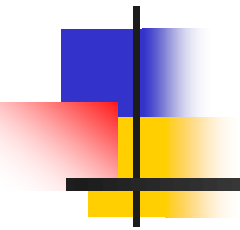
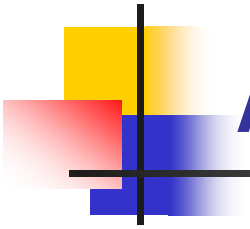


Effective Java, Chapter 4

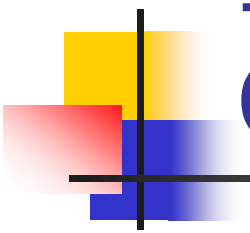
Classes and Interfaces



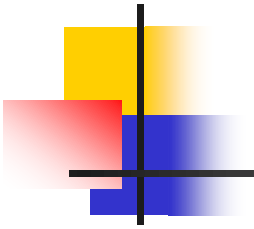
Agenda

- Material From Joshua Bloch
  - Effective Java: Programming Language Guide
- Cover Parts of Items 15 through 24
  - "Classes and Interfaces" Chapter
- Moral:
  - Inheritance requires careful programming

Item 15: Minimize Accessibility of Classes and Members

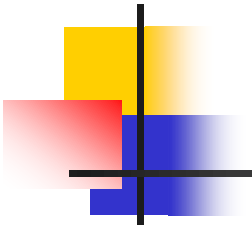


- Information hiding /encapsulation
  - Most important factor to distinguish well or poorly designed module
  - Hide the implementation detail
  - Modules are communicated through API



More on Item 15

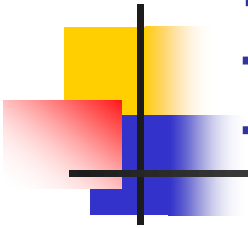
- Reasons for Information Hiding
  - Decouples modules
  - Allows isolated development and maintenance
    - Developed/tested/optimized/used/understood /modified in isolation
- Java has an *access control mechanism* to accomplish this goal



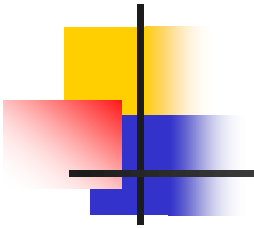
Accessibility Levels

- Standard list of accessibility levels
  - private
  - package-private (aka package friendly)
  - protected
  - public
- Huge difference between 2nd and 3rd
  - package-private: part of implementation
  - protected: part of public API

Make Each Class or Member as Inaccessible as Possible

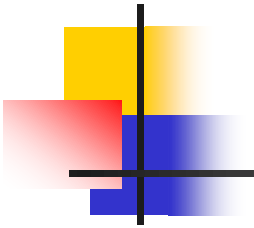


- For top-level classes/interfaces
  - Only *package-private* and *public* are applicable
  - Always favor *package-private* if possible
- If a package-private top-level class/interface is used by only one class
  - Consider making it a *private* nested class
- Choose the *less inaccessible level* if you are not sure



More on access level

- Accessibility of methods which overrides a superclass method
  - Not permitted to have a lower access level.
  - Note: all members of an ***interface*** are implicitly public, thus its overridden method in subclass must be public.
- Don't raise the level of accessibility just to facilitate testing



Module System in Java 9

- A module is a grouping of packages
- 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 public classes in unexported packages give rise to the two implicit access levels

Item 16 Use Accessors, Not Public Fields

- Avoid code such as:

```
Public Class Person { public String name; public int age; }
```

  - No possibility of encapsulation
  - No chance to check validity
  - public mutable fields are **not** thread safe

- Use get/set methods instead:

```
public double getX() { return x; }
public void setX(double x) { this.x = x}
```

- Advice holds for immutable fields as well
  - Limits possible ways for class to evolve

Example: Questionable Use of Immutable Public Fields

```
public final class Time {
   private static final int HOURS_PER_DAY = 24;
   private static final int MINUTES_PER_HOUR = 60;

   public final int hour;     // Not possible to change rep for class
   public final int minute;

   // But we can check invariants, since fields are final
   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("Minute: " + minute);
      this.hour = hour;
      this.minute = minute;
   }
   // Remainder omitted
}
```

