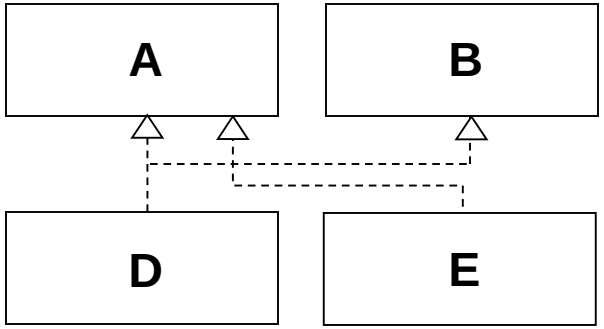
Inheritance Considered Harmful

- **Inheritance and Reentrance**
- **Example: StringOutputStream**
- **Robust Variants**
  - Forwarding
  - Template Methods / Hooks
  - Inner calls

Inheritance

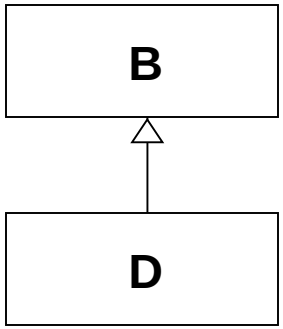
- **Interface Inheritance**
  - Subtyping
  - Reuse of interfaces only
  - Establishment of substitutability
  - Java: multiple subtyping

Extender inherits an obligation



- **Code inheritance**
  - Subclassing
  - Inheritance of code
  - Java: single subclassing

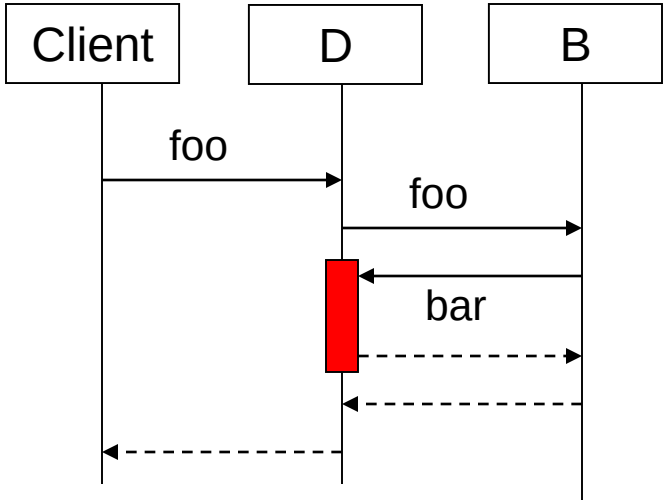
Extender inherits code



Subclassing: Example

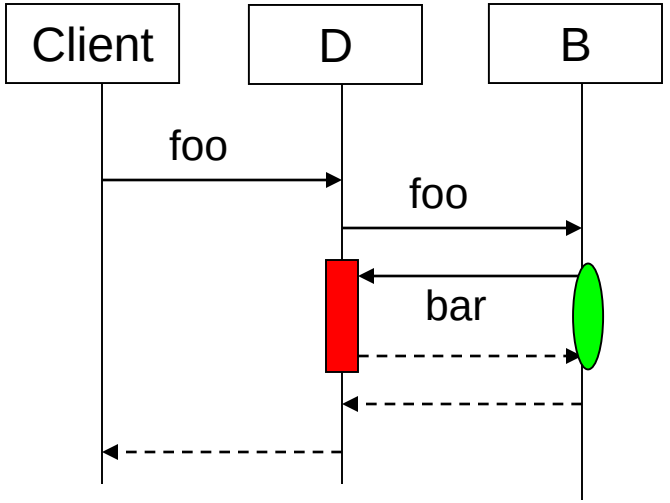
```
class B {
    void foo () { bar(); }
    void bar () { }
}

class D extends B {
    void foo () { super.foo(); }
    void bar () { }
}
```



- Methods defined in class D can call methods defined in class B (e.g. supercall, statically bound)
- Methods defined in class B can call methods implemented in class D (virtual method call, dynamically bound)

Reentrance



- **Observable State**
  - In which state is B when it calls back D.bar?
  - This state can be observed from within method D.bar

- **Method Replacement or Method Extension**
  - Can the overwritten method be replaced or is a supercall mandatory?

