

Principles of Software Construction: Objects, Design, and Concurrency

(Part 1: Designing Classes)

Design for Reuse (class level)

Jonathan Aldrich **Charlie Garrod**

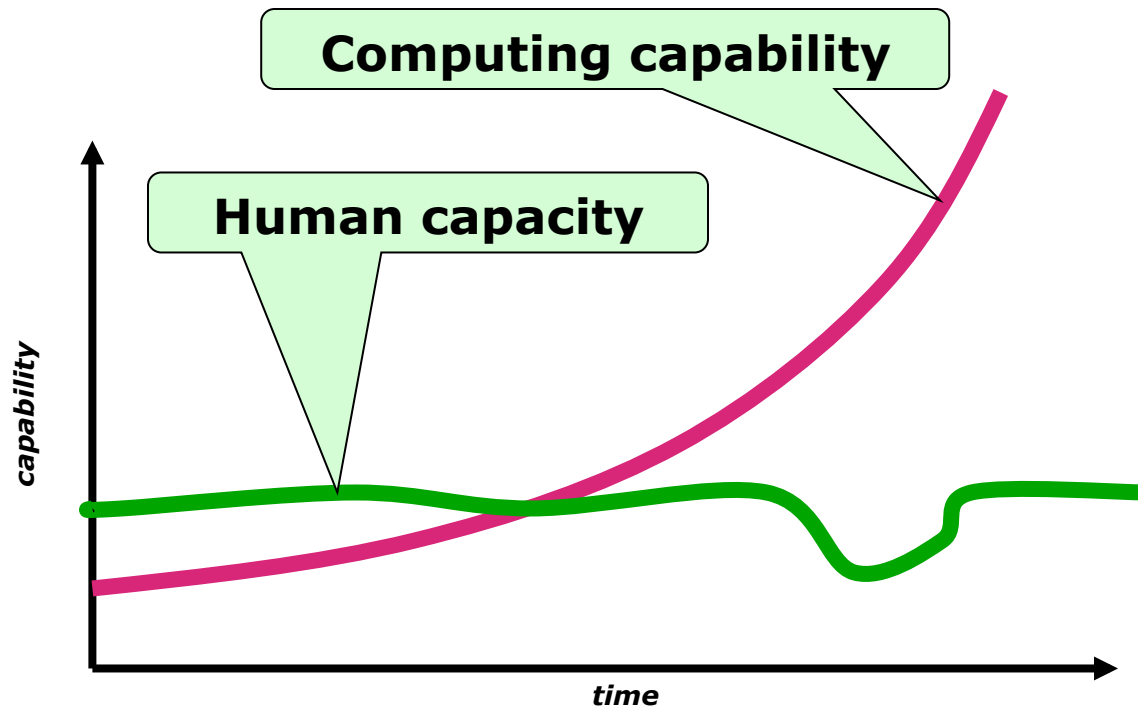
Learning goals for today

- Explain the need for and challenges of design for reuse
- Apply inheritance and delegation appropriately and understand their tradeoffs
- Behavioral subtyping and implications for specification and testing
 - The Object contracts in Java
- Identify applicability of and apply the decorator and template method design patterns
- Read and write UML interaction diagrams

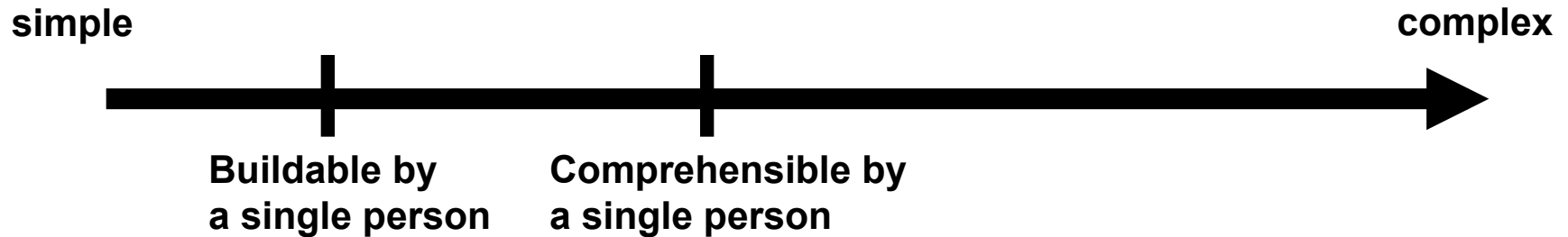
Design goals, principles, and patterns for reuse

- Design goals
 - Design for reuse
 - Design for division of labor
- Design principles
 - Small interfaces
 - Information hiding
- Design patterns
 - Decorator design pattern
 - Template method design pattern
- Language features that support reuse
 - Inheritance
 - Subtype polymorphism (dynamic dispatch)
 - Parametric polymorphism (generics)

The limits of exponentials



Building complex systems



- Division of labor
- Division of knowledge and design effort
- Reuse of existing implementations

Reuse and variation

15-214

Homework #0: A Friendship Graph

Due Tuesday, January 21st at 11:59 p.m.

The goals of this assignment are to familiarize you with our code repository, let you practice Java object-oriented programming. To complete this assignment, you will implement a simple graph class that could represent a network of friends.

Your learning goals for this assignment are to:

- Use Git and GitHub to revision and share work
- Get familiar with Java development environments
- Write a first Java program
- Practice Java style and coding conventions

Instructions

To begin your work, import the project into Eclipse from the homework repository.

Implement and test a `FriendGraph` class that represents friendships. The class should be able to compute the distance between two people in the graph. You can represent the network as an undirected graph where each person is connected to their friends.

For example, suppose you have the following social network:



Homework 0

15-214

Homework 1

Homework #1: Graph Algorithms for Social Networks

Due Tuesday, January 28th at 11:59 p.m.

In this assignment, you will implement a `Graph` interface using two different graph representations. You will then develop several algorithms that use the `Graph` interface that might be used in a social network.

Your goals for this assignment are to:

- Understand and apply the concepts of polymorphism and encapsulation
- Understand interfaces
- Familiarize yourself with Java and Eclipse
- Learn to compile, run, and debug Java programs
- Practice good Java coding practices and style