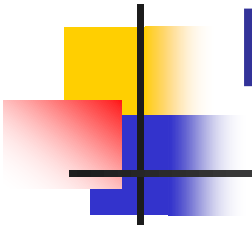
Code example: violation of this rule in Java platform lib

```
package java.awt;

// ...

public class Dimension extends Dimension2D
    implements java.io.Serializable {

    /**
     * The width dimension; negative values can be used.
     */
    public int width;

    /**
     * The height dimension; negative values can be used.
     */
    public int height;

    // ...

    public Dimension getSize() {
    return new Dimension(width, height);
    }

    //...
}
```

Not possible to make the fields immutable later.

```
package java.awt;

// ...

public class Point extends Point2D
    implements java.io.Serializable {
    /**
     * The X coordinate of this <code>Point</code>.
     * If no X coordinate is set it will default to 0.
     */
    public int x;

    /**
     * The Y coordinate of this <code>Point</code>.
     * If no Y coordinate is set it will default to 0.
     */
    public int y;

    // ...
}
```

Exception: Exposing Constants



- Public constants must contain either *primitive values* or references to *immutable objects*
- Wrong – Potential Security Hole:

```
public static final Type[] VALUES = {…};
```

- Problem:
  - `VALUES` is final; entries in `VALUES` are not!

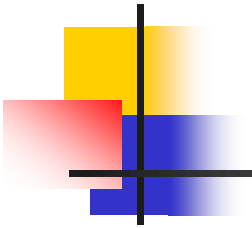
Exposing Constants - Solutions

- Correct:

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

- Also Correct:

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



Item 17: Minimize Mutability

- An object is considered *immutable* if its state cannot change after it is constructed.
  - e.g String, Integer, BigInteger, BigDecimal
- Reasons to make a class immutable:
  1. Thread safe inherently
  2. Can be shared freely
  3. No need to make defensive copy
  4. Its internals can also be shared
  5. Great building blocks for other object
     - Great to be map keys and set elements

Code example: a mutable class

```
public class SynchronizedRGB {

    private int red;
    private int green;
    private int blue;
    private String name;

    public SynchronizedRGB(int red,
                           int green,
                           int blue,
                           String name) {
        this.red = red;
        this.green = green;
        this.blue = blue;
        this.name = name;
    }

    public void set(int red,
                    int green,
                    int blue,
                    String name) {
        synchronized (this) {
            this.red = red;
            this.green = green;
            this.blue = blue;
            this.name = name;
        }
    }
```

The class, SynchronizedRGB, defines objects that represent colors. Each object represents the color as three integers that stand for primary color values and a string that gives the name of the color.

```
    public synchronized int getRGB() {
        return ((red << 16) | (green << 8) | blue);
    }

    public synchronized String getName() {
        return name;
    }

    public synchronized void invert() {
        red = 255 - red;
        green = 255 - green;
        blue = 255 - blue;
        name = "Inverse of " + name;
    }
}
```

Use of a mutable class should be very careful in concurrent application

```
SynchronizedRGB color =
    new SynchronizedRGB(0, 0, 0, "Pitch Black");
...
int myColorInt = color.getRGB();      //Statement 1
String myColorName = color.getName(); //Statement 2
```

The SynchronizedRGB object may be seen in an inconsistent state:

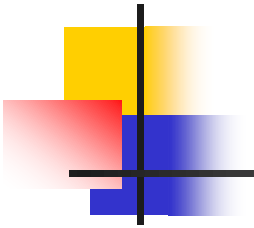
If another thread invokes color.set after Statement 1 but before Statement 2, the value of myColorInt won't match the value of myColorName.

```
synchronized (color) {
    int myColorInt = color.getRGB();
    String myColorName = color.getName();
}
```

To avoid this outcome, the two statements must be bound together.

This kind of inconsistency is only possible for mutable objects — it will not be an issue for the immutable ones.

Strategy for Defining Immutable Objects



1) Don’t provide any *mutators* (“setter” methods)
2) Make all fields *final* and *private*
3) Don't allow subclasses to *override* methods.
4) If the instance fields include *references to mutable objects*, don't allow those objects to be changed:
   - Don't provide methods that modify the mutable objects.
   - Don't share references to the mutable objects.

