        return b.first == first && b.second == second;
    }
    public int hashCode() {
        return 31 * first + second;
    }

    public static void main(String[] args) {
        Set<Bigram> s = new HashSet<>();
        for (int i = 0; i < 10; i++)
            for (char ch = 'a'; ch <= 'z'; ch++)
                s.add(new Bigram(ch, ch));
        System.out.println(s.size());
    }
}
```

```
@Override public boolean equals(Bigram b) {
    return b.first == first && b.second == second;
}
```



```
@Override public boolean equals(Object o) {
  if (!(o instanceof Bigram))
    return false;
  Bigram b = (Bigram) o;
  return b.first == first && b.second == second;
}
```

Therefore, you should **use the `Override` annotation on every method declaration that you believe to override a superclass declaration.** There is one minor exception to this

ITEM 41: USE MARKER INTERFACES TO DEFINE TYPES

A marker interface is an interface that contains no method declarations but merely designates (or “marks”) a class that implements the interface as having some property.

marker interfaces define a type that is implemented by instances of the marked class; marker annotations do not.

Another advantage of marker interfaces over marker annotations is that they can be targeted more precisely

If you find yourself writing a marker annotation type whose target is ElementType.TYPE, take the time to figure out whether it really should be an annotation type or whether a marker interface would be more appropriate.

For instance, let's add a restriction that only a *Shape* type can be removed from the database:

```
public interface Shape {
    double getArea();
    double getCircumference();
}
```

In this case, our marker interface, *DeletableShape*, will look like this:

```
public interface DeletableShape extends Shape {
}
```

Then our class will implement the marker interface:

```
public class Rectangle implements DeletableShape {
    // implementation details
}
```

Therefore, **all *DeletableShape* implementations are also *Shape* implementations**. Obviously, **we can't do that using annotations**.

However, every design decision has trade-offs, and we can use **polymorphism as a counter-argument** against marker interfaces. **In our example, every class extending *Rectangle* will automatically implement *DeletableShape*.**

Chapter 7. Lambdas and Streams

ITEM 42: PREFER LAMBDAS TO ANONYMOUS CLASSES

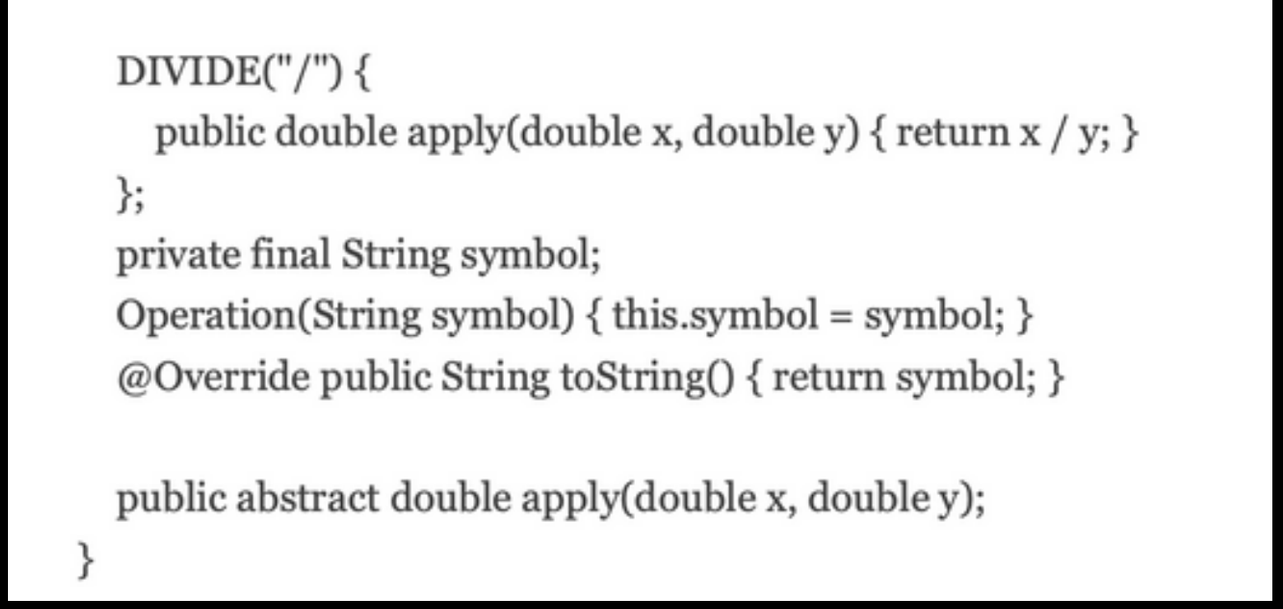
```
// Anonymous class instance as a function object - obsolete!
Collections.sort(words, new Comparator<String>() {
    public int compare(String s1, String s2) {
        return Integer.compare(s1.length(), s2.length());
    }
});

// Lambda expression as function object (replaces anonymous class)
Collections.sort(words,
        (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

Collections.sort(words, comparingInt(String::length));

words.sort(comparingInt(String::length));

```
// Enum type with constant-specific class bodies & data (Item 34)
public enum Operation {
    PLUS("+") {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        public double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        public double apply(double x, double y) { return x * y; }
    },
```



```
    DIVIDE("/") {
      public double apply(double x, double y) { return x / y; }
    };
    private final String symbol;
    Operation(String symbol) { this.symbol = symbol; }
    @Override public String toString() { return symbol; }

    public abstract double apply(double x, double y);
}
```

lambdas lack names and documentation; if a computation isn’t self-explanatory, or exceeds a few lines, don’t put it in a lambda

you should rarely, if ever, serialize a lambda

Don’t use anonymous classes for function objects unless you have to create instances of types that aren’t functional interfaces

ITEM 43: PREFER METHOD REFERENCES TO LAMBDAS

```
map.merge(key, 1, (count, incr) -> count + incr);
map.merge(key, 1, Integer::sum);

service.execute(GoshThisClassNameIsHumongous::action);

service.execute(() -> action());
```

Where method references are shorter and clearer, use them; where they aren’t, stick with lambdas.

Method Ref Type	Example	Lambda Equivalent
Static	`Integer::parseInt`	`str -> Integer.parseInt(str)`
Bound	`Instant.now()::isAfter`	`Instant then = Instant.now(); t -`
Unbound	`String::toLowerCase`	`str -> str.toLowerCase()`
Class Constructor	`TreeMap<K,V>::new`	`() -> new TreeMap<K,V>`
Array Constructor	`int[]::new`	`len -> new int[len]`

ITEM 44: FAVOR THE USE OF STANDARD FUNCTIONAL INTERFACES

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
   return size() > 100;
}
```

```
// Unnecessary functional interface; use a standard one
instead.
@FunctionalInterface interface
EldestEntryRemovalFunction<K,V>{
   boolean remove(Map<K,V> map, Map.Entry<K,V> eldest);
}
```

If one of the standard functional interfaces does the job, you should generally use it in preference to a purpose-built functional interface

Always annotate your functional interfaces with the @FunctionalInterface annotation. java.util.function.Function

ITEM 45: USE STREAMS JUDICIOUSLY

```
// Prints all large anagram groups in a dictionary iteratively
public class Anagrams {

    public static void main(String[] args) throws IOException {
        File dictionary = new File(args[0]);
        int minGroupSize = Integer.parseInt(args[1]);

        Map<String, Set<String>> groups = new HashMap<>();
        try (Scanner s = new Scanner(dictionary)) {
            while (s.hasNext()) {
                String word = s.next();
                groups.computeIfAbsent(alphabetize(word),
                    (unused) -> new TreeSet<>()).add(word);
            }
        }

        for (Set<String> group : groups.values())
            if (group.size() >= minGroupSize)
                System.out.println(group.size() + ": " + group);
    }

    private static String alphabetize(String s) {
        char[] a = s.toCharArray();
        Arrays.sort(a);
        return new String(a);
    }
}
```

```
// Overuse of streams - don't do this!
public class Anagrams {
  public static void main(String[] args) throws IOException {
    Path dictionary = Paths.get(args[0]);
    int minGroupSize = Integer.parseInt(args[1]);

    try (Stream<String> words = Files.lines(dictionary)) {
      words.collect(
        groupingBy(word -> word.chars().sorted()
                .collect(StringBuilder::new,
                 (sb, c) -> sb.append((char) c),
                 StringBuilder::append).toString()))
        .values().stream()
        .filter(group -> group.size() >= minGroupSize)
        .map(group -> group.size() + ": " + group)
        .forEach(System.out::println);
      }
    }
}
```

Overusing streams makes programs hard to read and maintain.

In the absence of explicit types, careful naming of lambda parameters is essential to the readability of stream pipelines.

Using helper methods is even more important for readability in stream pipelines than in iterative code