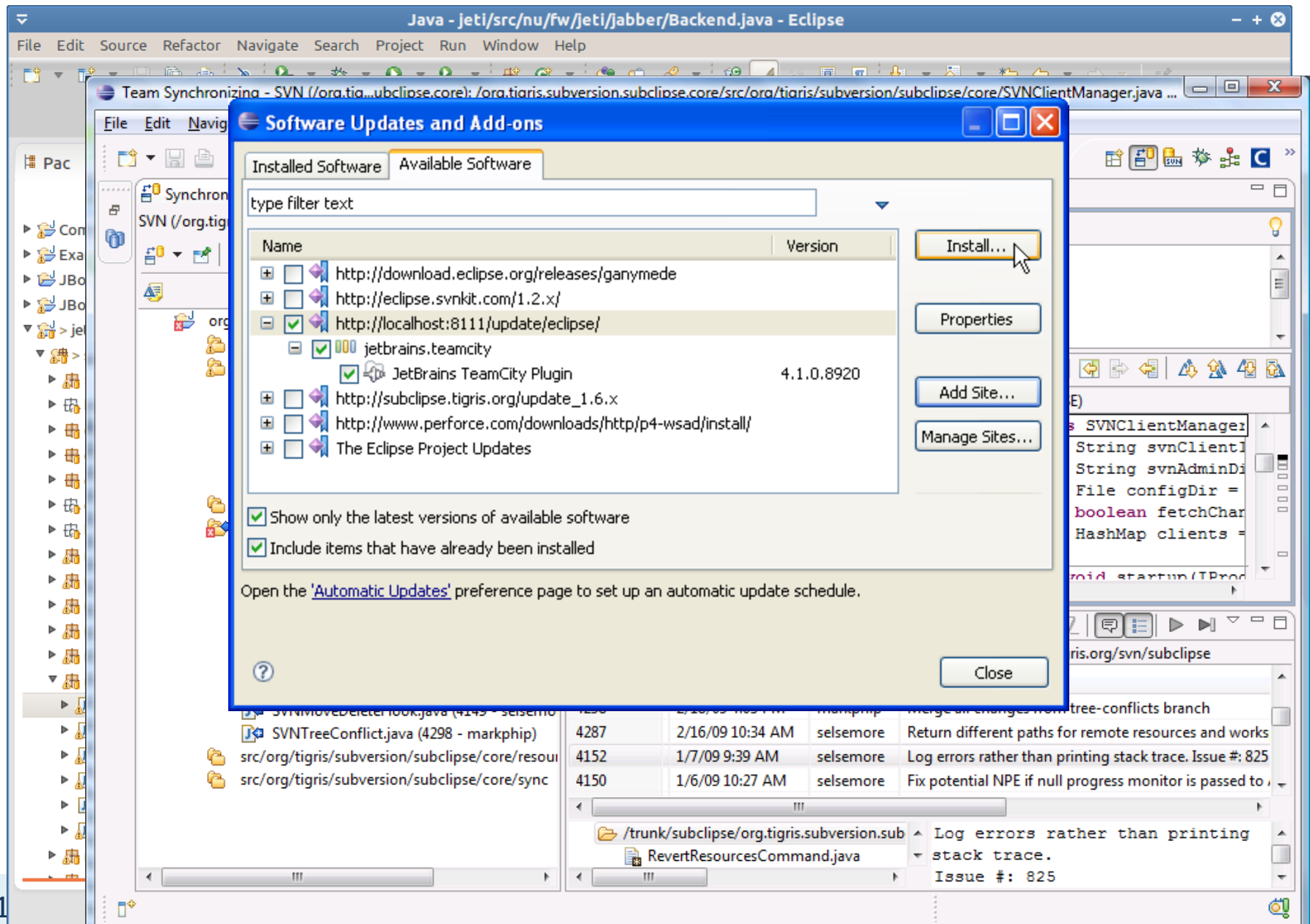
Instructions

Graph Implementations

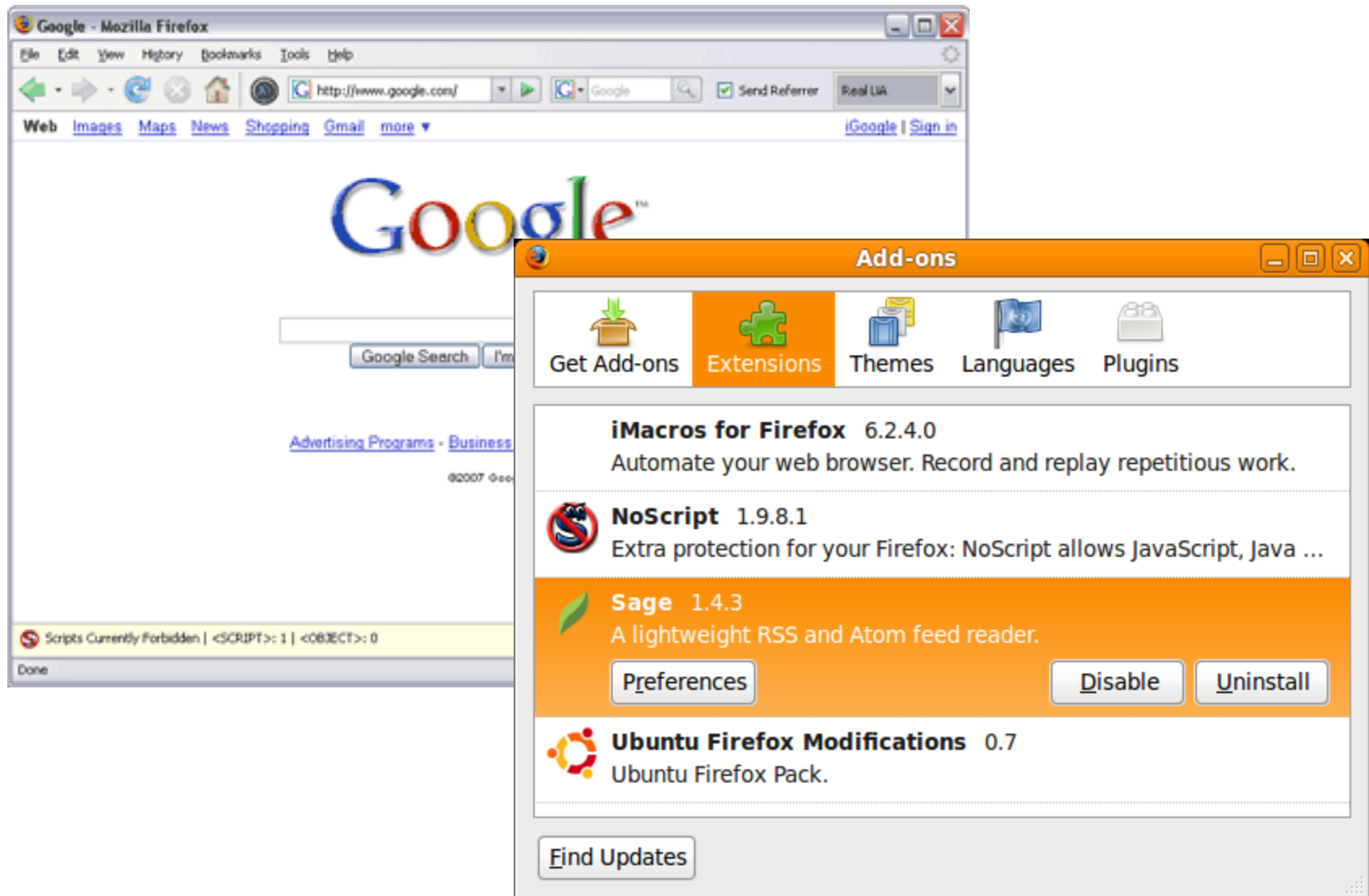
First, write two classes that implement the `edu.cmu.cs.cs214.hw1.staff.Graph` interface, which represents an undirected graph.

- **Adjacency List:** Inside the package `edu.cmu.cs.cs214.hw1.graph`, implement the `AdjacencyListGraph` class. Your implementation must internally represent the graph as an adjacency list. Your class should provide a constructor with a single `int` argument, `maxVertices`. Your implementation must then support a graph containing as many as `maxVertices` vertices. Your implementation may behave arbitrarily

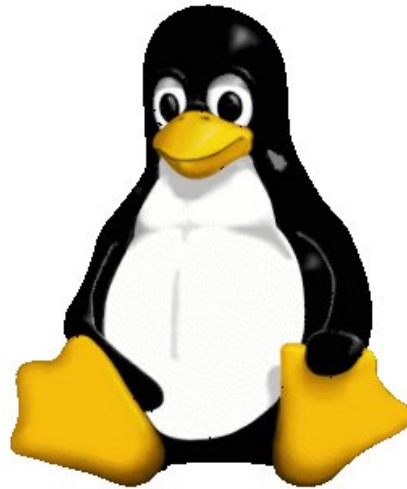
Reuse: Family of development tools



Reuse: Web browser extensions



Reuse and variation: Flavors of Linux



Reuse and variation: Flavors of Linux

Linux Kernel v2.6.18-53.1.14.el5.customxen Configuration

Network File Systems

Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [] excluded <M> module < > module

~(-)

[] Provide NFS client caching support (EXPERIMENTAL)

[*] Allow direct I/O on NFS files (EXPERIMENTAL)

<M> NFS server support

[*] Provide NFSv3 server support

[*] Provide server support for the NFSv3 ACL protocol extension

[*] Provide NFSv4 server support (EXPERIMENTAL)

--- Provide NFS server over TCP support

[*] Root file system on NFS

--- Secure RPC: Kerberos V mechanism (EXPERIMENTAL)

v(+)