Example: StringOutputStream

- **Consider the following class**

```
public class StringOutputStream extends FilterOutputStream {
  public StringOutputStream(OutputStream s) { … }
  void write (char ch) { … }
  void write (String s) { … }
}
```

  - suppose, we want to add a mechanism to this class that allows us to count the characters written (by either of the two write methods)

```
public class CountedOutputStream extends StringOutputStream {
  public CountedOutputStream(OutputStream s) { … }
  public int writtenChars() { … }
  ….
}
```

Inheritance Breaks Information Hiding

- **Specialization Interface**
  - Strong relationship between base and derived class
  - In order to extend a class with subclassing, in-depth knowledge about the implementation of the base class is necessary!
    - which virtual methods are called in base class methods?
    - in which state is the base class upon invocation of virtual methods?
  - Specialization interface =
    - Protected (and public) methods with
    - Specialization semantics

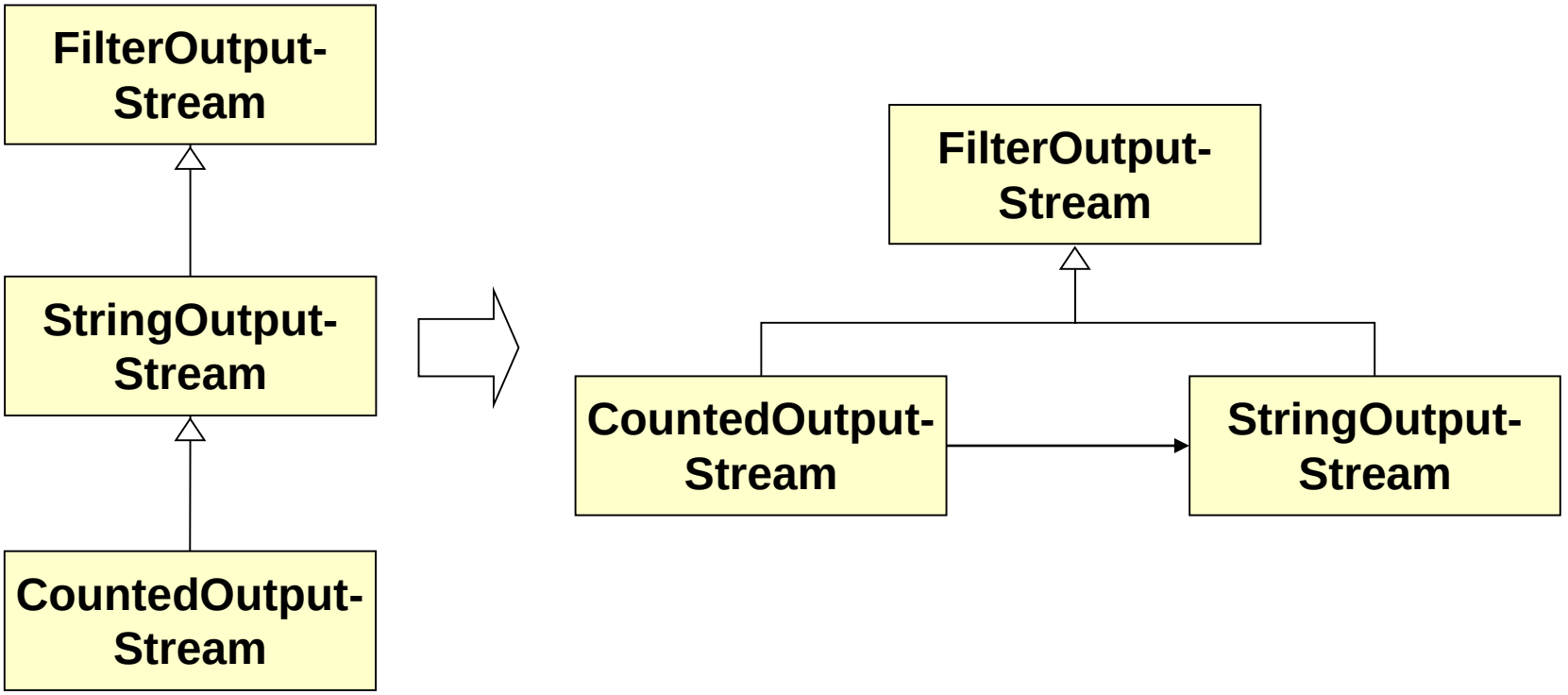
- **Inheritance breaks encapsulation**
  - Support for inheritance implies that (some) implementation details have to be published!

