

Software Engineering Design & Construction

Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

A Critical View on Inheritance

A Critical View On Inheritance

Inheritance is *the main* built-in variability mechanism of OO languages.

Desired Properties

(of Programming Languages)

- Built-in support for OCP
- Good Modularity
- Support for structural variations
- Variations can be represented in type declarations

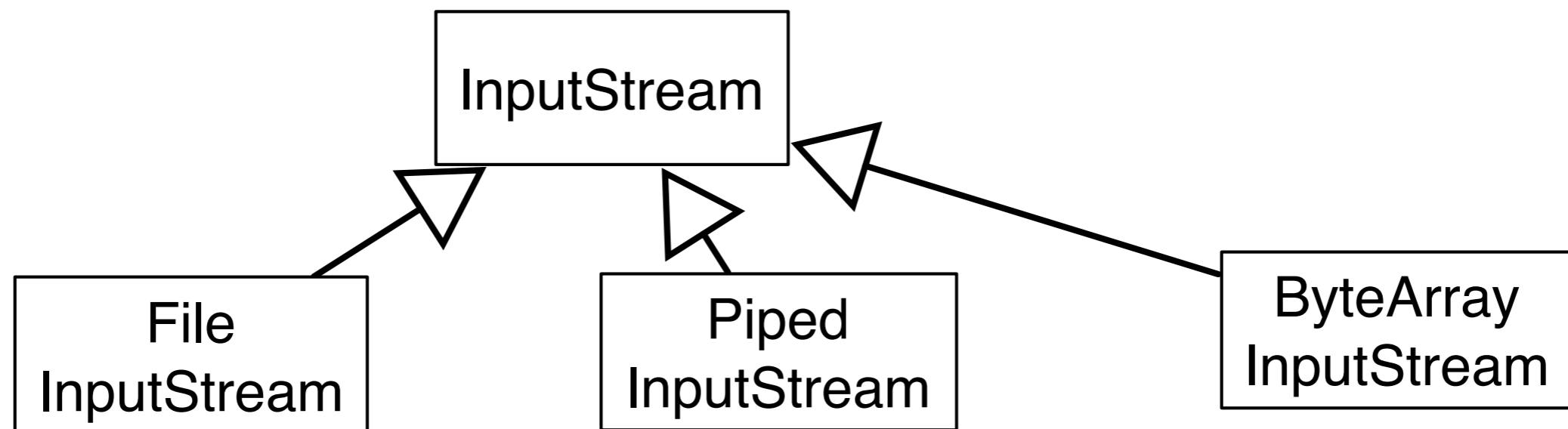
Variation of selection functionality of table widgets.

Desired Properties By Example

```
class TableBase extends Widget {
    TableModel model;
    String getCellText(int row, int col){ return model.getCellText(row, col); }
    void paintCell(int r, int c){ getCellText(row, col) ... }
}
abstract class TableSel extends TableBase {
    abstract boolean isSelected(int row, int col);
    void paintCell(int row, int col) { if (isSelected(row, col)) ... }
}
class TableSingleCellSel extends TableSel {
    int currRow; int currCol;
    void selectCell(int r, int c){ currRow = r; currCol = c; }
    boolean isSelected(int r, int c){ return r == currRow && c == currCol; }
}
class TableSingleRowSel extends TableSel {
    int currRow;
    void selectRow(int row) { currRow = row; }
    boolean isSelected(int r, int c) { return r == currRow; }
}
class TableRowRangeSel extends TableSel { ... }
class TableCellRangeSel extends TableSel { ... }
```