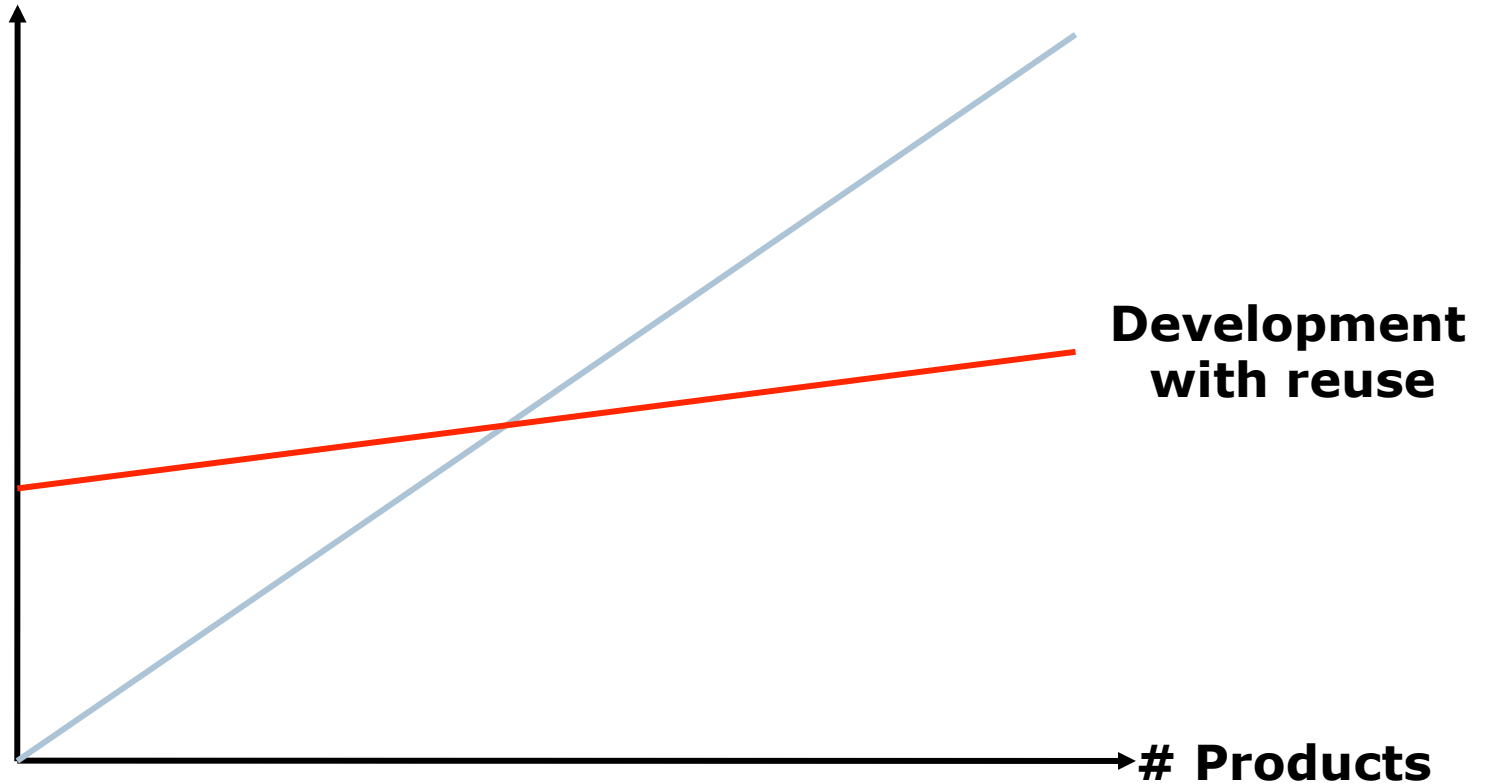
<Select>

< Exit >

< Help >

The promise:

Costs



Agenda

- This week:
 - Class-level reuse with inheritance and delegation
- Later in the course:
 - System level reuse with libraries and frameworks
 - Systematic reuse with software product lines

COMPOSITION AND DELEGATION

Recall our earlier sorting example:

```
public class Sorter {  
    void sort(int[] list, Comparator cmp) {  
        ...  
        boolean mustswap;  
        mustswap = cmp.compare(list[i], list[j]);  
        ...  
    }  
}  
  
interface Comparator {  
    boolean compare(int i, int j);  
}  
  
class UpComparator implements Comparator {  
    boolean compare(int i, int j) { return i<j; }  
}  
  
class DownComparator implements Comparator {  
    boolean compare(int i, int j) { return i>j; }  
}
```

Delegation

- *Delegation* is simply when one object relies on another object for some subset of its functionality
 - In the previous example, the `Sorter` is delegating functionality to some `Comparator` implementation

Delegation

- *Delegation* is simply when one object relies on another object for some subset of its functionality
 - In the previous example, the `Sorter` is delegating functionality to some `Comparator` implementation
- Judicious delegation enables code reuse

Delegation

- *Delegation* is simply when one object relies on another object for some subset of its functionality
 - In the previous example, the `Sorter` is delegating functionality to some `Comparator` implementation
- Judicious delegation enables code reuse
 - `Sorter` can be reused with arbitrary sort orders
 - `Comparators` can be reused with arbitrary client code that needs to compare integers

Using delegation to extend functionality

- Consider the `java.util.List` (excerpted):

```
public interface List<E> {  
    public boolean add(E e);  
    public E      remove(int index);  
    public void   clear();  
    ...  
}
```

- Suppose we want to create a list that logs its operations to the console...

Using delegation to extend functionality

- One solution:

```
public class LoggingList<E> {  
    private List<E> list;  
    public LoggingList<E>(List<E> list) { this.list = list; }  
    public boolean add(E e) {  
        System.out.println("Adding " + e);  
        return this.list.add(e);  
    }  
    public E remove(int index) {  
        System.out.println("Removing at " + index);  
        return this.list.remove(index);  
    }  
    ...  
}
```

The `LoggingList` is composed of a `List`, and delegates (the non-logging) functionality to that `List`

Aside: A sequence diagram for the `LoggingList`

Delegation and design

- Small interfaces
- Classes to encapsulate algorithms
 - E.g., the Comparator, the Strategy pattern

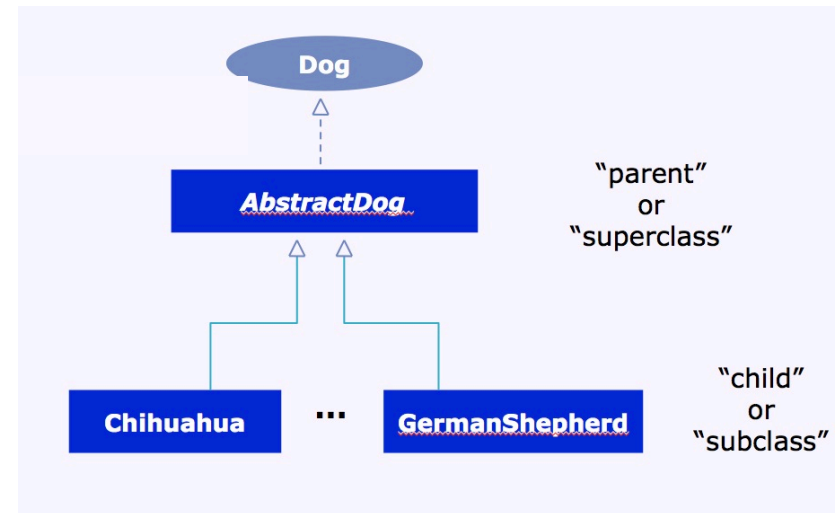
Delegation and design

- Small interfaces with clear contracts
- Classes to encapsulate algorithms, behaviors
 - E.g., the Comparator, the Strategy pattern, the RoutePlanner, the RoutePlannerBuilder, ...

CLASS INHERITANCE AND ABSTRACT CLASSES

An introduction to inheritance

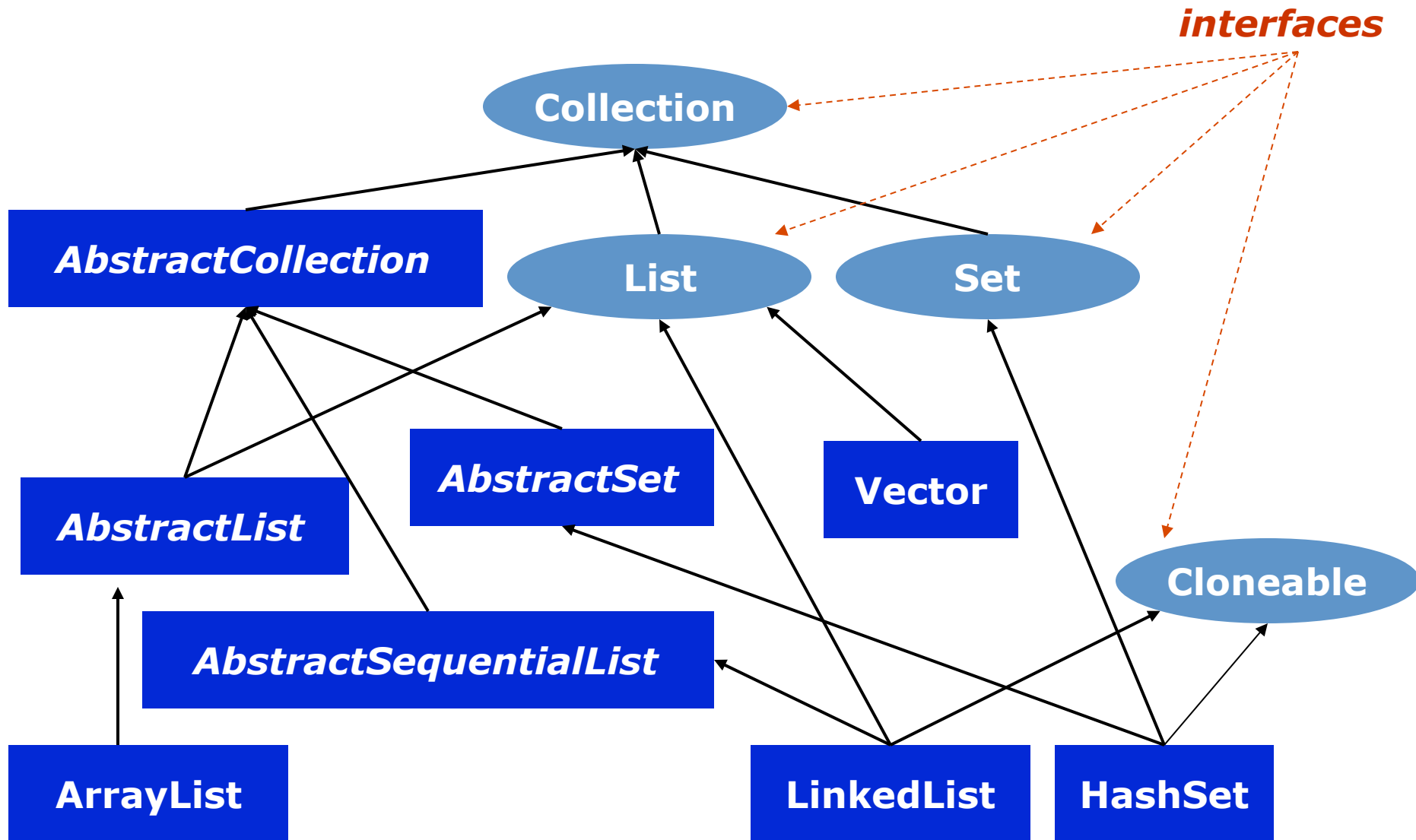
- A dog of an example:
 - Dog.java
 - AbstractDog.java
 - Chihuahua.java
 - GermanShepherd.java
- Typical roles:
 - An interface defines expectations / commitment for clients
 - An abstract class is a convenient hybrid between an interface and a full implementation
 - A subclass overrides a method definition to specialize its implementation