```
"Hello world!".chars().forEach(System.out::print);
"Hello world!".chars().forEach(x -> System.out.print((char) x));
```

```
static Stream<BigInteger> primes() {
return Stream.iterate(TWO, BigInteger::nextProbablePrime);
}
```

```
public static void main(String[] args) {
   primes().map(p -> TWO.pow(p.intValueExact()).subtract(ONE))
      .filter(mersenne -> mersenne.isProbablePrime(50))
      .limit(20)
      .forEach(System.out::println);
}
```

```
  // Iterative Cartesian product computation
  private static List<Card> newDeck() {
     List<Card> result = new ArrayList<>();
     for (Suit suit : Suit.values())
        for (Rank rank : Rank.values())
           result.add(new Card(suit, rank));
     return result;
  }

// Stream-based Cartesian product computation
private static List<Card> newDeck() {
   return Stream.of(Suit.values())
      .flatMap(suit ->
         Stream.of(Rank.values())
            .map(rank -> new Card(suit, rank)))
      .collect(toList());
}
```

ITEM 46: PREFER SIDE-EFFECT-FREE FUNCTIONS IN STREAMS

```
// Uses the streams API but not the paradigm--Don't do this!
Map<String, Long> freq = new HashMap<>();
try (Stream<String> words = new Scanner(file).tokens()) {
   words.forEach(word -> {
      freq.merge(word.toLowerCase(), 1L, Long::sum);

// Proper use of streams to initialize a frequency table
Map<String, Long> freq;
try (Stream<String> words = new Scanner(file).tokens()) {
   freq = words
      .collect(groupingBy(String::toLowerCase, counting()));
}
```

The forEach operation should be used only to report the result of a stream computation, not to perform the computation

import all members of Collectors because it makes stream pipelines more readable.

```
// Pipeline to get a top-ten list of words from a frequency table
List<String> topTen = freq.keySet().stream()
   .sorted(comparing(freq::get).reversed())
   .limit(10)
   .collect(toList());
```

```
private static final Map<String, Operation> stringToEnum =
Stream.of(values()).collect( toMap(Object::toString, e -> e));

Map<Artist, Album> topHits = albums.collect(
toMap(Album::artist, a->a, maxBy(comparing(Album::sales))));

// Collector to impose last-write-wins policy
toMap(keyMapper, valueMapper, (v1, v2) -> v2)

words.collect(groupingBy(word -> alphabetize(word)))

Map<String, Long> freq = words
.collect(groupingBy(String::toLowerCase,
counting()));
```

ITEM 47: PREFER COLLECTION TO STREAM AS A RETURN TYPE

```
for (ProcessHandle ph : ProcessHandle.allProcesses()::iterator) {
```

```
// Adapter from Iterable<E> to Stream<E>
public static <E> Stream<E> streamOf(Iterable<E> iterable) {
    return StreamSupport.stream(iterable.spliterator(), false);
}
```

```
// Adapter from  Stream<E> to Iterable<E>
public static <E> Iterable<E> iterableOf(Stream<E> stream) {
   return stream::iterator;
}
```

With this adapter, you can iterate over any stream with a for-each statement:

Click here to view code image

```
for (ProcessHandle p : iterableOf(ProcessHandle.allProcesses())) {
   // Process the process
}
```

```
// Returns a stream of all the sublists of its input list
public class SubLists {
   public static <E> Stream<List<E>> of(List<E> list) {
      return Stream.concat(Stream.of(Collections.emptyList()),
         prefixes(list).flatMap(SubLists::suffixes));
   }

   private static <E> Stream<List<E>> prefixes(List<E> list) {
      return IntStream.rangeClosed(1, list.size())
         .mapToObj(end -> list.subList(0, end));
   }

private static <E> Stream<List<E>> suffixes(List<E> list) {
   return IntStream.range(0, list.size())
      .mapToObj(start -> list.subList(start, list.size()));
}
}
```

```
// Returns a stream of all the sublists of its input list
public static <E> Stream<List<E>> of(List<E> list) {
  return IntStream.range(0, list.size())
    .mapToObj(start ->
      IntStream.rangeClosed(start + 1, list.size())

        .mapToObj(end -> list.subList(start, end)))
    .flatMap(x -> x);
}
```

ITEM 48: USE CAUTION WHEN MAKING STREAMS PARALLEL

```
// Stream-based program to generate the first 20
Mersenne primes
public static void main(String[] args) {
   primes().map(p -> TWO.pow(p.intValueExact()).subtract(ONE))
      .filter(mersenne -> mersenne.isProbablePrime(50))
      .limit(20)
      .forEach(System.out::println);
}

static Stream<BigInteger> primes() {

   return Stream.iterate(TWO, BigInteger::nextProbablePrime);
}
```

parallelizing a pipeline is unlikely to increase its performance if the source is from Stream.iterate, or the intermediate operation limit is used

Do not parallelize stream pipelines indiscriminately

Performance gains from parallelism are best on streams over ArrayList, HashMap, HashSet, and ConcurrentHashMap instances; arrays; int ranges; and long ranges

Not only can parallelizing a stream lead to poor performance, including liveness failures; it can lead to incorrect results and unpredictable behavior

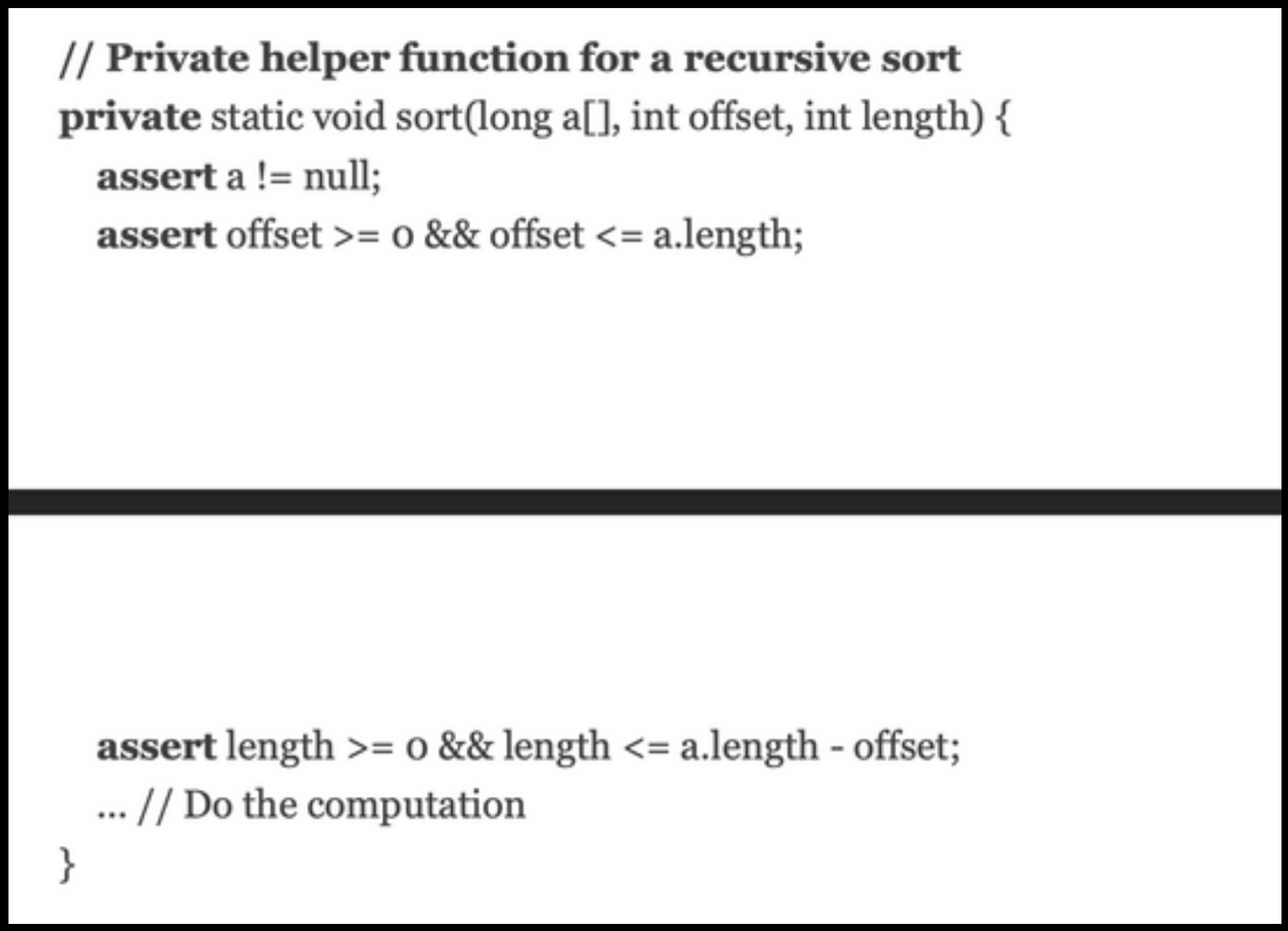
Under the right circumstances, it is possible to achieve near-linear speedup in the number of processor cores simply by adding a parallel call to a stream pipeline

```
static long pi(long n) {
   return LongStream.rangeClosed(2, n)
      .parallel()
      .mapToObj(BigInteger::valueOf)
      .filter(i -> i.isProbablePrime(50))
      .count();
}
```

Chapter 8. Methods

ITEM 49: CHECK PARAMETERS FOR V ALIDITY

The Objects.requireNonNull method, added in Java 7, is flexible and convenient, so there's no reason to perform null checks manually anymore.