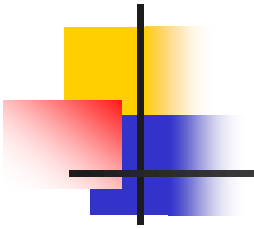
Code example: immutable version

ImmutableRGB.java

```
final public class ImmutableRGB {

    // Values must be between 0 and 255.
    private final int red;
    private final int green;
    private final int blue;
    private final String name;

    public ImmutableRGB(int red,
                        int green,
                        int blue,
                        String name) {
        this.red = red;
        this.green = green;
        this.blue = blue;
        this.name = name;
    }

    public int getRGB() {
        return ((red << 16) | (green << 8) | blue);
    }

    public String getName() {
        return name;
    }

    public ImmutableRGB invert() {
        return new ImmutableRGB(255 - red,
                                255 - green,
                                255 - blue,
                                "Inverse of " + name);
    }
}
```

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

Disadvantage: Performance



- Typical approach:
  - Provide *immutable class*
  - Provide *mutable companion class* for situations where performance is an issue
- Clients choose on performance needs
- Example in Java Library:
  - `String` (Immutable)
  - `StringBuilder` (Companion Mutable Class)
    - `StringBuffer` (Deprecated Companion Mutable Class)
- Static factories can cache frequently used items

More discussions: should all fields of immutable class be marked as final?

Why String class is immutable even though it has a non -final field called "hash"

11

1

I was reading through Item 15 of Effective Java by Joshua Bloch. Inside Item 15 which speaks about 'minimizing mutability' he mentions five rules to make objects immutable. One of them is is to make all fields final . Here is the rule :

> **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].

I know that String class is an example of a immutable class. Going through the source code I see that it actually has a hash instance which is not final .

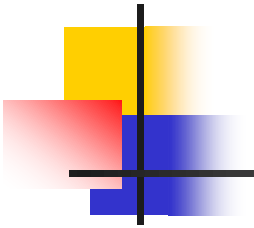
```
//Cache the hash code for the string
private int hash; // Default to 0
```

How does String become immutable then ?

More discussions: should all fields of immutable class be marked as final?

- "No methods may be modify the object and that all its fields must be final" is *a bit stronger than necessary*.
- In truth, no method may produce an "*external visible*" change in the object's state.
  - The *hash* field in String is used to cache the hash code for the string, see detail explanation at stackoverflow

Item 18: Favor Composition over Inheritance



- Issue ONLY for *implementation inheritance*
  - Interface inheritance does NOT have these problems
  - Interface inheritance is better for lots of reasons…
- Inheritance breaks encapsulation!
  - In other words, a subclass *depends on the implementation details* of its superclass for its proper function (see example in next few slides)
  - Difficult to evolve superclass without breaking subclasses

Example: Broken Subtype

```
// This is very common, but broken
public class InstrumentedHashSet<E> extends HashSet<E>
  private int addCount = 0; // add() calls
  public InstrumentedHashSet() {}
  public InstrumentedHashSet(Collection<? extends E> c)
     { super(c); }
  public boolean add(E o) {
     addCount++; return super.add(o);
  }
  public boolean addAll(Collection<? extends E> c) {
     addCount += c.size(); return super.addAll(c);
  }
  // accessor method to get the count
  public int addCount() { return addCount; }
}
```

Broken Example, continued

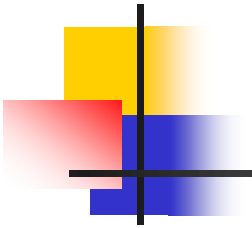
- So, what's the problem?