Inheritance: a glimpse at the hierarchy

- Examples from Java
 - `java.lang.Object`
 - Collections library

Java Collections API (excerpt)



Benefits of inheritance

- Reuse of code
- Modeling flexibility
- A Java aside:
 - Each class can directly extend only one parent class
 - A class can implement multiple interfaces

Another example: different kinds of bank accounts

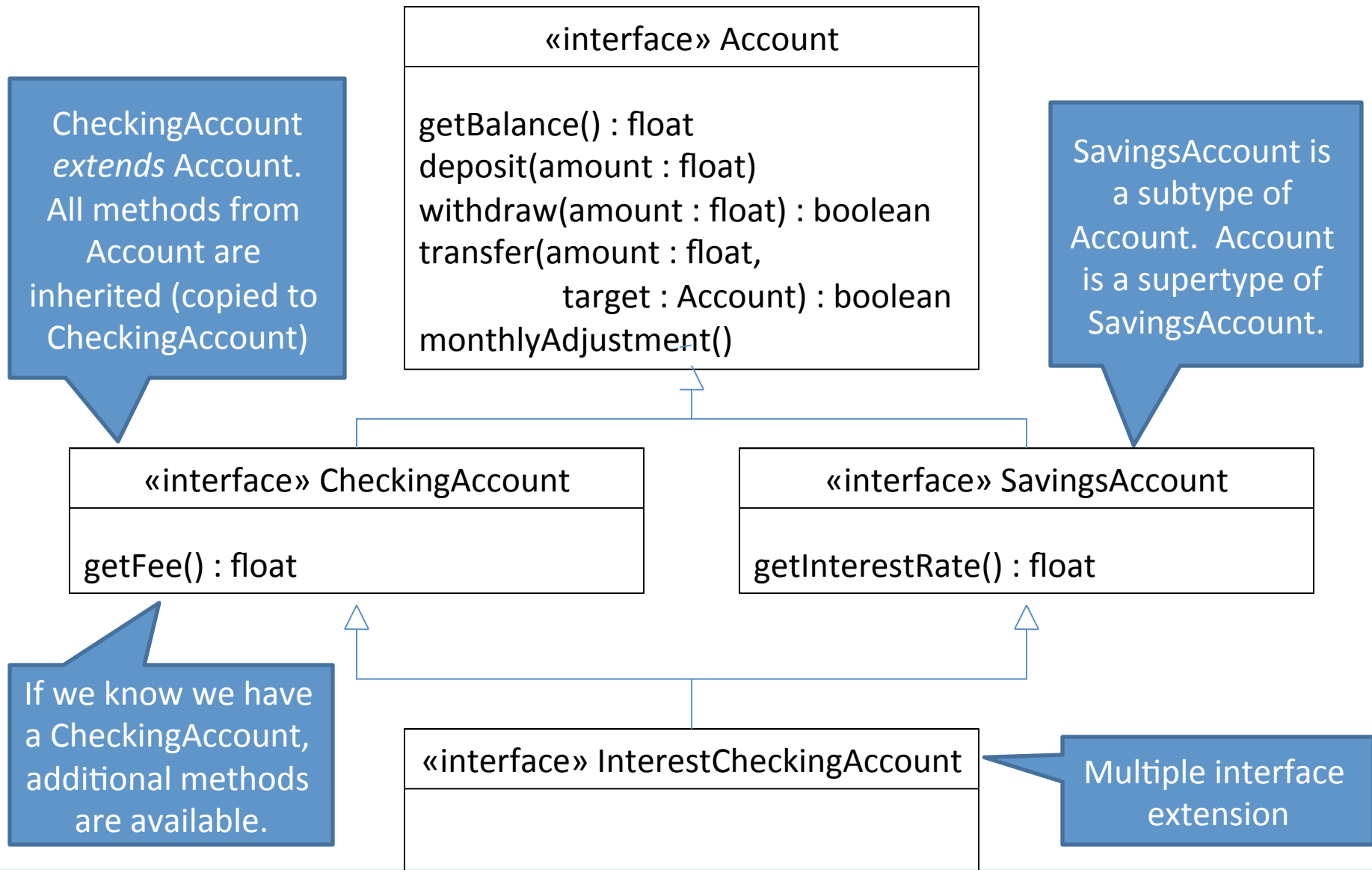
«interface» CheckingAccount

```
getBalance() : float
deposit(amount : float)
withdraw(amount : float) : boolean
transfer(amount : float,
         target : Account) : boolean
getFee() : float
```

«interface» SavingsAccount

```
getBalance() : float
deposit(amount : float)
withdraw(amount : float) : boolean
transfer(amount : float,
         target : Account) : boolean
getInterestRate() : float
```