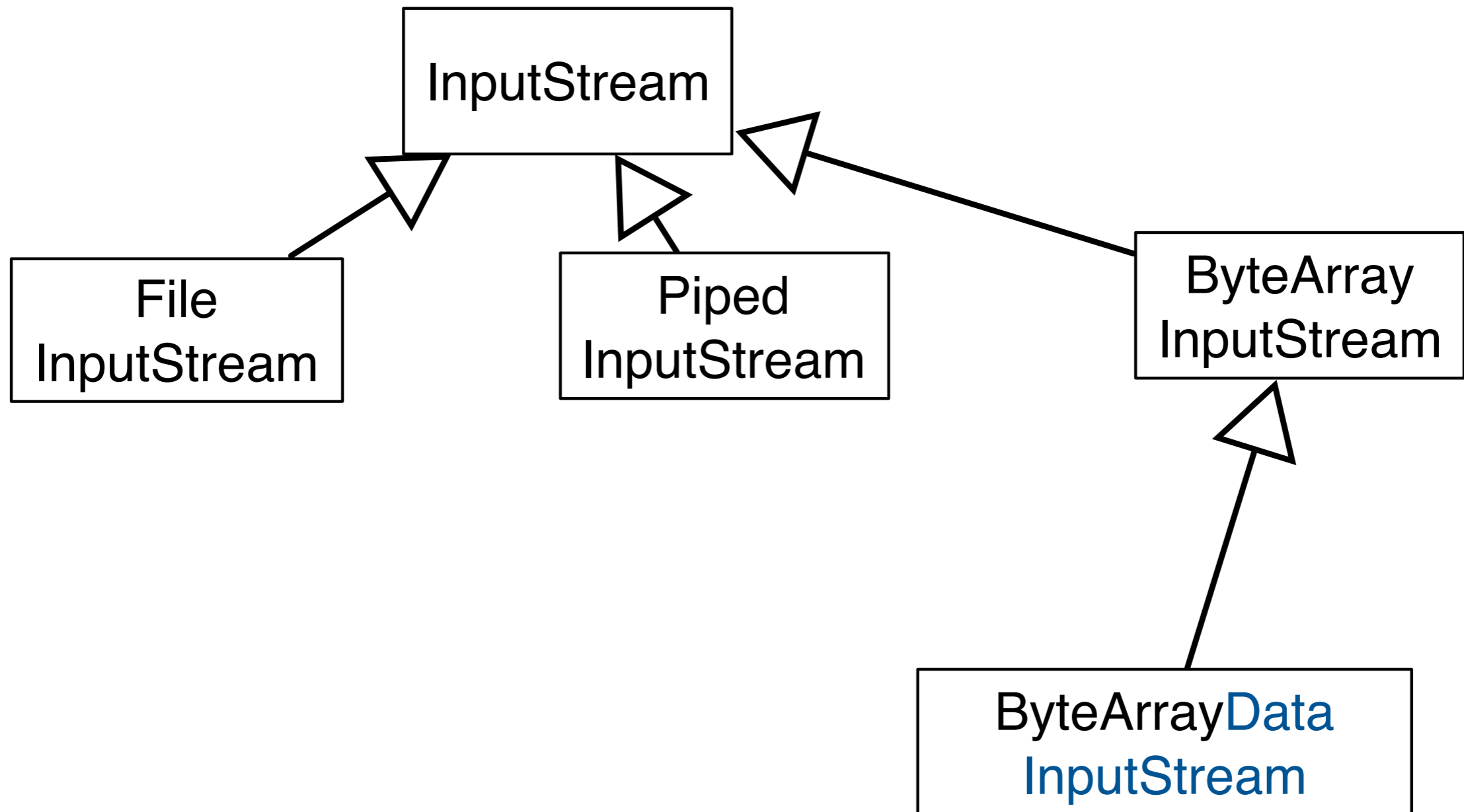
Non-Reusable, Hard-to-Compose Extensions