Demo

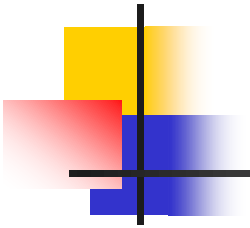
```
InstrumentedHashSet<String> s =
    new InstrumentedHashSet<String>();
s.addAll(Arrays.asList("Snap", "Crackle", "Pop"));
```

- What does `addCount()` return?
  - 3?
  - 6?
- Internally, `HashSet addAll()` is implemented on top of `add()`, which is overridden. Note that this is an implementation detail, so we *can't* get it right in `InstrumentedHashSet`

Source of Difficulty: Overridden Methods



- Overriding methods can be tricky
  - May break the superclass invariant
    - Overridden method does not maintain invariant
  - May break subclass invariant
    - New methods may be added to superclass
    - What if new method matches subclass method?
- Also, recall problems with `equals()` and `hashCode()`



Composition

- Fortunately, composition solves all of these problems – even though it makes the programmer work harder

```
// Inheritance
public Class Foo extends File {…}

// Composition
public class Foo {
   private File f = …;
   // Note: forwarded(转发) methods
...
}
```

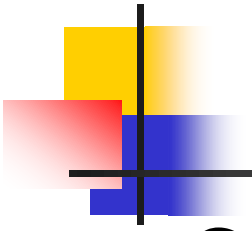
Revisiting the Example

```
public class InstrumentedSet<E> implements Set<E> {
   private final Set<E> s;
   private int addCount = 0;

   public InstrumentedSet (Set<E> s) { this.s = s}
   public boolean add(E o) {
      addCount++; return s.add(o); }
   public boolean addAll (Collection<? extends E> c) {
      addCount += c.size();
      return s.addAll(c);
   }
   // forwarded methods from Set interface
}
```

Revisiting the Example

Note that an InstrumentedSet IS-A Set

```
public class InstrumentedSet<E> implements Set<E> {
    private final Set<E> s;
    private int addCount = 0;

    public InstrumentedSet (Set<E> s) { this.s = s}
    public boolean add(E o) {
        addCount++; return s.add(o); }
    public boolean addAll (Collection<? extends E> c) {
        addCount += c.size();
        return s.addAll(c);
    }
    // forwarded methods from Set interface
}
```



- Consider temporarily instrumenting a Set
- Note that Set is an interface

```
Set<Date> s = new InstrumentedSet<Date>(new TreeSet<Date>(cmp))
Set<E> s2 = new InstrumentedSet<E>(new HashSet<E>(capacity));

static void f(Set<Dog> s) {
    InstrumentedSet<Dog> myS = new InstrumentedSet<Dog> (s);
    // use myS instead of s
    // all changes are reflected in s!
}
```

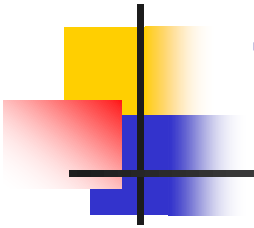
Violations in Java platform libs

- There are a number of obvious violations of this principle in the Java platform libraries.

```
public
class Stack<E> extends Vector<E> {

public
class Properties extends Hashtable<Object,Object> {
```

- In case of *Properties*, the designer intended that only strings be allowed as keys and values
  - *Properties.setProperty only accepts only strings*
  - But *Hashtable.put can accept any object*
  - Invariant of Properties is violated and it was too late to corret ...

Item 19: Design and Document for Inheritance



- Or else prohibit it.
  - Mark the class as *final*
  - Make the constructor *private*
- Document effects of overriding any method
  - Document "*self use*" of overridable methods
  - This is "implementation detail", but unavoidable, since subclass "sees" implementation.
- Inheritance violates encapsulation!

Code example

java.util.AbstractCollection

`public boolean` **`remove`**`(Object o)`

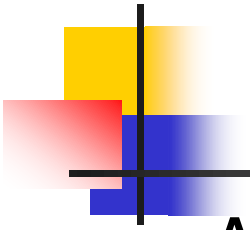
Removes a single instance of the specified element from this collection, if it is present (optional operation). More formally, removes an element `e` such that `(o==null ? e==null : o.equals(e))`, if this collection contains one or more such elements. Returns `true` if this collection contained the specified element (or equivalently, if this collection changed as a result of the call).

This implementation iterates over the collection looking for the specified element. If it finds the element, it removes the element from the collection using the iterator's remove method.