A better design: An account type hierarchy



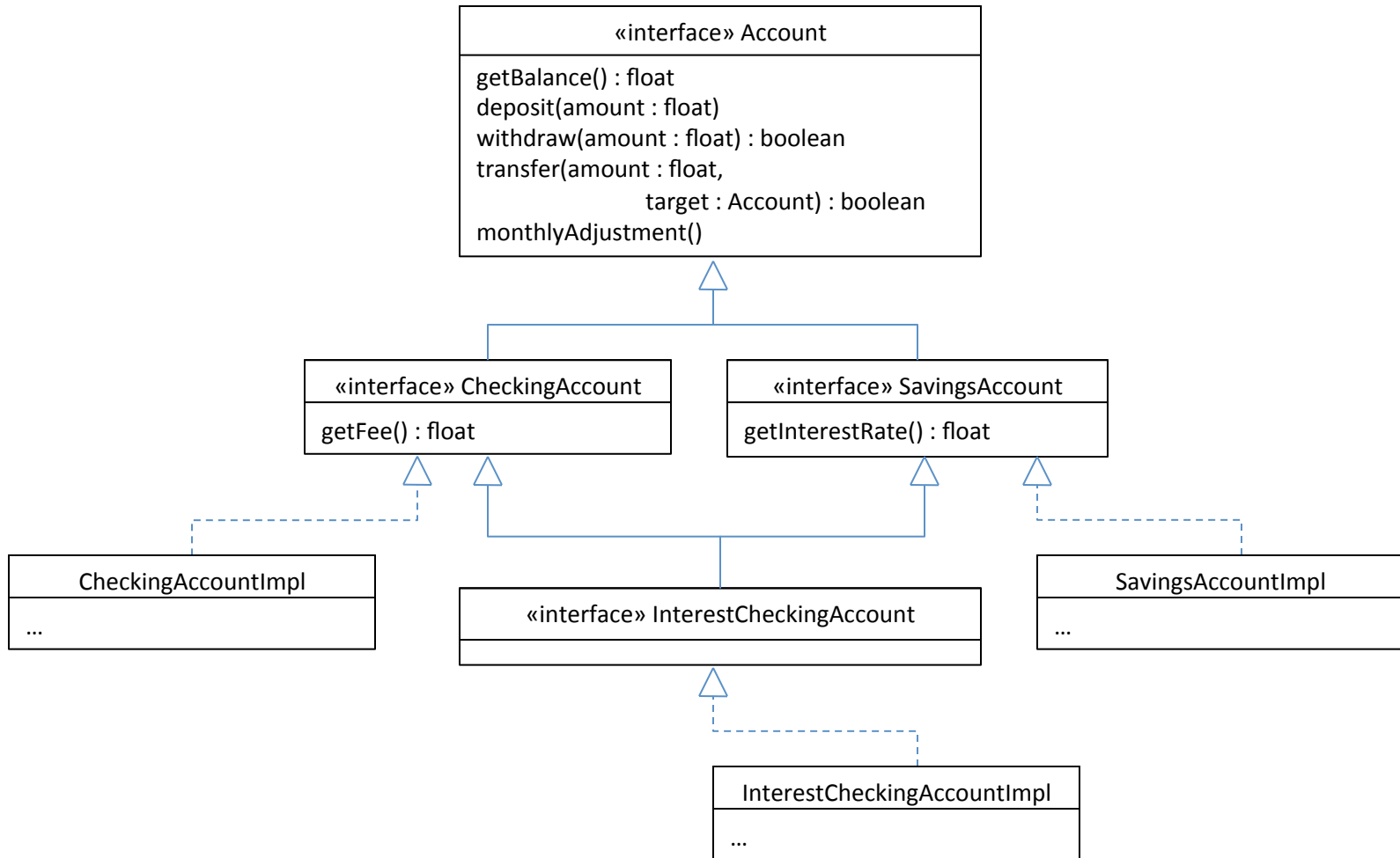
The power of object-oriented interfaces

- Subtype polymorphism
 - Different kinds of objects can be treated uniformly by client code
 - e.g., a list of all accounts
 - Each object behaves according to its type
 - If you add new kind of account, client code does not change
 - Consider this pseudocode:

– See

```
If today is the last day of the month:  
  For each acct in allAccounts:  
    acct.monthlyAdjustment( );
```

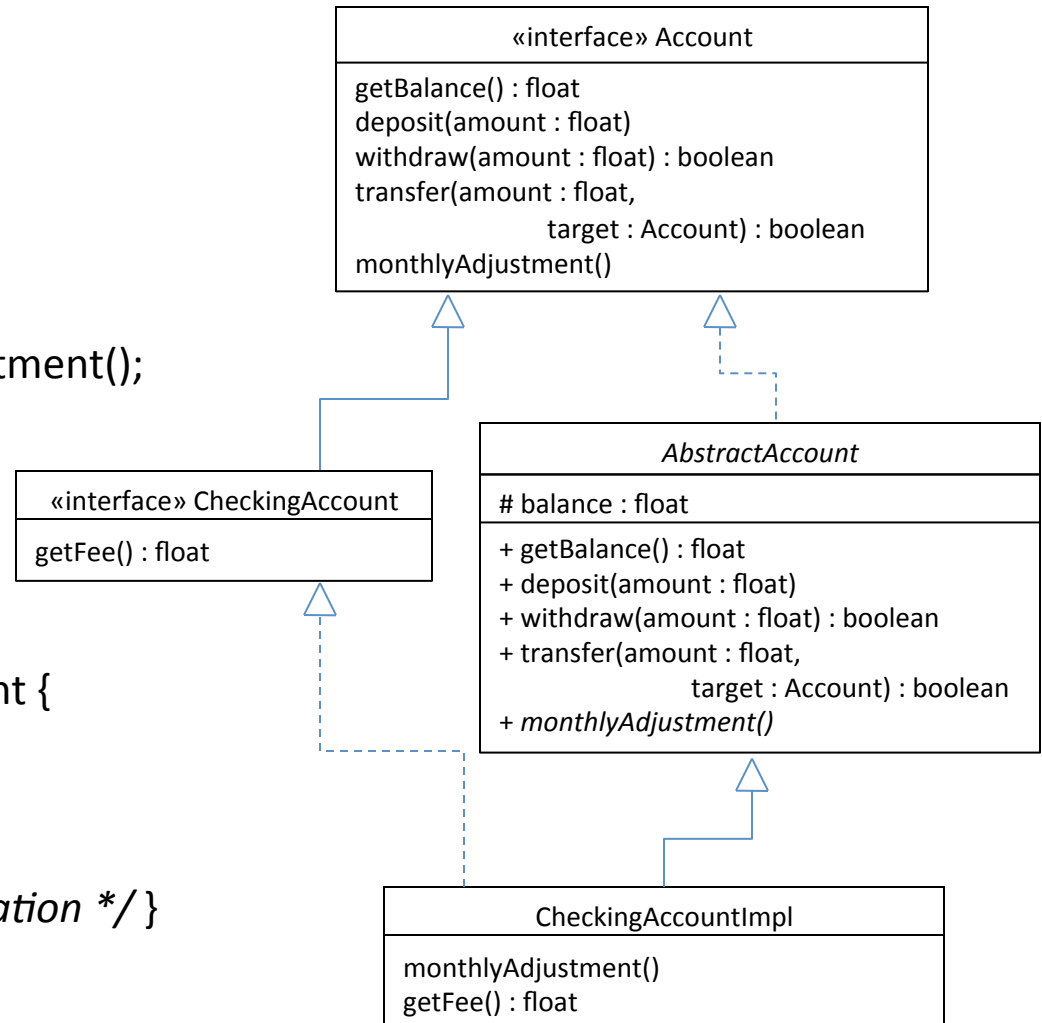
One implementation:



Better: Reuse abstract account code

```
public abstract class AbstractAccount
    implements Account {
    protected float balance = 0.0;
    public float getBalance() {
        return balance;
    }
    abstract public void monthlyAdjustment();
    // other methods...
}

public class CheckingAccountImpl
    extends AbstractAccount
    implements CheckingAccount {
    public void monthlyAdjustment() {
        balance -= getFee();
    }
    public float getFee() { /* fee calculation */ }
}
```



Better: Reuse abstract account code

```
public abstract class AbstractAccount
    implements Account {
    protected float balance = 0.0;
    public float getBalance() {
        return balance;
    }
    abstract public void monthlyAdjustment();
    // other methods
}
```

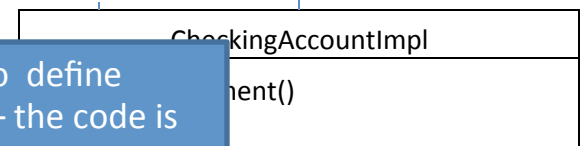
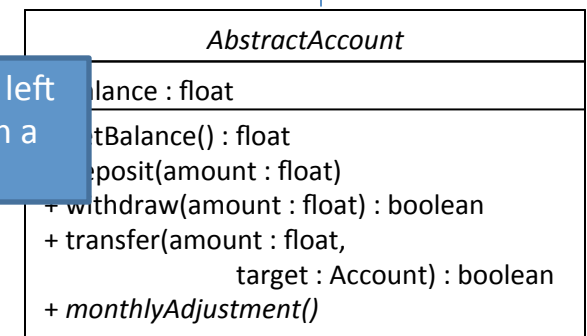
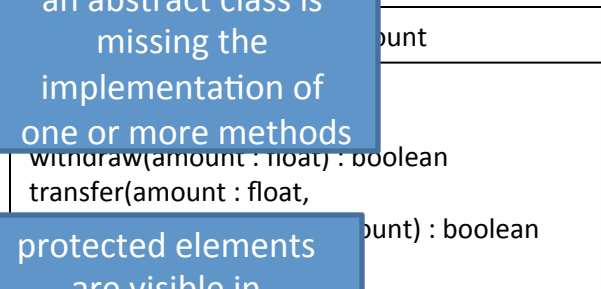
```
public class CheckingAccountImpl
    extends AbstractAccount
    implements CheckingAccount {
    public void monthlyAdjustment() {
        balance -= getFee();
    }
    public float getFee() { /* fee calculation */ }
}
```