Inheritance Considered Dangerous

- **Inheritance and Callbacks**
  - every method can call every other and become a callback
  - cyclic dependencies introduced by inheritance are difficult to see
- **Fragility**
  - a problem may show only with a specific configuration of components (i.e. it is hard (impossible) to detect through testing)
  - if a base class is modified, its extensions may break
- **Specialization Interface Specification**
  - we need a specification of the specialization interface

Robust Variant: Forwarding

- **Forwarding**



Robust Variant: Forwarding

- **Forwarding**

```
public class CountedOutputStream extends FilterOutputStream {
    private int c = 0;
    private StringOutputStream inner;
    public CountedOutputStream (OutputStream inner) {
      this.inner = new StringOutputStream(inner);
    }

    public  int writtenChars() { return c; }
    void write (char ch) { c=c+1; inner.write(ch);}
    void write (String s) { c=c+s.length(); inner.write(s); }

    // forwarding of all additional methods
}
```

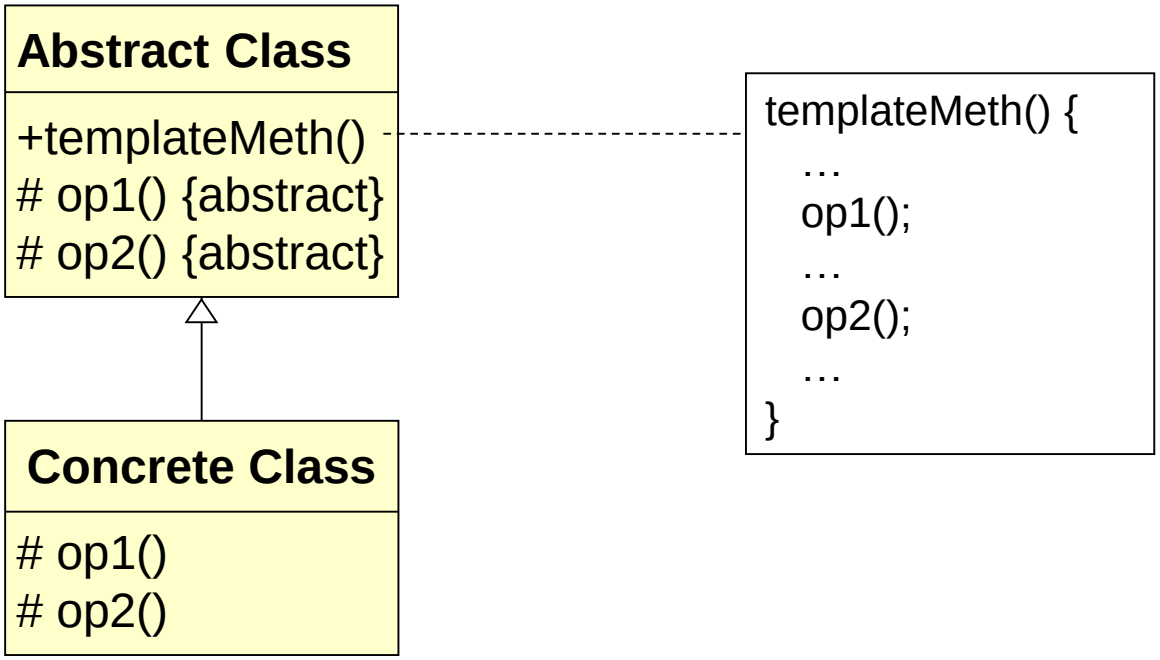
Robust Variant: Forwarding

- **Forwarding**
  - Only the client interface is used
  - No call-backs (in contrast to the inheritance based solutions)
  - No dependency on the implementation

  - Remark:
    - If call-backs are needed then this can be provided
    - => delegation: **explicit** passing of the this reference (or an implementation of a call-back interface)

Robust Variant: Templates/Hooks

- **Template Methods**
  - Template method pattern provides a robust method to allow for extensions
  - Base class provides extension points (op1/op2)