```
// Private helper function for a recursive sort
private static void sort(long a[], int offset, int length) {
    assert a != null;
    assert offset >= 0 && offset <= a.length;

    assert length >= 0 && length <= a.length - offset;
    ... // Do the computation
}
```

ITEM 50: MAKE DEFENSIVE COPIES WHEN NEEDED

You must program defensively, with the assumption that clients of your class will do their best to destroy its invariants

```
// Broken "immutable" time period class
public final class Period {
   private final Date start;
   private final Date end;

   /**
    * @param  start the beginning of the period
    * @param  end the end of the period; must not precede start
    * @throws IllegalArgumentException if start is after end
    * @throws NullPointerException if start or end is null
    */
   public Period(Date start, Date end) {
      if (start.compareTo(end) > 0)
         throw new IllegalArgumentException(
            start + " after " + end);
      this.start = start;
      this.end   = end;
   }

   public Date start() {
      return start;
   }

   public Date end() {
      return end;
   }

   ...   // Remainder omitted
}
```

```
// Repaired constructor - makes defensive copies of
parameters
public Period(Date start, Date end) {
   this.start = new Date(start.getTime());
   this.end   = new Date(end.getTime());

   if (this.start.compareTo(this.end) > 0)
    throw new IllegalArgumentException(
       this.start + " after " + this.end);
}
```

defensive copies are made before checking the validity of the parameters (Item 49), and the validity check is performed on the copies rather than on the originals

do not use the clone method to make a defensive copy of a parameter whose type is subclassable by untrusted parties

```
public Date start() {
   return new Date(start.getTime());
}

public Date end() {
   return new Date(end.getTime());
}
```

ITEM 51: DESIGN METHOD SIGNATURES CAREFULLY

Choose method names carefully

Don’t go overboard in providing convenience methods.
When in doubt, leave it out. Avoid long parameter lists
Long sequences of identically typed parameters are especially harmful
For parameter types, favor interfaces over classes
Prefer two-element enum types to boolean parameters

ITEM 52: USE OVERLOADING JUDICIOUSLY

```
// Broken! - What does this program print?
public class CollectionClassifier {
   public static String classify(Set<?> s) {
      return "Set";
   }

   public static String classify(List<?> lst) {
      return "List";
   }

   public static String classify(Collection<?> c) {
      return "Unknown Collection";
   }

   public static void main(String[] args) {
      Collection<?>[] collections = {
         new HashSet<String>(),
         new ArrayList<BigInteger>(),
         new HashMap<String, String>().values()
      };

      for (Collection<?> c : collections)
         System.out.println(classify(c));
   }
}
```

Why does this happen? Because
the classifymethod is overloaded, and the choice of which overloading to invoke is made at compile time.

```
class Wine {
   String name() { return "wine"; }
}

class SparklingWine extends Wine {
   @Override String name() { return "sparkling wine"; }
}

class Champagne extends SparklingWine {
   @Override String name() { return "champagne"; }
}

public class Overriding {
   public static void main(String[] args) {
      List<Wine> wineList = List.of(

         new Wine(), new SparklingWine(), new Champagne());

      for (Wine wine : wineList)
         System.out.println(wine.name());
   }
}
```

ITEM 53: USE VARARGS JUDICIOUSLY

```
// Simple use of varargs
static int sum(int... args) {
    int sum = 0;
    for (int arg : args)
        sum += arg;
    return sum;
}

    // The WRONG way to use varargs to pass one or more
    arguments!
    static int min(int... args) {
        if (args.length == 0)
            throw new IllegalArgumentException("Too few arguments");
        int min = args[0];
        for (int i = 1; i < args.length; i++)
            if (args[i] < min)
                min = args[i];
        return min;
    }

// The right way to use varargs to pass one or more
arguments
static int min(int firstArg, int... remainingArgs) {
    int min = firstArg;
    for (int arg : remainingArgs)
        if (arg < min)
            min = arg;
    return min;
}
```

a single varargs method for use when the number of arguments exceeds three

Click here to view code image

```
public void foo() { }
public void foo(int a1) { }
public void foo(int a1, int a2) { }
public void foo(int a1, int a2, int a3) { }
public void foo(int a1, int a2, int a3, int... rest) { }
```

ITEM 54: RETURN EMPTY COLLECTIONS OR ARRAYS, NOT NULLS

```
// Returns null to indicate an empty collection. Don't do this!
private final List<Cheese> cheesesInStock = ...;

/**
 * @return a list containing all of the cheeses in the shop,
 *     or null if no cheeses are available for purchase.
 */
public List<Cheese> getCheeses() {
   return cheesesInStock.isEmpty() ? null
      : new ArrayList<>(cheesesInStock);
}

// Optimization - avoids allocating empty collections
public List<Cheese> getCheeses() {
   return cheesesInStock.isEmpty() ? Collections.emptyList()
      : new ArrayList<>(cheesesInStock);
}
```

```
//The right way to return a possibly empty array
public Cheese[] getCheeses() {
   return cheesesInStock.toArray(new Cheese[0]);
}
```

```
// Optimization - avoids allocating empty arrays
private static final Cheese[] EMPTY_CHEESE_ARRAY = new
Cheese[0];

public Cheese[] getCheeses() {
   return cheesesInStock.toArray(EMPTY_CHEESE_ARRAY);
}
```

// Don't do this - preallocating the array harms performance!

```
return cheesesInStock.toArray(new
Cheese[cheesesInStock.size()]);
```

never return null in place of an empty array or collection

ITEM 55: RETURN OPTIONALS JUDICIOUSLY

```
// Returns maximum value in collection - throws
exception if empty
```

```
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;
}

// Returns maximum value in collection as an
Optional<E>
public static <E extends Comparable<E>>
      Optional<E> max(Collection<E> c) {
   if (c.isEmpty())
      return Optional.empty();

   E result = null;
   for (E e : c)
      if (result == null || e.compareTo(result) > 0)
         result = Objects.requireNonNull(e);

   return Optional.of(result);
}
```

Never return a null value from an Optional-returning method

```
streamOfOptionals
   .filter(Optional::isPresent)
   .map(Optional::get)

streamOfOptionals.
   .flatMap(Optional::stream)
```

An optional that contains nothing is said to be empty. A value is said to be present in an optional that is not empty. An optional is essentially an immutable

collection that can hold at most one element. Optional<T> does not implement Collection<T>, but it could in principle.