an abstract class is missing the implementation of one or more methods

protected elements are visible in subclasses

an abstract method is left to be implemented in a subclass

no need to define getBalance() – the code is inherited from AbstractAccount



Inheritance and subtyping

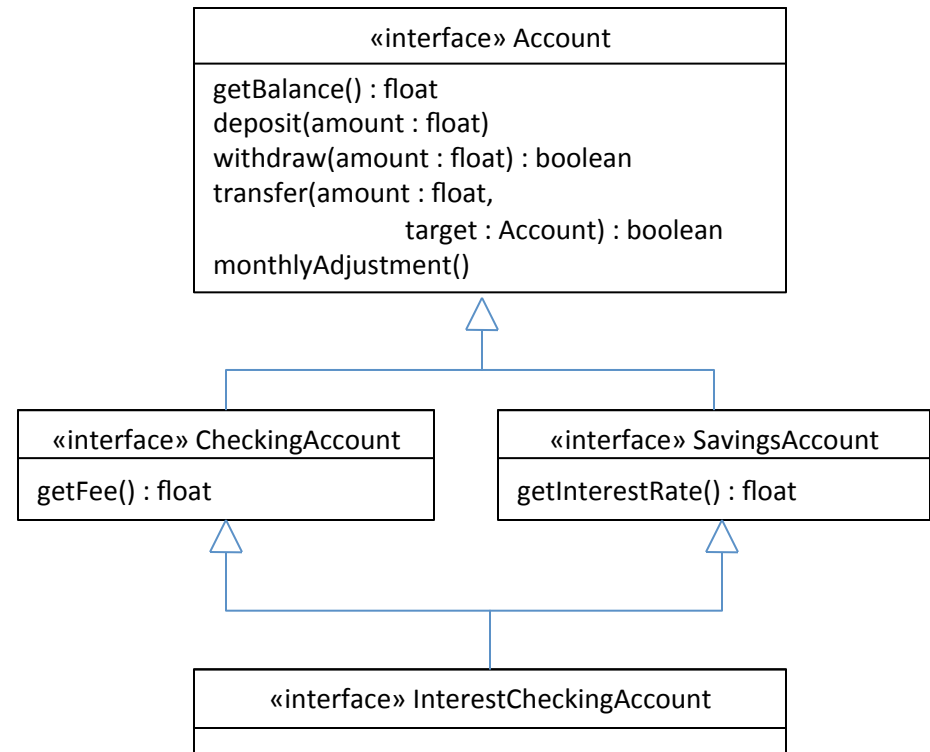
- Inheritance is for code reuse
 - Write code once and only once
 - Superclass features implicitly available in subclass
- Subtyping is for polymorphism
 - Accessing objects the same way, but getting different behavior
 - Subtype is substitutable for supertype

```
class A extends B
```

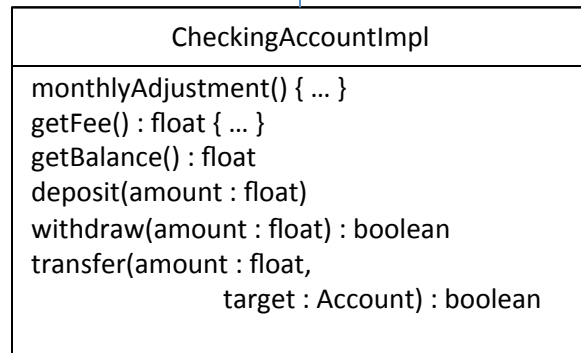
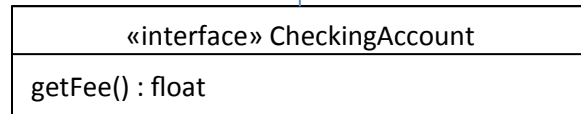
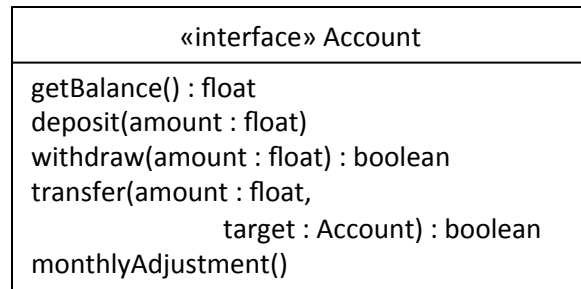
```
class A implements I  
class A extends B
```

Challenge: Is inheritance necessary?

- Can we get good code reuse without inheritance?



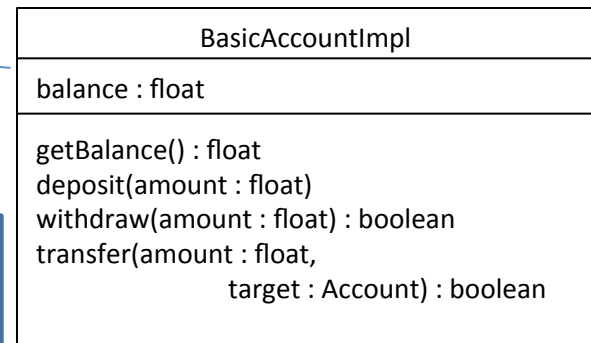
Yes! (Reuse via composition and delegation)



◆ -basicAcct

CheckingAccountImpl is
composed of a
BasicAccountImpl

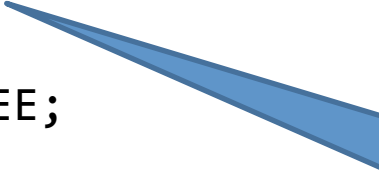
```
public class CheckingAccountImpl
    implements CheckingAccount {
    BasicAccountImpl basicAcct = new(...);
    public float getBalance() {
        return basicAcct.getBalance();
    }
    // ...
}
```



Java details: extended reuse with super

```
public abstract class AbstractAccount implements Account {  
    protected float balance = 0.0;  
    public boolean withdraw(float amount) {  
        // withdraws money from account (code not shown)  
    }  
}
```

```
public class ExpensiveCheckingAccountImpl  
    extends AbstractAccount implements CheckingAccount {  
    public boolean withdraw(float amount) {  
        balance -= HUGE_ATM_FEE;  
        boolean success = super.withdraw(amount)  
        if (!success)  
            balance += HUGE_ATM_FEE;  
        return success;  
    }  
}
```



Overrides `withdraw` but
also uses the superclass
`withdraw` method

Java details: constructors with `this` and `super`

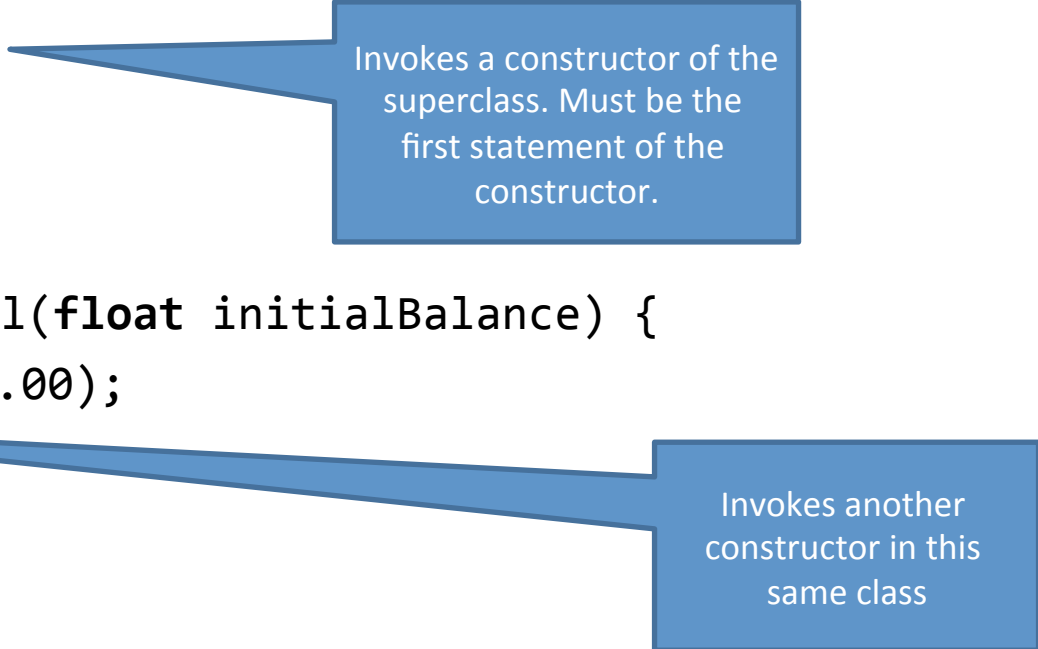
```
public class CheckingAccountImpl
    extends AbstractAccount implements CheckingAccount {

    private float fee;

    public CheckingAccountImpl(float initialBalance, float fee) {
        super(initialBalance);
        this.fee = fee;
    }

    public CheckingAccountImpl(float initialBalance) {
        this(initialBalance, 5.00);
    }
    /* other methods... */ }

```



The diagram consists of two blue callout boxes. The first callout box, pointing to the `super(initialBalance);` line, contains the text: "Invokes a constructor of the superclass. Must be the first statement of the constructor." The second callout box, pointing to the `this(initialBalance, 5.00);` line, contains the text: "Invokes another constructor in this same class".