Note that this implementation throws an `UnsupportedOperationException` if the iterator returned by this collection's iterator method does not implement the `remove` method and this collection contains the specified object.

```
public boolean remove(Object o) {
Iterator<E> e = iterator();
if (o==null) {
    while (e.hasNext()) {
    if (e.next()==null) {
        e.remove();
        return true;
    }
    }
} else {
    while (e.hasNext()) {
    if (o.equals(e.next())) {
        e.remove();
        return true;
    }
    }
}
return false;
}
```

The description reveals the implementation detail…



Inheritance is Forever

- A commitment to allow inheritance is *part of public API*
  - If you provide a poor interface, you (and all of the subclasses) are stuck with it.
  - You cannot change the interface in subsequent releases.
- Write subclass to test a class designed for inheritance
  - To find whether a method should be made *protected*
  - Usually three subclasses are sufficient

Constructors Must Not Invoke Overridable Methods

```
// Problem - constructor invokes overridden m()
public class Super {
  public Super() { m();}
  public void m() {…};
}
public class Sub extends Super {
  private final Date date;
  public Sub() {date = new Date();}
  public void m() { // access date variable}
}
```

Demo

What Is the Problem?

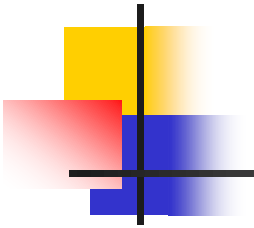
- Consider the code

```
Sub s = new Sub();
```

1) The first thing that happens in `Sub()` constructor is a call to constructor in `Super()`
2) The call to `m()` in `Super()` is overridden
3) But `date` variable is not yet initialized!
4) Further, initialization in `Super m()` never happens!

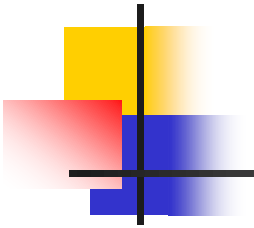
- Yuk!

Inheritance and Cloneable, Serializable



- Since `clone()` and `readObject()` behave a lot like constructors, these methods *cannot invoke overridable methods either*.

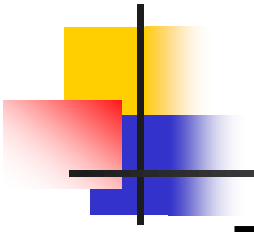
- Problem – access to uninitialized state



Bottom Line

- Be sure you want inheritance, and design for it
- Don't inherit from ordinary concrete class consider:
  - Make class *final*,
  - Make constructors *private* (or package-private)