```
Optional<String> checkForWord = Optional.ofNullable(sentence[9]);
```

```
import java.util.Optional;

class OptionalExample {
   public static void main(String[] args) {
      Optional<String> grade = Optional.of("B+");
      String opt1 = "Not null";
      String opt2 = null;
      System.out.println("Non-Empty Optional: " + grade);
      System.out.println("Value of Non-Empty Optional: " + grade.get());
      System.out.println("Empty Optional: " + Optional.empty());
```

```
    System.out.println("ofNullable on Non-Empty Optional: " +
Optional.ofNullable(opt1));
    System.out.println("ofNullable on Empty Optional: " +
Optional.ofNullable(opt2));
    if(Optional.ofNullable(opt1).isPresent()){
      System.out.println("of on Non-Empty Optional: " + Optional.of(opt1));
    }
    if(Optional.ofNullable(opt2).isPresent()){
      System.out.println("of on Empty Optional: " + Optional.of(opt2));
    }
  }
}
```

ITEM 56: WRITE DOC COMMENTS FOR ALL EXPOSED API ELEMENTS

To document your API properly, you must precede every exported class, interface, constructor, method, and field declaration with a doc comment

The doc comment for a method should describe succinctly the contract between the method and its client

```
/**
 * Returns the element at the specified position in this list.
 *
 * <p>This method is <i>not</i> guaranteed to run in constant
 * time. In some implementations it may run in time proportional
 * to the element position.
 *
 * @param  index index of element to return; must be
 *         non-negative and less than the size of this list
 * @return the element at the specified position in this list
 * @throws IndexOutOfBoundsException if the index is out of
range
 *         ({@code index < 0 || index >= this.size()})
 */
E get(int index);

/**
 * Returns true if this collection is empty.
 *
 * @implSpec
 * This implementation returns {@code this.size() == 0}.
 *
 * @return true if this collection is empty
 */
public boolean isEmpty() { ... }
```

doc comments should be readable both in the source code and in the generated documentation

Chapter 9. General Programming

ITEM 57: MINIMIZE THE SCOPE OF LOCAL VARIABLES

The most powerful technique for minimizing the scope of a local variable is to declare it where it is first used.

Nearly every local variable declaration should contain an initializer

```
Iterator<Element> i = c.iterator();
while (i.hasNext()) {
   doSomething(i.next());
}
...
Iterator<Element> i2 = c2.iterator();
while (i.hasNext()) {          // BUG!
   doSomethingElse(i2.next());
}
```

```
for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
   Element e = i.next();
   ... // Do something with e and i
}
```

```
// Compile-time error - cannot find symbol i
for (Iterator<Element> i2 = c2.iterator(); i.hasNext(); ) {
   Element e2 = i2.next();
   ... // Do something with e2 and i2
}
```

ITEM 58: PREFER FOR-EACH LOOPS TO TRADITIONAL FOR LOOPS

```
// Not the best way to iterate over an array!
for (int i = 0; i < a.length; i++) {
   ... // Do something with a[i]
}

// The preferred idiom for iterating over collections and
arrays
for (Element e : elements) {

  // Fixed, but ugly - you can do better!
  for (Iterator<Suit> i = suits.iterator(); i.hasNext(); ) {
     Suit suit = i.next();

    for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )
       deck.add(new Card(suit, j.next()));
 }
```

```
// Preferred idiom for nested iteration on collections and arrays
for (Suit suit : suits)
    for (Rank rank : ranks)
        deck.add(new Card(suit, rank));
```

Unfortunately, there are three common situations where you *can't* use for-each:

• **Destructive filtering**—If you need to traverse a collection removing selected elements, then you need to use an explicit iterator so that you can call its `remove` method. You can often avoid explicit traversal by using `Collection`'s `removeIf` method, added in Java 8.

• **Transforming**—If you need to traverse a list or array and replace some or all of the values of its elements, then you need the list iterator or array index in order to replace the value of an element.

• **Parallel iteration**—If you need to traverse multiple collections in parallel, then you need explicit control over the iterator or index variable so that all iterators or index variables can be advanced in lockstep (as demonstrated unintentionally in the buggy card and dice examples above).

```
public interface Iterable<E> {
  // Returns an iterator over the elements in this iterable
  Iterator<E> iterator();
}
```

ITEM 59: KNOW AND USE THE LIBRARIES

```
// Common but deeply flawed!
static Random rnd = new Random();

static int random(int n) {
    return Math.abs(rnd.nextInt()) % n;
}
public static void main(String[] args) {
  int n = 2 * (Integer.MAX_VALUE / 3);
  int low = 0;
  for (int i = 0; i < 1000000; i++)
    if (random(n) < n/2)
      low++;

  System.out.println(low);
}
```

every programmer should be familiar with the basics of java.lang, java.util, and java.io, and their subpackages

```
public static void main(String[] args) {
    final BigDecimal TEN_CENTS = new BigDecimal(".10");
    int itemsBought = 0;
    BigDecimal funds = new BigDecimal("1.00");
    for (BigDecimal price = TEN_CENTS;
         funds.compareTo(price) >= 0;
         price = price.add(TEN_CENTS)) {
      funds = funds.subtract(price);
      itemsBought++;
    }
    System.out.println(itemsBought + " items bought.");
    System.out.println("Money left over: $" + funds);
}
```

ITEM 61: PREFER PRIMITIVE TYPES TO BOXED PRIMITIVES

```
// Broken comparator - can you spot the flaw?
Comparator<Integer> naturalOrder =
    (i, j) -> (i < j) ? -1 : (i == j ? 0 : 1);
```

Applying the == operator to boxed primitives is almost always wrong.

```
Comparator<Integer> naturalOrder = (iBoxed, jBoxed) -> {
    int i = iBoxed, j = jBoxed; // Auto-unboxing
    return i < j ? -1 : (i == j ? 0 : 1);
};
```

Applying the == operator to boxed primitives is almost always wrong.

ITEM 62: AVOID STRINGS WHERE OTHER TYPES ARE MORE APPROPRIATE

Strings are poor substitutes for other value types

Strings are poor substitutes for enum types

Strings are poor substitutes for aggregate types

```
// Broken - inappropriate use of string as capability!
public class ThreadLocal {
    private ThreadLocal() { } // Noninstantiable

    // Sets the current thread's value for the named variable.
    public static void set(String key, Object value);

    // Returns the current thread's value for the named variable.
    public static Object get(String key);
}
```

Click here to view code image

```
public class ThreadLocal {
    private ThreadLocal() { }    // Noninstantiable

    public static class Key {    // (Capability)
        Key() { }
    `

   // Generates a unique, unforgeable key
   public static Key getKey() {
       return new Key();
   }

   public static void set(Key key, Object value);
   public static Object get(Key key);
}
```

```
public final class ThreadLocal {
    public ThreadLocal();
    public void set(Object value);
    public Object get();
}
```

```
public final class ThreadLocal<T> {
    public ThreadLocal();
    public void set(T value);
    public T get();
}
```

ITEM 63: BEWARE THE PERFORMANCE OF STRING CONCATENATION

```
// Inappropriate use of string concatenation - Performs
poorly!
public String statement() {
   String result = "";
   for (int i = 0; i < numItems(); i++)
      result += lineForItem(i);  // String concatenation
   return result;
}
```

The method performs abysmally if the number of items is large. **To achieve acceptable performance, use a `StringBuilder` in place of a `String`** to store the statement under construction:

```
public String statement() {
   StringBuilder b = new StringBuilder(numItems() *
LINE_WIDTH);
   for (int i = 0; i < numItems(); i++)
      b.append(lineForItem(i));
   return b.toString();
}
```

Don’t use the string concatenation operator to combine more than a few strings

ITEM 64: REFER TO OBJECTS BY THEIR INTERFACES

If appropriate interface types exist, then parameters, return values, variables, and fields should all be declared using interface types

```
// Good - uses interface as type
Set<Son> sonSet = new LinkedHashSet<>();
```

not this:

Click here to view code image

```
// Bad - uses class as type!
LinkedHashSet<Son> sonSet = new LinkedHashSet<>();
```

If you get into the habit of using interfaces as types, your program will be much more flexible

It is entirely appropriate to refer to an object by a class rather than an interface if no appropriate interface exists

If there is no appropriate interface, just use the least specific class in the class hierarchy that provides the required functionality.

ITEM 65: PREFER INTERFACES TO REFLECTION

You lose all the benefits of compile-time type checking

The code required to perform reflective access is clumsy and verbose

You can obtain many of the benefits of reflection while incurring few of its costs by using it only in a very limited form

```
// Reflective instantiation with interface access
public static void main(String[] args) {
   // Translate the class name into a Class object
   Class<? extends Set<String>> cl = null;
   try {
      cl = (Class<? extends Set<String>>)  // Unchecked cast!
           Class.forName(args[0]);
   } catch (ClassNotFoundException e) {
      fatalError("Class not found.");
   }
   // Get the constructor
   Constructor<? extends Set<String>> cons = null;
   try {
      cons = cl.getDeclaredConstructor();
   } catch (NoSuchMethodException e) {
      fatalError("No parameterless constructor");
   }
   // Instantiate the set
   Set<String> s = null;
   try {
      s = cons.newInstance();
   } catch (IllegalAccessException e) {
```