Java details: `final`

- A final field: prevents reassignment to the field after initialization
- A final method: prevents overriding the method
- A final class: prevents extending the class
 - e.g., `public final class CheckingAccountImpl { ...`

Note: type-casting in Java

- Sometimes you want a different type than you have
 - e.g.,

```
float pi = 3.14;  
int indianaPi = (int) pi;
```
- Useful if you know you have a more specific subtype:
 - e.g.,

```
Account acct = ...;  
CheckingAccount checkingAcct =  
    (CheckingAccount) acct;  
float fee = checkingAcct.getFee();
```

 - Will get a `ClassCastException` if types are incompatible
- Advice: avoid downcasting types
 - Never(?) downcast within superclass to a subclass

Note: instanceof

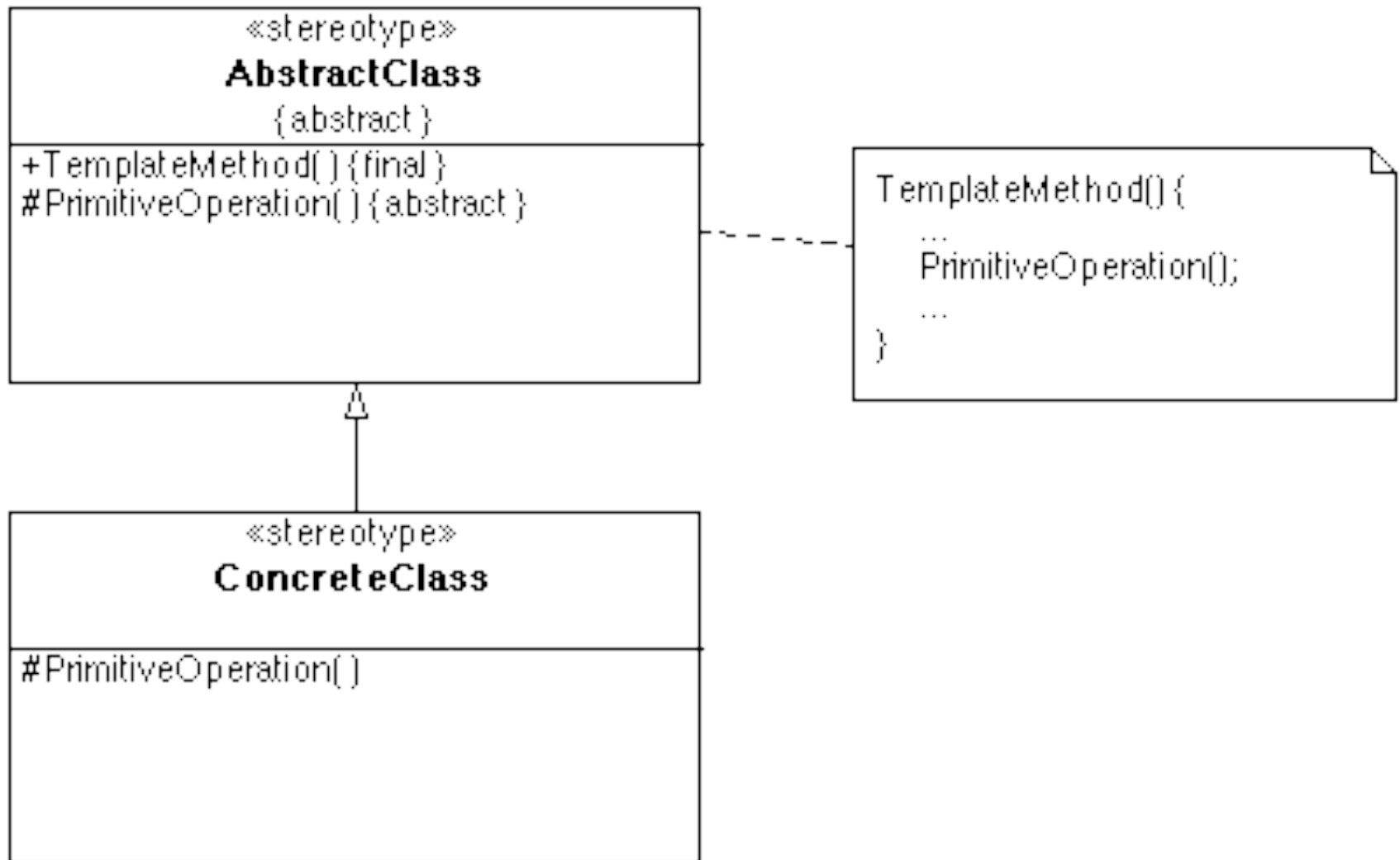
- Operator that tests whether an object is of a given class

```
public void doSomething(Account acct) {  
    float adj = 0.0;  
    if (acct instanceof CheckingAccount) {  
        checkingAcct = (CheckingAccount) acct;  
        adj = checkingAcct.getFee();  
    } else if (acct instanceof SavingsAccount) {  
        savingsAcct = (SavingsAccount) acct;  
        adj = savingsAcct.getInterest();  
    }  
    ...  
}
```

- Advice: avoid instanceof if possible
 - Never(?) use instanceof in a superclass to check type against subclass

THE TEMPLATE-METHOD DESIGN PATTERN

The *Template Method* design pattern



Note: instanceof

- Operator that tests whether an object is of a given class

```
public void doSomething(Account acct) {  
    float adj = 0.0;  
    if (acct instanceof CheckingAccount) {  
        checkingAcct = (CheckingAccount) acct;  
        adj = checkingAcct.getFee();  
    } else if (acct instanceof SavingsAccount) {  
        savingsAcct = (SavingsAccount) acct;  
        adj = savingsAcct.getInterest();  
    }  
    ...  
}
```

- Advice: avoid instanceof if possible
 - Never(?) use instanceof in a superclass to check type against subclass

Avoiding instanceof with the Template Method pattern

```
public interface Account {  
    ...  
    public float getMonthlyAdjustment();  
}
```

```
public class CheckingAccount implements Account {  
    ...  
    public float getMonthlyAdjustment() {  
        return getFee();  
    }  
}
```

```
public class SavingsAccount implements Account {  
    ...  
    public float getMonthlyAdjustment() {  
        return getInterest();  
    }  
}
```

Avoiding instanceof with the Template Method pattern

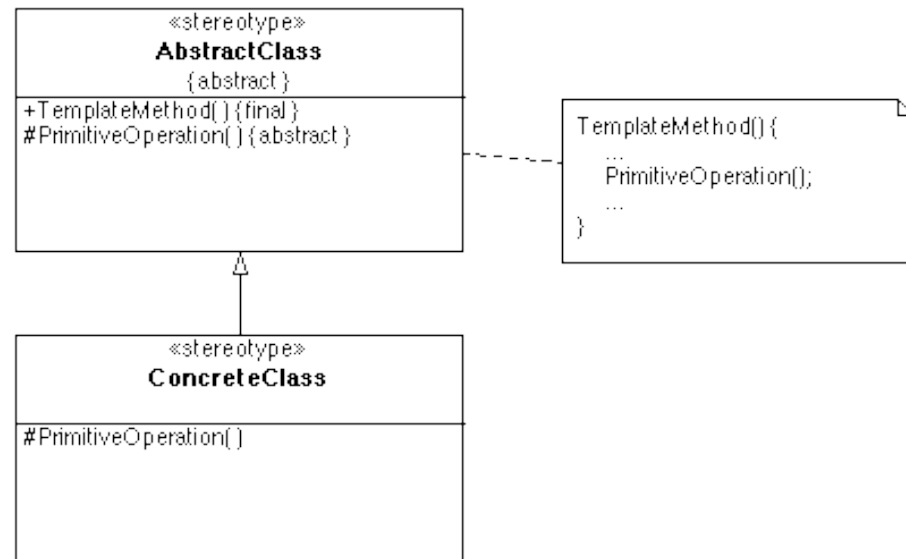
```
public void doSomething(Account acct) {  
    float adj = 0.0;  
    if (acct instanceof CheckingAccount) {  
        checkingAcct = (CheckingAccount) acct;  
        adj = checkingAcct.getFee();  
    } else if (acct instanceof SavingsAccount) {  
        savingsAcct = (SavingsAccount) acct;  
        adj = savingsAcct.getInterest();  
    }  
    ...  
}
```