Robust Variant: Templates/Hooks

- **Template Methods**

```
public class StringOutputStream extends FilterOutputStream {
    public StringOutputStream (OutputStream s) { super(s); }
    void write (char ch) { … }
    void write (String s) { … }
    protected char prepareChar(char ch) { return ch; }
}
```

prepareChar is called once for each written character. Might be used to encode the written characters. It is under the responsibility of StringOutputStream to guarantee this property!

```
public class CountedOutputStream extends StringOutputStream {
    private int c = 0;
    public  int writtenChars() { return c; }
    protected char prepareChar(char ch) { c++; return ch; }
}
```

Robust Variant: Templates/Hooks

- **Discussion**
  - State in which extension method is called is defined by the implementation of the base class
  - Implementations of hook methods not necessarily have to invoke a supercall
- **Remarks**
  - Template method may be declared final, extension possible by overwriting the extension hooks

Base Class defines algorithm and provides extension points
=> *PLANNED REUSE*

Robust Variant: Templates/Hooks

- **Example**

```
class GenericServlet implements Servlet, ServletConfig, Serializable {
    public void init(ServletConfig config) throws ServletException {
      this.config = config;
      this.init();
    }
    /* A convenience method which can be overridden so that there's no
     * need to call super.init(config).
     * Instead of overriding init(ServletConfig), simply override this method
     * and it will be called by GenericServlet.init(ServletConfig config).
     * The ServletConfig object can still be retrieved via getServletConfig.
    */
    public void init() throws ServletException {  }
}
```

Robust Variant: Inner Calls

- **Idea**
  - Application of the template method construct to provide robust "supercalls"
  - Problem with supercall
    - Should supercall be invoked at the beginning or end of the implementation?
    - How can the base class assert that supercall is invoked at all?
  - Solution: Inversion of the responsibility:
    - Base class invokes extension ("inner call")
    - Method augmentation (cf. Beta like languages)

Example: Externalizable with super call

```
class Rectangle implements Figure, Externalizable {
   private int x, y, w, h;
   public void move(int dx, int dy) {x += dx; y += dy;}
   public void draw() { /* … */ }
   public void writeExternal(java.io.ObjectOutput s) {
     try { s.writeInt(x); s.writeInt(y); s.writeInt(w); s.writeInt(h); }
     catch(Exception e) { }
   }
}

class TextBox extends Rectangle {
   String text;
   public void writeExternal(java.io.ObjectOutput s) {
      super.writeExternal(s);
      try {  s.writeChars(text); }
      catch(Exception e) { }
   }
}
```

Example: Externalizable with inner call

```
class Rectangle implements Figure, Externalizable {
   private int x, y, w, h;
   public void move(int dx, int dy) {x += dx; y += dy;}
   public void draw() { /* … */ }
   public final void writeExternal(java.io.ObjectOutput s) {
     try { s.writeInt(x); s.writeInt(y); s.writeInt(w); s.writeInt(h); }
     catch(Exception e) { }
     writeExternal2();
   }
   protected void writeExternal2(java.io.ObjectOutput s) { }
}

class TextBox extends Rectangle {
   private String text;
   protected final void writeExternal2(java.io.ObjectOutput s) {
      try{ s.writeChars(text); } catch(Exception e) { }
      writeExternal3();
   }
   protected void writeExternal3(java.io.ObjectOutput s) { }
}
```

CheckStyle Design For Extension

- **CheckStyle**
  - Source Code formatting tool
  - http://checkstyle.sourceforge.net/

- **Design For Extension**
  - Checks that classes are designed for extension. More specifically, it enforces a programming style where superclasses provide empty "hooks" that can be implemented by subclasses.
  - The exact rule is that non-private, non-static methods of classes that can be subclassed must either be
    - abstract
    - final
    - have an empty implementation

CheckStyle Design For Extension

- **Rationale:**
  - This API design style protects superclasses against being broken by subclasses.
  - The downside is that subclasses are limited in their flexibility, in particular they cannot prevent execution of code in the superclass, but that also means that subclasses cannot corrupt the state of the superclass by forgetting to call the super method.

  - http://checkstyle.sourceforge.net/config_design.html#DesignForExtension

Summary

- **Code Inheritance (across package boundaries) is dangerous**
  - Hidden cyclic dependencies
  - Bad evolvability (FBCP)
  - Bad testability
- **However**
  - You can get "yoyo problems" even without using code inheritance, but code inheritance provokes them specifically
- **Advise**
  - Design especially carefully all those interfaces that involve call backs or may involve reentrance into an object
  - Rule of thumb: watch out for cyclic dependencies between classes