```
            fatalError("Constructor not accessible");
        } catch (InstantiationException e) {
            fatalError("Class not instantiable.");
        } catch (InvocationTargetException e) {
            fatalError("Constructor threw " + e.getCause());
        } catch (ClassCastException e) {
            fatalError("Class doesn't implement Set");
        }
        // Exercise the set
        s.addAll(Arrays.asList(args).subList(1, args.length));
        System.out.println(s);
    }
    private static void fatalError(String msg) {
        System.err.println(msg);
        System.exit(1);
    }
```

ITEM 66: USE NATIVE METHODS JUDICIOUSLY

It is rarely advisable to use native methods for improved performance

ITEM 67: OPTIMIZE JUDICIOUSLY

Strive to write good programs rather than fast ones

Strive to avoid design decisions that limit performance

Consider the performance consequences of your API design decisions

measure performance before and after each attempted optimization

ITEM 68: ADHERE TO GENERALLY ACCEPTED NAMING CONVENTIONS

Identifier Type	Examples
Package or module	`org.junit.jupiter.api`, `com.google.common.c`
Class or Interface	`Stream`, `FutureTask`, `LinkedHashMap`, `HttpClier`
Method or Field	`remove`, `groupingBy`, `getCrc`
Constant Field	`MIN_VALUE`, `NEGATIVE_INFINITY`
Local Variable	`i`, `denom`, `houseNum`
Type Parameter	`T`, `E`, `K`, `V`, `X`, `R`, `U`, `V`, `T1`, `T2`

Chapter 10. Exceptions

exceptions can improve a program's readability, reliability, and maintainability

ITEM 69: USE EXCEPTIONS ONLY FOR EXCEPTIONAL CONDITIONS

```
// Horrible abuse of exceptions. Don't ever do this!
try {
   int i = 0;
   while(true)
      range[i++].climb();
} catch (ArrayIndexOutOfBoundsException e) {
}
```

Exceptions are, as their name implies, to be used only for exceptional conditions; they should never be used for ordinary control flow.

A well-designed API must not force its clients to use exceptions for ordinary control flow

```
// Do not use this hideous code for iteration over a
collection!
try {
   Iterator<Foo> i = collection.iterator();
   while(true) {
      Foo foo = i.next();
      ...
   }
} catch (NoSuchElementException e) {
}
```

exceptions are designed for exceptional conditions. Don’t use them for ordinary control flow, and don’t write APIs that force others to do so.

ITEM 70: USE CHECKED EXCEPTIONS FOR RECOVERABLE CONDITIONS AND RUNTIME EXCEPTIONS FOR PROGRAMMING ERRORS

Java provides three kinds of throwables: checked exceptions, runtime exceptions, and errors

use checked exceptions for conditions from which the caller can reasonably be expected to recover

Use runtime exceptions to indicate programming errors

all of the unchecked throwables you implement should subclass RuntimeException (directly or indirectly)

ITEM 71: AVOID UNNECESSARY USE OF CHECKED EXCEPTIONS

```
} catch (TheCheckedException e) {
    throw new AssertionError(); // Can't happen!
}
```

```
} catch (TheCheckedException e) {
    e.printStackTrace();       // Oh well, we lose.
    System.exit(1);
}

// Invocation with checked exception
try {
    obj.action(args);
} catch (TheCheckedException e) {
    ... // Handle exceptional condition
}

// Invocation with state-testing method and unchecked exception
if (obj.actionPermitted(args)) {
    obj.action(args);
} else {
    ... // Handle exceptional condition
}
```

ITEM 72: FAVOR THE USE OF STANDARD EXCEPTIONS

Exception	Occasion for Use
`IllegalArgumentException`	Non-null parameter value is inappropriate
`IllegalStateException`	Object state is inappropriate for method invocat
`NullPointerException`	Parameter value is null where prohibited
`IndexOutOfBoundsException`	Index parameter value is out of range
`ConcurrentModificationException`	Concurrent modification of an object has been d prohibited
`UnsupportedOperationException`	Object does not support method

the rule is to throw IllegalStateException if no argument values would have worked, otherwise throw IllegalArgumentException.

ITEM 73: THROW EXCEPTIONS APPROPRIATE TO THE ABSTRACTION

higher layers should catch lower-level exceptions and, in their place, throw exceptions that can be explained in terms of the higher-level abstraction.This idiom is known as exception translation:

```
// Exception Translation
try {
   ... // Use lower-level abstraction to do our bidding
} catch (LowerLevelException e) {


       throw new HigherLevelException(...);
    }
/**
 * Returns the element at the specified position in this list.
 * @throws IndexOutOfBoundsException if the index is out of
range
 *         ({@code index <  0 || index >= size()}).
 */
public E get(int index) {
   ListIterator<E> i = listIterator(index);
   try {
      return i.next();
   } catch (NoSuchElementException e) {
      throw new IndexOutOfBoundsException("Index: " + index);
   }
}
```

```
// Exception Chaining
try {
   ... // Use lower-level abstraction to do our bidding
} catch (LowerLevelException cause) {
   throw new HigherLevelException(cause);
}
```

The higher-level exception's constructor passes the cause to a *chaining-aware* superclass constructor, so it is ultimately passed to one of `Throwable`'s chaining-aware constructors, such as `Throwable(Throwable)`:

Click here to view code image

```
// Exception with chaining-aware constructor
class HigherLevelException extends Exception {
   HigherLevelException(Throwable cause) {
      super(cause);
   }
}
```

While exception translation is superior to mindless propagation of exceptions from lower layers, it should not be overused

ITEM 74: DOCUMENT ALL EXCEPTIONS THROWN BY EACH METHOD

Always declare checked exceptions individually, and document precisely the conditions under which each one is thrown using the Javadoc @throws tag

Use the Javadoc @throws tag to document each exception that a method can throw, but do not use the throws keyword on unchecked exceptions

If an exception is thrown by many methods in a class for the same reason, you can document the exception in the class's documentation comment rather than documenting it individually for each method

ITEM 75: INCLUDE FAILURE-CAPTURE INFORMATION IN DETAIL MESSAGES

To capture a failure, the detail message of an exception should contain the values of all parameters and fields that contributed to the exception

do not include passwords, encryption keys, and the like in detail messages

```
/**
* Constructs an IndexOutOfBoundsException.
*
* @param lowerBound the lowest legal index value
* @param upperBound the highest legal index value plus one
* @param index the actual index value
*/
```

```
public IndexOutOfBoundsException(int lowerBound, int
upperBound,
                                 int index) {
  // Generate a detail message that captures the failure
  super(String.format(
      "Lower bound: %d, Upper bound: %d, Index: %d",
      lowerBound, upperBound, index));

  // Save failure information for programmatic access
  this.lowerBound = lowerBound;
  this.upperBound = upperBound;
  this.index = index;
}
```

ITEM 76: STRIVE FOR FAILURE ATOMICITY

After an object throws an exception, it is generally desirable that the object still be in a well-defined, usable state, even if the failure occurred in the midst of performing an operation. This is especially true for checked exceptions, from which the caller is expected to recover. Generally speaking, a failed method invocation should leave the object in the state that it was in prior to the invocation. A method with this property is said to
be failure-atomic.

```
public Object pop() {
  if (size == 0)
```

```
    throw new EmptyStackException();
  Object result = elements[--size];
  elements[size] = null; // Eliminate obsolete reference
  return result;
}
```

any generated exception that is part of a method's specification should leave the object in the same state it was in prior to the method invocation. Where this rule is violated, the API documentation should clearly indicate what state the object will be left in. Unfortunately, plenty of existing API documentation fails to live up to this ideal.

ITEM 77: DON'T IGNORE EXCEPTIONS

```
// Empty catch block ignores exception - Highly suspect!
try {
   ...
} catch (SomeException e) {
}
```

An empty catch block defeats the purpose of exceptions, which is to force you to handle exceptional conditions.

If you choose to ignore an exception, the catch block should contain a comment explaining why it is appropriate to do so, and the variable should be named ignored:

```
Future<Integer> f = exec.submit(planarMap::chromaticNumber);
int numColors = 4; // Default; guaranteed sufficient for any map
try {
   numColors = f.get(1L, TimeUnit.SECONDS);
} catch (TimeoutException | ExecutionException ignored) {
   // Use default: minimal coloring is desirable, not
required
}
```

Chapter 11. Concurrency

THREADS allow multiple activities to proceed concurrently. Concurrent programming is harder than single-threaded programming, because more things can go wrong, and failures can be hard to reproduce.

ITEM 78: SYNCHRONIZE ACCESS TO SHARED MUTABLE DATA

The synchronized keyword ensures that only a single thread can execute a method or block at one time. Many programmers think of synchronization solely as a means of mutual exclusion, to prevent an object from being seen in an inconsistent state by one thread while it’s being modified by another

an object is created in a consistent state and locked by the methods that access it. These methods observe the state and optionally cause a state transition, transforming the object from one consistent state to another. Proper use of synchronization guarantees that no method will ever observe the object in an inconsistent state.

Synchronization is required for reliable communication between threads as well as for mutual exclusion.