An Extract from Java's Stream Hierarchy



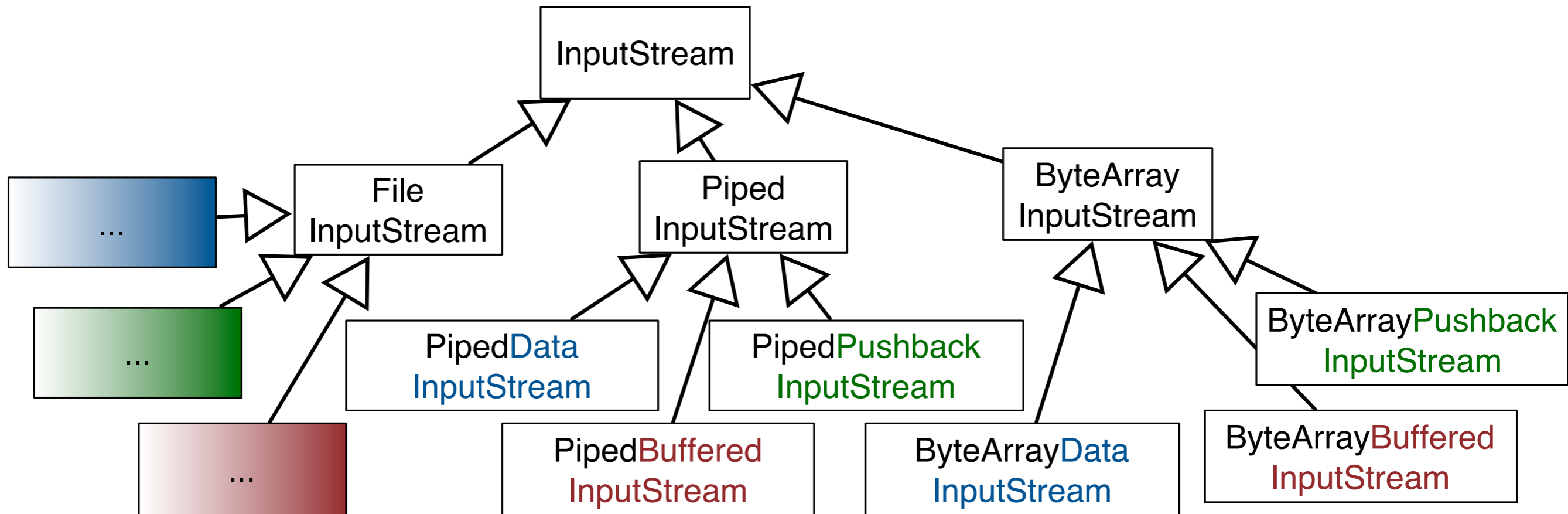
Non-Reusable, Hard-to-Compose Extensions

An Extract from Java's Stream Hierarchy - A Simple Variation



Non-Reusable, Hard-to-Compose Extensions

An Extract from Java's Stream Hierarchy - A Simple Variation



Each kind of variation would have to be re-implemented for all kinds of streams, for all meaningful combinations of variations

Non-Reusable, Hard-to-Compose Extensions

Extensions defined in subclasses of a base class cannot be reused with other base classes.

E.g., the *Pushback* related functionality in `FilePushbackInputStream` cannot be reused.

Weak Support for Dynamic Variability

Variations supported by an object are fixed at object creation time and cannot be (re-)configured dynamically.

A buffered stream is a buffered stream is a buffered stream... It is not easily possible to turn buffering on/off, if buffering is implemented by means of subclassing.

Dynamic Variability Illustrated

The configuration of an object's implementation may depend on values from the runtime context.

- **Potential Solution:**

Mapping from runtime values to classes to be instantiated can be implemented by conditional statements.

```
...if(x) new Y() else new Z() ...
```

- **Issue:**

Such a mapping is error-prone and not extensible. When new variants of the class are introduced, the mapping from configuration variables to classes to instantiate must be changed.

Dynamic Variability Illustrated

The configuration of an object's implementation may depend on values from the runtime context.

- **Potential Solution:**
Using dependency injection.
- **Issue:**
Comprehensibility suffers:
 - Objects are (implicitly) created by the Framework
 - Dependencies are not directly visible/are rather implicit

Dynamic Variability Illustrated

The behavior of an object may vary depending on its state or context of use.

- **Potential Solution:**

Mapping from runtime values to object behavior can be implemented by conditional statements in the implementation of object's methods.

- **Issue:**

Such a mapping is error-prone and not extensible. When new variants of the behavior are introduced, the mapping from dynamic variables to implementations must be changed.

The Fragile Base Class Problem

Cf. Item 17 of Joshua Bloch's, *Effective Java*.

An Instrumented HashSet

The Fragile Base Class Problem Illustrated

```
import java.util.*;
public class InstrumentedHashSet<E> extends HashSet<E> {
    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 static void main(String[] args) {
        InstrumentedHashSet<String> s = new InstrumentedHashSet<String>();
        s.addAll(Arrays.asList("aaa", "bbb", "ccc"));
        System.out.println(s.getAddCount());
    }
}
```

Output?