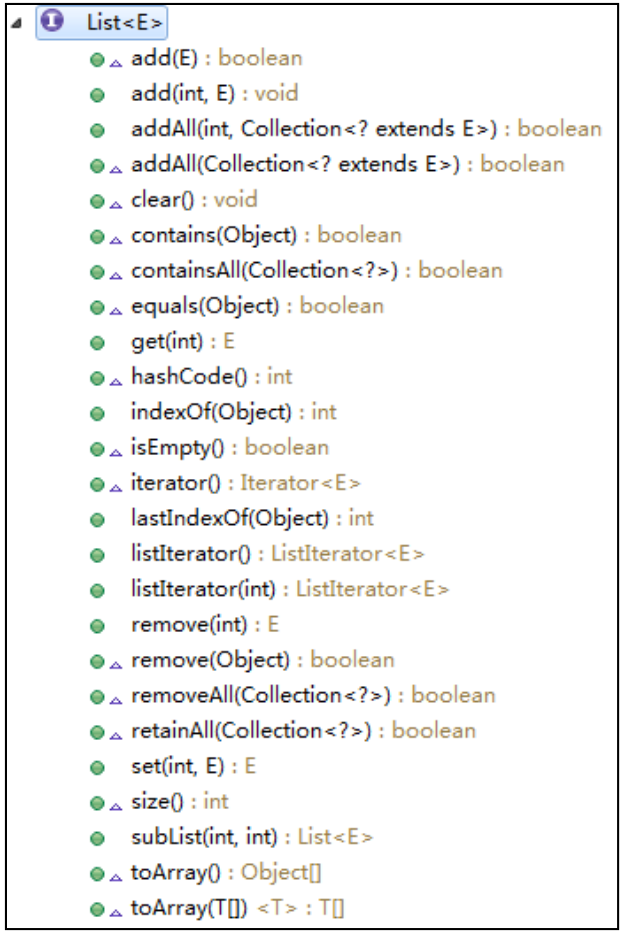
Item 20
Interfaces & Abstract Classes



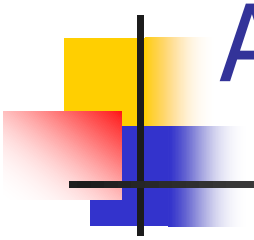
- Existing classes can be easily retrofitted(*改进, 翻新*) to *implement a new interface*
  - Same is not true for abstract classes due to single inheritance model in Java
- Interfaces are ideal for "*mixins*"
  - Example: `Comparable` interface
- Interfaces aren't required to form a *hierarchy*
  - Some things aren't hierarchical

More on interface

- But interface can not contain method implementation
  - Sometimes too tedious to use
  - You have to implement a dozen methods before you can create an object



Advantage of Abstract Classes



- It is far *easier to evolve* an abstract class than an interface
  - It is relatively easy to *add a new method* to an abstract class
  - But almost impossible to add a method to a public interface
    - Should be very carefully to design public interfaces

Skeletal implementation to rescue

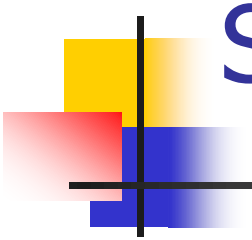
- Combine the *virtues of interfaces & abstract* classes
  - The *interface* still defines the type
  - But the *skeletal implementation* takes all of the work out of implementing it
- By convention, skeletal implementation are named **Abstract*Interface***
  - E.g. `AbstractCollection`, `AbstractSet`, `AbstractList`, and `AbstractMap`

Example for AbstractList

```
// Concrete implementation built on abstract skeleton
static List<Integer> intArrayAsList(final int[] a) {
   if (a == null) throw new NPE(…);
   // Note anonymous class
   return new AbstractList() {
     public Object get(int i) {
        return new Integer(a[i]);
      }
      public int size() { return a.length; }

      public Object set(int i, Object o) {
         int oldVal = a[i];
         a[i] = ((Integer) o).intValue();
         return new Integer(oldVal);
      }
   };
}
```

This Implementation Does a Lot!

- The `List` interface includes many methods.
- Only 3 of them are explicitly provided here.
- This is an "anonymous class" example
  - Certain methods are overridden



Summary

- *Favor interface* over abstract class.
  - Unless *ease of evolution* is more important than flexibility and power.
- Do understand the *limitations of abstract class.*
- *Skeletal implementation* can be considered to go with a nontrivial interface.
- Design *public interface* carefully and do through test before publishing.

Item 23: Prefer Class Hierarchies to Tagged Classes

```
//Tagged Class – vastly inferior to a class hierarchy
class Figure {
   enum Shape { RECTANGLE, CIRCLE };

   final Shape shape;            // Tag field
   double length; double width; // for RECTANGLE
   double radius;               // for CIRCLE

   Figure (double length, double width) {…} // RECTANGLE
   Figure (double radius) {…}   // CIRCLE

   double area() {
       switch (shape) {     // Gag! Roll-your-own dispatching!
           case RECTANGLE: return length*width;
           case CIRCLE:  return Math.PI*(radius * radius);
           default:  throw new AssertionError();
    }
}
```

Add new shape is not easy

More on Item 20

```
//Tagged Class – vastly inferior to a class hierarchy
class Figure {
   enum Shape { RECTANGLE, CIRCLE };

   final Shape shape;              // Tag field
   double length; double width; // for RECTANGLE
   double radius;                  // for CIRCLE

   Figure (double length, double width) {…} // RECTANGLE
   Figure (double radius) {…}    // CIRCLE

   double area() {
      switch (shape) {     // Gag! Roll-your-own dispatching!
         case RECTANGLE: return length*width;
         case CIRCLE:  return Math.PI*(radius * radius);
         default:  throw new AssertionError();
      }
}
```