```
// Broken! - How long would you expect this program to run?
public class StopThread {
  private static boolean stopRequested;

  public static void main(String[] args)
      throws InterruptedException {
    Thread backgroundThread = new Thread(() -> {
      int i = 0;
      while (!stopRequested)
        i++;
    });
    backgroundThread.start();

    TimeUnit.SECONDS.sleep(1);
    stopRequested = true;
  }
}
```

the absence of synchronization, it's quite acceptable for the virtual machine to transform this code:

```
while (!stopRequested)
    i++;
```

into this code:

```
if (!stopRequested)
    while (true)
        i++;
```

This optimization is known as hoisting, and it is precisely what the OpenJDK Server VM does. The result is a liveness failure: the program fails to make progress. One way to fix the problem is to synchronize access to the stopRequested field.

```
// Properly synchronized cooperative thread termination
public class StopThread {
    private static boolean stopRequested;

    private static synchronized void requestStop() {
        stopRequested = true;
    }

    private static synchronized boolean stopRequested() {
        return stopRequested;
    }
```

```
public static void main(String[] args)
      throws InterruptedException {
   Thread backgroundThread = new Thread(() -> {

         int i = 0;
         while (!stopRequested())
            i++;
      });
      backgroundThread.start();

      TimeUnit.SECONDS.sleep(1);
      requestStop();
   }
}
```

Synchronization is not guaranteed to work unless both read and write operations are synchronized

```
// Cooperative thread termination with a volatile field
public class StopThread {


  private static volatile boolean stopRequested;
```

```
    public static void main(String[] args)
            throws InterruptedException {
        Thread backgroundThread = new Thread(() -> {
            int i = 0;
            while (!stopRequested)
                i++;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}
```

```
// Broken - requires synchronization!
private static volatile int nextSerialNumber = 0;

public static int generateSerialNumber() {
    return nextSerialNumber++;
}
```

A volatile keyword is used to modify the value of a variable by different threads. It is also used to make classes thread-safe. It means that multiple threads can use a method and instance of the classes at the same time without any problem. The volatile keyword can be used either with primitive type or objects

```
// Lock-free synchronization with
java.util.concurrent.atomic
private static final AtomicLong nextSerialNum = new
AtomicLong();

public static long generateSerialNumber() {

 return nextSerialNum.getAndIncrement();
}
```

confine mutable data to a single thread

when multiple threads share mutable data, each thread that reads or writes the data must perform synchronization

ITEM 79: AVOID EXCESSIVE SYNCHRONIZATION

To avoid liveness and safety failures, never cede control to the client within a synchronized method or block

```
public class ObservableSet<E> extends ForwardingSet<E> {
  public ObservableSet(Set<E> set) { super(set); }
private final List<SetObserver<E>> observers
      = new ArrayList<>();

public void addObserver(SetObserver<E> observer) {
   synchronized(observers) {
      observers.add(observer);
   }
}

public boolean removeObserver(SetObserver<E> observer) {
   synchronized(observers) {
      return observers.remove(observer);
   }
}

private void notifyElementAdded(E element) {
   synchronized(observers) {
      for (SetObserver<E> observer : observers)
         observer.added(this, element);
   }
}
```

```
@Override public boolean add(E element) {
   boolean added = super.add(element);
   if (added)
      notifyElementAdded(element);
   return added;
}

@Override public boolean addAll(Collection<? extends E> c) {
   boolean result = false;
   for (E element : c)

         result |= add(element);  // Calls notifyElementAdded
      return result;
    }
 }
 @FunctionalInterface public interface SetObserver<E> {
    // Invoked when an element is added to the observable set
    void added(ObservableSet<E> set, E element);
 }
public static void main(String[] args) {
   ObservableSet<Integer> set =
         new ObservableSet<>(new HashSet<>());

   set.addObserver((s, e) -> System.out.println(e));
```

```
    for (int i = 0; i < 100; i++)
        set.add(i);
}
set.addObserver(new SetObserver<>() {
    public void added(ObservableSet<Integer> s, Integer e) {
        System.out.println(e);
        if (e == 23)
            s.removeObserver(this);
    }
});

// Observer that uses a background thread needlessly
set.addObserver(new SetObserver<>() {
  public void added(ObservableSet<Integer> s, Integer e) {
    System.out.println(e);
    if (e == 23) {
        ExecutorService exec =
            Executors.newSingleThreadExecutor();
        try {
          exec.submit(() -> s.removeObserver(this)).get();
        } catch (ExecutionException | InterruptedException ex) {
          throw new AssertionError(ex);
        } finally {
          exec.shutdown();
        }
    }
  }
});
```

```
// Alien method moved outside of synchronized block - open calls
private void notifyElementAdded(E element) {
   List<SetObserver<E>> snapshot = null;
   synchronized(observers) {
      snapshot = new ArrayList<>(observers);
   }
   for (SetObserver<E> observer : snapshot)
      observer.added(this, element);
}
```

```
// Thread-safe observable set with CopyOnWriteArrayList
private final List<SetObserver<E>> observers =
       new CopyOnWriteArrayList<>();

 public void addObserver(SetObserver<E> observer) {
    observers.add(observer);
 }

 public boolean removeObserver(SetObserver<E> observer) {
    return observers.remove(observer);
 }

 private void notifyElementAdded(E element) {
    for (SetObserver<E> observer : observers)
       observer.added(this, element);
 }
```

As a rule, you should do as little work as possible inside synchronized regions

ITEM 80: PREFER EXECUTORS, TASKS, AND STREAMS TO THREADS

```
ExecutorService exec = Executors.newSingleThreadExecutor();

exec.execute(runnable);
exec.shutdown();
```

ITEM 81: PREFER CONCURRENCY UTILITIES TO WAIT AND NOTIFY

Given the difficulty of using wait and notify correctly, you should use the higher-level concurrency utilities instead.

it is impossible to exclude concurrent activity from a concurrent collection; locking it will only slow the program.

```
// Concurrent canonicalizing map atop ConcurrentMap -
not optimal
private static final ConcurrentMap<String, String> map =
      new ConcurrentHashMap<>();

public static String intern(String s) {
   String previousValue = map.putIfAbsent(s, s);
   return previousValue == null ? s : previousValue;
}
```

```
// Concurrent canonicalizing map atop ConcurrentMap - faster!
public static String intern(String s) {
   String result = map.get(s);
   if (result == null) {
      result = map.putIfAbsent(s, s);
      if (result == null)
         result = s;
   }
   return result;
}
```

```
// Simple framework for timing concurrent execution
public static long time(Executor executor, int concurrency,
        Runnable action) throws InterruptedException {
    CountDownLatch ready = new CountDownLatch(concurrency);
    CountDownLatch start = new CountDownLatch(1);
    CountDownLatch done  = new CountDownLatch(concurrency);

    for (int i = 0; i < concurrency; i++) {
        executor.execute(() -> {
            ready.countDown(); // Tell timer we're ready
            try {
                start.await(); // Wait till peers are ready
                action.run();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            } finally {
                done.countDown();  // Tell timer we're done
            }
        });
    }
```

```
    ready.await();     // Wait for all workers to be ready
    long startNanos = System.nanoTime();
    start.countDown(); // And they're off!
    done.await();      // Wait for all workers to finish
    return System.nanoTime() - startNanos;
}
```

For interval timing, always use System.nanoTime rather than System.currentTimeMillis. System.nanoTime is both more accurate and more precise and is unaffected by adjustments to the system's real-time clock.

```
// The standard idiom for using the wait method
synchronized (obj) {
   while (<condition does not hold>)
      obj.wait(); // (Releases lock, and reacquires on wakeup)
```

Always use the wait loop idiom to invoke the wait method; never invoke it outside of a loop.

There is seldom, if ever, a reason to use wait and notify in new code

ITEM 82: DOCUMENT THREAD SAFETY

The presence of the synchronized modifier in a method declaration is an implementation detail, not a part of its API.

To enable safe concurrent use, a class must clearly document what level of thread safety it supports.

Immutable Conditionally thread-safe Thread-hostile

Unconditionally thread-safe Not thread-safe

```
Map<K, V> m = Collections.synchronizedMap(new HashMap<>());
Set<K> s = m.keySet();  // Needn't be in synchronized block
   ...
synchronized(m) {  // Synchronizing on m, not s!
   for (K key : s)
      key.f();
}
```

To prevent this denial-of-service attack, you can use a private lock object instead of using synchronized methods

```
// Private lock object idiom - thwarts denial-of-service
attack
private final Object lock = new Object();

public void foo() {
   synchronized(lock) {
      ...
   }
}
```

Lock fields should always be declaredfinal.

ITEM 83: USE LAZY INITIALIZATION JUDICIOUSLY

Lazy initialization is the act of delaying the initialization of a field until its value is needed. If the value is never needed, the field is never initialized

Under most circumstances, normal initialization is preferable to lazy initialization.

If you use lazy initialization to break an initialization circularity, use a synchronized accessor because it is the simplest

```
private FieldType field;

private synchronized FieldType getField() {
   if (field == null)
      field = computeFieldValue();
   return field;
}
```

If you need to use lazy initialization for performance on a static field, use the lazy initialization holder class idiom.