An Instrumented HashSet

The Fragile Base Class Problem Illustrated

```
import java.util.*;
public class InstrumentedHashSet<E> extends HashSet<E> {
    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 static void main(String[] args) {
        InstrumentedHashSet<String> s = new InstrumentedHashSet<String>();
        s.addAll(Arrays.asList("aaa", "bbb", "ccc"));
        System.out.println(s.getAddCount());
    }
}
```

Does this really(!)
solve the problem?

The Fragile Base Class Problem in a Nutshell

Changes in base classes may lead to unforeseen problems in subclasses.

Inheritance Breaks Encapsulation

Fragility by dependencies on the self-call structure

The Fragile Base Class Problem in a Nutshell

- The fragility considered so far is caused by dependencies on the self-call structure of the base class.
- Subclasses make assumptions about the calling relationship between non-**private** methods of the base class.
- These assumptions are implicitly encoded in the overriding decisions of the subclass.
- If these assumptions are wrong or violated by future changes of the structure of superclass' self-calls, the subclass's behavior is broken.

Fragility by addition of new methods

The Fragile Base Class Problem in a Nutshell

- Fragility by **extending a base class with new methods** that were not there when the class was subclassed.
- Example
 - Consider a base collection class.
 - To ensure some (e.g., security) property, we want to enforce that all elements added to the collection satisfy a certain predicate.
 - We override every method that is relevant for ensuring the security property to consistently check the predicate.
 - Yet, the **security may be defeated unintentionally** if a new method is added to the base class which is relevant for the (e.g., security) property.

Fragility by addition of new methods

The Fragile Base Class Problem in a Nutshell

- Fragility by **extending a base class with a method** that was also added to a subclass.
I.e., we accidentally capture a new method; the new release of the base class accidentally includes a method with the same name and parameter types.
- If the return types differ, **your code will not compile anymore** because of conflicting method signatures.
- If the signatures are compatible, **your methods may get involved in things you never thought about.**

Fragility by addition of new methods

The Fragile Base Class Problem in a Nutshell

- Fragility by **adding an overloaded method to the base class**; the new release of the base class accidentally includes a method which makes it impossible for the compiler (in 3rd party code) to determine the call target of your call.

```
class X { void m(String){...} ; void m(Object o){...}/*added*/ }
```

```
<X>.m(null) // the call target is not unique (anymore)
```

Taming Inheritance

Implementation inheritance
(**extends**) is a powerful way to
achieve code reuse.

But, if used inappropriately, it leads
to fragile software.

Dos and Don'ts

Taming Inheritance

- It is *always* safe to use inheritance within a package.
The subclass and the superclass implementation are under the control of the same programmers.
- It is also OK to **extend classes specifically designed and documented for extension.**
- **Avoid inheriting from concrete classes not designed and documented for inheritance across package boundaries.**

Design and document for inheritance or else prohibit it.

-Joshua Bloch, Effective Java

Classes Must Document Self-Use

Taming Inheritance

- Each public/protected method/constructor must **indicate self-use**:
 - Which overridable methods it invokes.
 - In what sequence.
 - How the results of each invocation affect subsequent processing.
- A class must document any circumstances under which it might invoke an overridable method.

(Invocations might come from background threads or initializers; indirect invocations can also come from static initializers.)

Packages are considered closed!

Common Conventions for Documenting Self-Use

Taming Inheritance

- The description of self-inocations to overridable methods is given at the end of a method's documentation comment.
- The description starts with “**This implementation ...**”. Indicates that the description tells something about the internal working of the method.

Example of Documentation On Self-Invocation

- Taken from: `java.util.AbstractCollection`

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

Removes a single instance of the specified element from this collection.

...

This implementation 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.

Documenting Self-Use In API Documentation

Do implementation details have a rightful place in a good API documentation?

White-Box Use

Black-Box Use

Example of Documentation On Self-Invocation

- Taken from: `java.util.AbstractList`

```
protected void removeRange(int fromIndex, int toIndex)
```

Removes from a list ...

This method is called by the clear operation on this list and its sub lists. Overriding this method to take advantage of the internals of the list implementation can substantially improve the performance of the clear operation on this list and its sub lists...

This implementation gets a list iterator positioned before fromIndex and repeatedly calls `ListIterator.next` and `ListIterator.remove`. Note: If `ListIterator.remove` requires linear time, this implementation requires quadratic time.