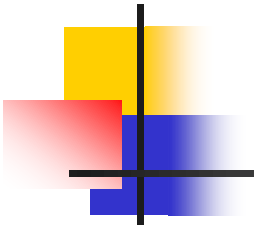
Add new shape is not easy

Fields not applicable to all Figures and can not make final

Multiple implementation

Potential code redundancy

Convert tagged class into class hierarchy

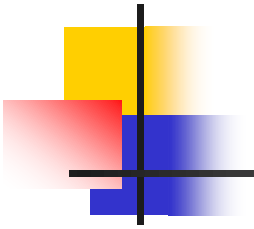


- Define an abstract class
  - *Abstract method* – each method in the tagged class whose behavior depends on the tag value
    - E.g. *area()* in *Figure* class
  - *Final method* – each method whose behavior does not depend on the tag value
  - *Common fields* – shared by all the specific types
- Define a concrete class for each specific type
  - Appropriate implementation for each abstract method.
  - Data fields particular to this type

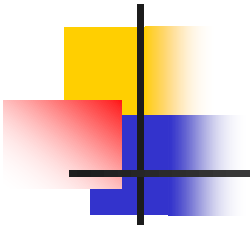
A much better solution

```
//Class hierarchy replacement for a tagged class
abstract class Figure {          // Note: NOT instantiable!
    abstract double area();
}
class Circle extends Figure {
   final double radius;
   Circle(double rad) { radius = rad; }
   double area() {
        return Math.PI * (radius * radius);
   }
}
class Rectangle extends Figure {
   final double length; final double width;
   Rectangle (double len; double wid)
      { length = len; width = wid; }
   double area() {
      return length * width;
    }
}
```

Item 24: Favor Static Member Classes Over Nonstatic



- Nested classes (嵌套类)
  - Defined within another class
  - Used to serve its enclosing class (外围类)
    - Inner class methods can access the private data from the scope in which they are defined.
    - Hidden from other classes in the same package.
    - *Anonymous inner classes* are handy to define callbacks without writing a lot of code



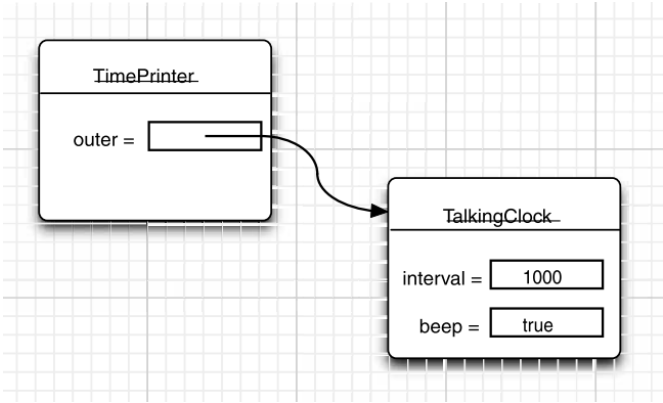
Four flavors of nested classes

- Static member class
- Nonstatic member class (inner)
- Anonymous class (inner)
- Local class (inner)

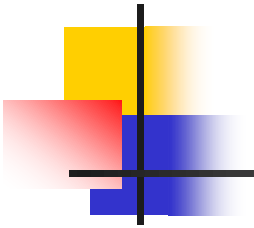
See code examples to understand the differences.

Static vs NonStatic

- Static *requires no connection* to enclosing instance.
- Nonstatic always *associated with enclosing instance*
  - Possible performance issue
  - Possible desire to create by itself



- Hence recommendation to favor static



Local classes

- Declared anywhere a local variable may be declared
- Same scoping rules
- Have names like member classes
- May be static or nonstatic (depending on whether context is static or nonstatic)

More facts about nested class

- Inner classes are a phenomenon of the *compiler*, not the *virtual machine*.
- Local/anonymous class can access the outer *local final variable*.
  - The beep in the class TimePrinter is a copy.

(Refer to CJ1 6.4.5 for detail)

```
public void start(int interval, final boolean beep)
{
   class TimePrinter implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
          Date now = new Date();
          System.out.println("At the tone, the time is " + now);
          if (beep) Toolkit.getDefaultToolkit().beep();
      }
  }

   ActionListener listener = new TimePrinter();
   Timer t = new Timer(interval, listener);
   t.start();
}
```