```
// Lazy initialization holder class idiom for static fields
private static class FieldHolder {
   static final FieldType field = computeFieldValue();
}

private static FieldType getField() { return FieldHolder.field; }
```

```
// Double-check idiom for lazy initialization of instance fields
private volatile FieldType field;

private FieldType getField() {
   FieldType result = field;
   if (result == null) {  // First check (no locking)
      synchronized(this) {
         if (field == null)  // Second check (with locking)
            field = result = computeFieldValue();
      }
   }
   return result;
}
```

```
// Single-check idiom - can cause repeated initialization!
private volatile FieldType field;

private FieldType getField() {
   FieldType result = field;
   if (result == null)
      field = result = computeFieldValue();
   return result;
}
```

ITEM 84: DON’T DEPEND ON THE THREAD SCHEDULER

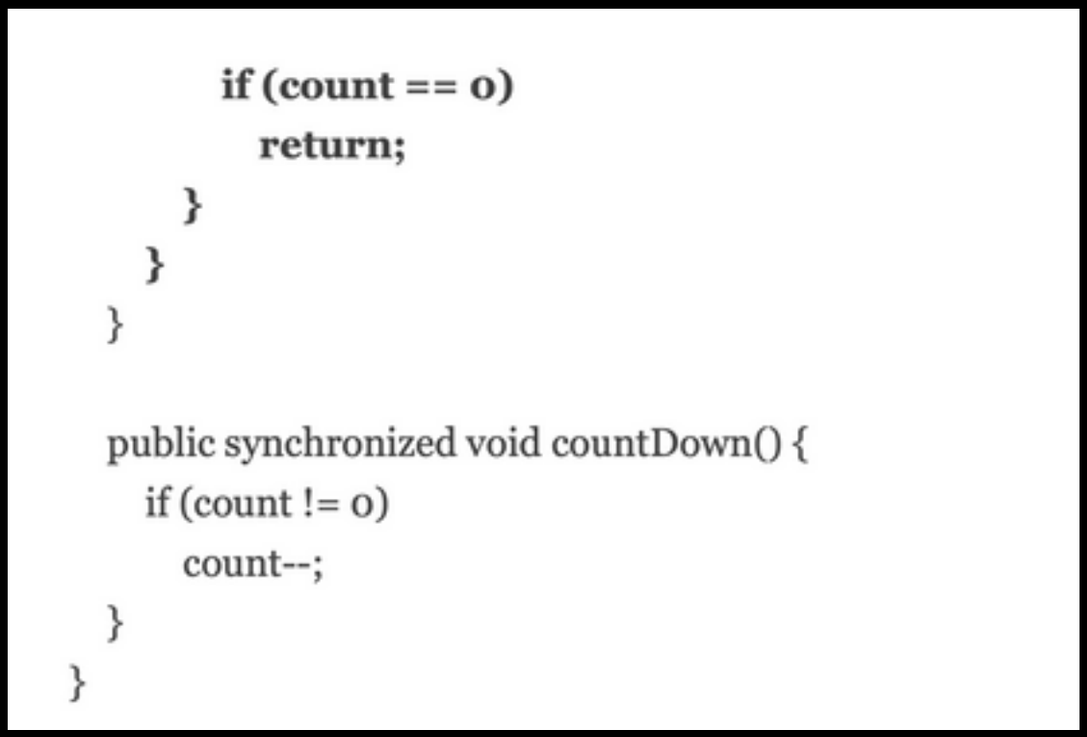
Any program that relies on the thread scheduler for correctness or performance is likely to be nonportable.

Threads should not run if they aren’t doing useful work.

```
// Awful CountDownLatch implementation - busy-waits
incessantly!
public class SlowCountDownLatch {
   private int count;

   public SlowCountDownLatch(int count) {
      if (count < 0)
         throw new IllegalArgumentException(count + " < 0");
      this.count = count;
   }

   public void await() {
      while (true) {
         synchronized(this) {
```



```
            if (count == 0)
                return;
        }
    }
}

public synchronized void countDown() {
    if (count != 0)
        count--;
}
}
```

resist the temptation to “fix” the program by putting in calls to Thread.yield

Thread.yield has no testable semantics

Thread priorities are among the least portable features of Java

Chapter 12. Serialization

THIS chapter concerns object serialization, which is Java’s framework for encoding objects as byte streams (serializing) and reconstructing objects from their encodings (deserializing). Once an object has been serialized, its encoding can be sent from one VM to another or stored on disk for later deserialization

ITEM 85: PREFER ALTERNATIVES TO JAVA SERIALIZATION

```
// Deserialization bomb - deserializing this stream takes forever
static byte[] bomb() {
    Set<Object> root = new HashSet<>();
    Set<Object> s1 = root;
    Set<Object> s2 = new HashSet<>();
    for (int i = 0; i < 100; i++) {
        Set<Object> t1 = new HashSet<>();
        Set<Object> t2 = new HashSet<>();
        t1.add("foo"); // Make t1 unequal to t2
        s1.add(t1);  s1.add(t2);
        s2.add(t1);  s2.add(t2);
        s1 = t1;
        s2 = t2;
    }
    return serialize(root); // Method omitted for brevity
}
```

The best way to avoid serialization exploits is never to deserialize anything

here is no reason to use Java serialization in any new system you write.

never deserialize untrusted data

Accepting classes by default and rejecting a list of potentially dangerous ones is known as blacklisting; rejecting classes by default and accepting a list of those that are presumed safe is known as whitelisting. Prefer whitelisting to blacklisting, as blacklisting only protects you against known threats. A tool called Serial Whitelist Application Trainer (SWAT) can be used to automatically prepare a whitelist for your application

ITEM 86: IMPLEMENT SERIALIZABLE WITH GREAT CAUTION

A major cost of implementing Serializable is that it decreases the flexibility to change a class's implementation once it has been released

A second cost of implementing Serializable is that it increases the likelihood of bugs and security holes (Item 85).

A third cost of implementing Serializable is that it increases the testing burden associated with releasing a new version of a class

Implementing Serializable is not a decision to be undertaken lightly

Classes designed for inheritance (Item 19) should rarely implement Serializable, and interfaces should rarely extend it.

```
// readObjectNoData for stateful extendable serializable
classes
private void readObjectNoData() throws InvalidObjectException {
   throw new InvalidObjectException("Stream data required");
}
```

Inner classes (Item 24) should not implement Serializable.

ITEM 87: CONSIDER USING A CUSTOM SERIALIZED FORM

Do not accept the default serialized form without first considering whether it is appropriate.

The default serialized form is likely to be appropriate if an object's physical representation is identical to its logical content.

```
// Good candidate for default serialized form
public class Name implements Serializable {
   /**
      * Last name. Must be non-null.
      * @serial
      */
   private final String lastName;
```

```
    /**
     * First name. Must be non-null.
     * @serial
     */
    private final String firstName;
    /**
     * Middle name, or null if there is none.
     * @serial
     */
    private final String middleName;

    ... // Remainder omitted
}
```

Even if you decide that the default serialized form is appropriate, you often must provide a readObject method to ensure invariants and security.