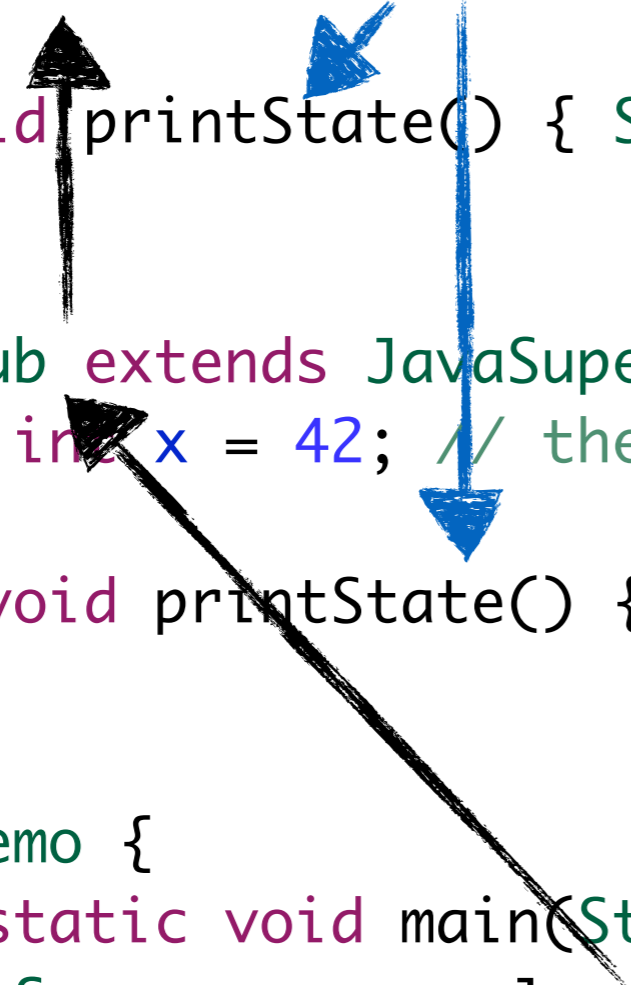
Carefully Design and Test Hooks To Internals

- Provide as few protected methods and fields as possible.
- Each of them represents a commitment to an implementation detail.
- Designing a class for inheritance places limitations on the class.
- Do not provide too few hooks.
- A missing protected method can render a class practically unusable for inheritance.

Constructors Must Not Invoke Overridable Methods

Constructors Must Not Invoke Overridable Methods

```
class JavaSuper {  
    public JavaSuper() { printState(); }  
  
    public void printState() { System.out.println("no state"); }  
}  
  
class JavaSub extends JavaSuper {  
    private int x = 42; // the result of a tough computation  
  
    public void printState() { System.out.println("x = " + x); }  
}  
  
class JavaDemo {  
    public static void main(String[] args) {  
        JavaSuper s = new JavaSub();  
        s.printState();  
    }  
}
```



Result:

x = 0

x = 42

Constructors Must Not Invoke Overridable Methods

```
class ScalaSuper {  
    printState(); // executed at the end of the initialization  
  
    def printState() : Unit = { println("no state") }  
}  
  
class ScalaSub extends ScalaSuper {  
    var y: Int = 42 // What was the question?  
    override def printState() : Unit = { println("y = "+y) }  
}  
  
object ScalaDemo extends App {  
    val s = new ScalaSub  
    s.printState() // after initialization  
}
```

Result:
y = 0
y = 42

Constructors Must Not Invoke Overridable Methods

```
class Super {  
    // executed at the end of the initialization  
    printState();  
  
    def printState() : Unit = { println("no state") }  
}  
  
class Sub(var y: Int = 42) extends Super {  
    override def printState() : Unit = { println("y = "+y) }  
}  
  
object Demo extends App {  
    val s = new Sub  
    s.printState() // after initialization  
}
```

Result:

y = 42

y = 42

Initializers Must Not Invoke Overridable Methods

```
trait Super {  
  val s: String  
  def printState() : Unit = { println(s) }  
  
  printState();  
}
```

```
class Sub1 extends Super { val s: String = 110.toString }  
class Sub2 extends { val s: String = 110.toString } with Super
```

```
new Sub1()  
new Sub2()
```

Result:
null
110