```
// Awful candidate for default serialized form
public final class StringList implements Serializable {
    private int size = 0;
    private Entry head = null;
```

```
    private static class Entry implements Serializable {
        String data;
        Entry  next;
        Entry  previous;
    }

    ... // Remainder omitted
}
```

Using the default serialized form when an object's physical representation differs substantially from its logical data content has four disadvantages:

It permanently ties the exported API to the current internal representation

It can consume excessive space

It can consume excessive time

It can cause stack overflows

```
// StringList with a reasonable custom serialized form
public final class StringList implements Serializable {
    private transient int size   = 0;
    private transient Entry head = null;
```

```
// No longer Serializable!
private static class Entry {
    String data;
    Entry  next;
    Entry  previous;
}

// Appends the specified string to the list
public final void add(String s) { ... }

/**
 * Serialize this {@code StringList} instance.
 *
 * @serialData The size of the list (the number of strings
 * it contains) is emitted ({@code int}), followed by all of
 * its elements (each a {@code String}), in the proper
 * sequence.
 */
private void writeObject(ObjectOutputStream s)
        throws IOException {
    s.defaultWriteObject();
    s.writeInt(size);

    // Write out all elements in the proper order.
```

```
    for (Entry e = head; e != null; e = e.next)
        s.writeObject(e.data);
}

private void readObject(ObjectInputStream s)
        throws IOException, ClassNotFoundException {
    s.defaultReadObject();
    int numElements = s.readInt();

    // Read in all elements and insert them in list
    for (int i = 0; i < numElements; i++)
        add((String) s.readObject());
}

... // Remainder omitted
}
```

Before deciding to make a field nontransient, convince yourself that its value is part of the logical state of the object

you must impose any synchronization on object serialization that you would impose on any other method that reads the entire state of the object

Regardless of what serialized form you choose, declare an explicit serial version UID in every serializable class you write

```
// writeObject for synchronized class with default
serialized form
private synchronized void writeObject(ObjectOutputStream s)
      throws IOException {
   s.defaultWriteObject();
}
```

Do not change the serial version UID unless you want to break compatibility with all existing serialized instances of a class.

ITEM 88: WRITE READOBJECT METHODS DEFENSIVELY

```
// Immutable class that uses defensive copying
public final class Period {
   private final Date start;
   private final Date end;
   /**
    * @param  start the beginning of the period
    * @param  end the end of the period; must not precede start
    * @throws IllegalArgumentException if start is after end
    * @throws NullPointerException if start or end is null
    */
   public Period(Date start, Date end) {
```

```
        this.start = new Date(start.getTime());
        this.end   = new Date(end.getTime());
        if (this.start.compareTo(this.end) > 0)
            throw new IllegalArgumentException(
                    start + " after " + end);
    }

    public Date start () { return new Date(start.getTime()); }

    public Date end () { return new Date(end.getTime()); }

    public String toString() { return start + " - " + end; }

    ... // Remainder omitted
}
```

```
public class BogusPeriod {
  // Byte stream couldn't have come from a real Period
instance!
  private static final byte[] serializedForm = {
    (byte)0xac, (byte)0xed, 0x00, 0x05, 0x73, 0x72, 0x00, 0x06,
    0x50, 0x65, 0x72, 0x69, 0x6f, 0x64, 0x40, 0x7e, (byte)0xf8,
    0x2b, 0x4f, 0x46, (byte)0xc0, (byte)0xf4, 0x02, 0x00, 0x02,
    0x4c, 0x00, 0x03, 0x65, 0x6e, 0x64, 0x74, 0x00, 0x10, 0x4c,
```

```
    public static void main(String[] args) {
      Period p = (Period) deserialize(serializedForm);
      System.out.println(p);
    }

    // Returns the object with the specified serialized form
    static Object deserialize(byte[] sf) {
      try {
        return new ObjectInputStream(
            new ByteArrayInputStream(sf)).readObject();
      } catch (IOException | ClassNotFoundException e) {
        throw new IllegalArgumentException(e);
      }
    }
  }
// readObject method with validity checking - insufficient!
private void readObject(ObjectInputStream s)
      throws IOException, ClassNotFoundException {
  s.defaultReadObject();

  // Check that our invariants are satisfied
  if (start.compareTo(end) > 0)
    throw new InvalidObjectException(start +" after "+ end);
}
```

```
public class MutablePeriod {
   // A period instance
   public final Period period;

   // period's start field, to which we shouldn't have access
   public final Date start;

   // period's end field, to which we shouldn't have access
   public final Date end;
public MutablePeriod() {
   try {
      ByteArrayOutputStream bos =
         new ByteArrayOutputStream();
      ObjectOutputStream out =
         new ObjectOutputStream(bos);

      // Serialize a valid Period instance
      out.writeObject(new Period(new Date(), new Date()));

      /*
       * Append rogue "previous object refs" for internal
       * Date fields in Period. For details, see "Java
       * Object Serialization Specification," Section 6.4.
       */
      byte[] ref = { 0x71, 0, 0x7e, 0, 5 };  // Ref #5
      bos.write(ref); // The start field
      ref[4] = 4;      // Ref # 4
      bos.write(ref); // The end field
```

```
            // Deserialize Period and "stolen" Date references
            ObjectInputStream in = new ObjectInputStream(
               new ByteArrayInputStream(bos.toByteArray()));
            period = (Period) in.readObject();
            start  = (Date)   in.readObject();
            end    = (Date)   in.readObject();
          } catch (IOException | ClassNotFoundException e) {
            throw new AssertionError(e);
          }
        }
      }

public static void main(String[] args) {
    MutablePeriod mp = new MutablePeriod();
    Period p = mp.period;
    Date pEnd = mp.end;

    // Let's turn back the clock
    pEnd.setYear(78);
    System.out.println(p);

    // Bring back the 60s!
    pEnd.setYear(69);
    System.out.println(p);
}
```

```
// readObject method with defensive copying and validity
checking
private void readObject(ObjectInputStream s)
      throws IOException, ClassNotFoundException {
   s.defaultReadObject();

   // Defensively copy our mutable components
   start = new Date(start.getTime());
   end    = new Date(end.getTime());

   // Check that our invariants are satisfied
   if (start.compareTo(end) > 0)
      throw new InvalidObjectException(start +" after "+ end);
}
```

ITEM 89: FOR INSTANCE CONTROL, PREFER ENUM TYPES TO READRESOLVE

```
public class Elvis {
   public static final Elvis INSTANCE = new Elvis();
   private Elvis() {  ... }

   public void leaveTheBuilding() { ... }
}
```

```
// readResolve for instance control - you can do better!
private Object readResolve() {
   // Return the one true Elvis and let the garbage collector
   // take care of the Elvis impersonator.
   return INSTANCE;
}
```

if you depend on readResolve for instance control, all instance fields with object reference types must be declared transient.

```
// Broken singleton - has nontransient object reference field!
public class Elvis implements Serializable {
   public static final Elvis INSTANCE = new Elvis();
   private Elvis() { }

   private String[] favoriteSongs =
     { "Hound Dog", "Heartbreak Hotel" };
   public void printFavorites() {
     System.out.println(Arrays.toString(favoriteSongs));

      }


      private Object readResolve() {
         return INSTANCE;
      }
   }
```

```
public class ElvisStealer implements Serializable {
    static Elvis impersonator;
    private Elvis payload;

    private Object readResolve() {
        // Save a reference to the "unresolved" Elvis instance
        impersonator = payload;

        // Return object of correct type for favoriteSongs field
        return new String[] { "A Fool Such as I" };
    }
    private static final long serialVersionUID = 0;
}
    // Enum singleton - the preferred approach
    public enum Elvis {
        INSTANCE;
        private String[] favoriteSongs =
            { "Hound Dog", "Heartbreak Hotel" };
        public void printFavorites() {
            System.out.println(Arrays.toString(favoriteSongs));
        }
    }
```

The accessibility of readResolve is significant

ITEM 90: CONSIDER SERIALIZATION PROXIES INSTEAD OF SERIALIZED INSTANCES

The serialization proxy pattern is reasonably straightforward. First, design a private static nested class that concisely represents the logical state of an instance of the enclosing class. This nested class is known as the serialization proxy of the enclosing class

```
// Serialization proxy for Period class
private static class SerializationProxy implements Serializable {
    private final Date start;
    private final Date end;

    SerializationProxy(Period p) {
        this.start = p.start;
        this.end = p.end;
    }

    private static final long serialVersionUID =
        234098243823485285L; // Any number will do (Item  87)
}

// writeReplace method for the serialization proxy
pattern
private Object writeReplace() {
    return new SerializationProxy(this);
}
```

```
// readObject method for the serialization proxy pattern
private void readObject(ObjectInputStream stream)
        throws InvalidObjectException {
    throw new InvalidObjectException("Proxy required");
}
// readResolve method for Period.SerializationProxy
private Object readResolve() {
    return new Period(start, end);    // Uses public constructor
}
// EnumSet's serialization proxy
private static class SerializationProxy <E extends Enum<E>>
        implements Serializable {
    // The element type of this enum set.
    private final Class<E> elementType;

    // The elements contained in this enum set.
    private final Enum<?>[] elements;

    SerializationProxy(EnumSet<E> set) {
        elementType = set.elementType;
        elements = set.toArray(new Enum<?>[0]);
    }

    private Object readResolve() {
        EnumSet<E> result = EnumSet.noneOf(elementType);
        for (Enum<?> e : elements)
            result.add((E)e);
        return result;
    }
    private static final long serialVersionUID =
        362491234563181265L;
}
```