Instead:

```
public void doSomething(Account acct) {  
    float adj = acct.getMonthlyAdjustment();  
    ...  
}
```

The *Template Method* design pattern

- Applicability
 - When an algorithm consists of varying and invariant parts that must be customized
 - When common behavior in subclasses should be factored and localized to avoid code duplication
 - To control subclass extensions to specific operations
- Consequences
 - Code reuse
 - Inverted “Hollywood” control: don’t call us, we’ll call you
 - Ensures the invariant parts of the algorithm are not changed by subclasses



Template Method vs. the Strategy Pattern

- Both support variations in larger common context
- Template method uses inheritance + an abstract method
- Strategy uses an interface and polymorphism (via object composition)
 - Strategy objects are reusable across multiple classes
 - Multiple strategy objects are possible per class

THE DECORATOR DESIGN PATTERN

Limitations of inheritance

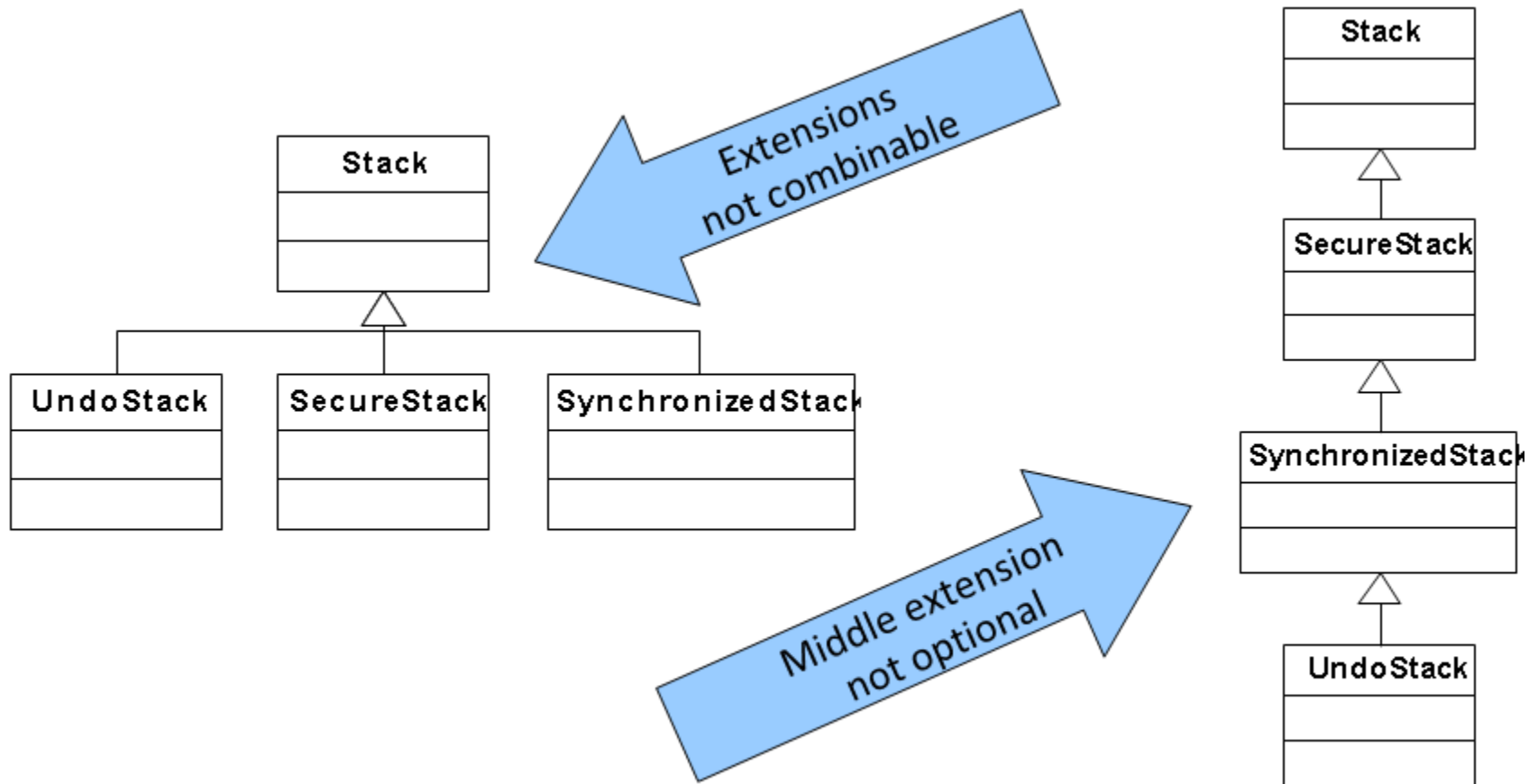
- Suppose you want various extensions of a Stack data structure...
 - UndoStack: A stack that lets you undo previous push or pop operations
 - SecureStack or LockedStack: A stack that requires a password to let you access the data
 - SynchronizedStack: A stack that serializes concurrent accesses

Limitations of inheritance

- Suppose you want various extensions of a Stack data structure...
 - UndoStack: A stack that lets you undo previous push or pop operations
 - SecureStack or LockedStack: A stack that requires a password to let you access the data
 - SynchronizedStack: A stack that serializes concurrent accesses
 - SecureUndoStack: A stack that requires a password to let you access the data, and also lets you undo previous operations
 - SynchronizedUndoStack: A stack that serializes concurrent accesses, and also lets you undo previous operations
 - SecureSynchronizedStack: ...
 - SecureSynchroniedUndoStack: ...

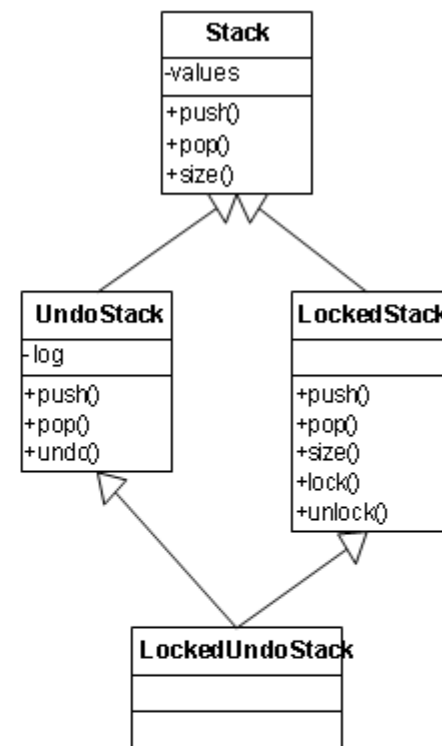
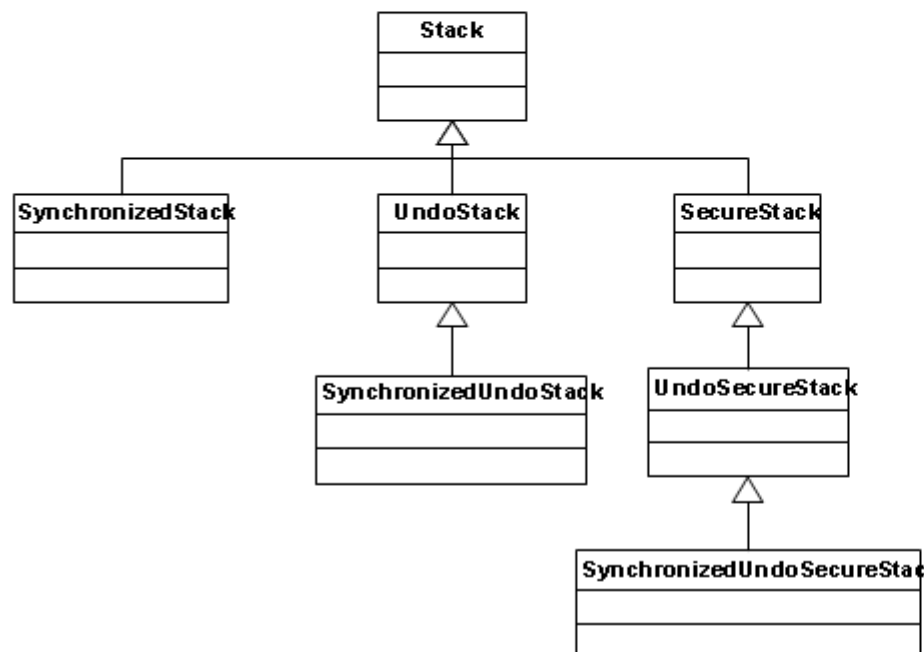
Goal: arbitrarily composable extensions

Limitations of inheritance



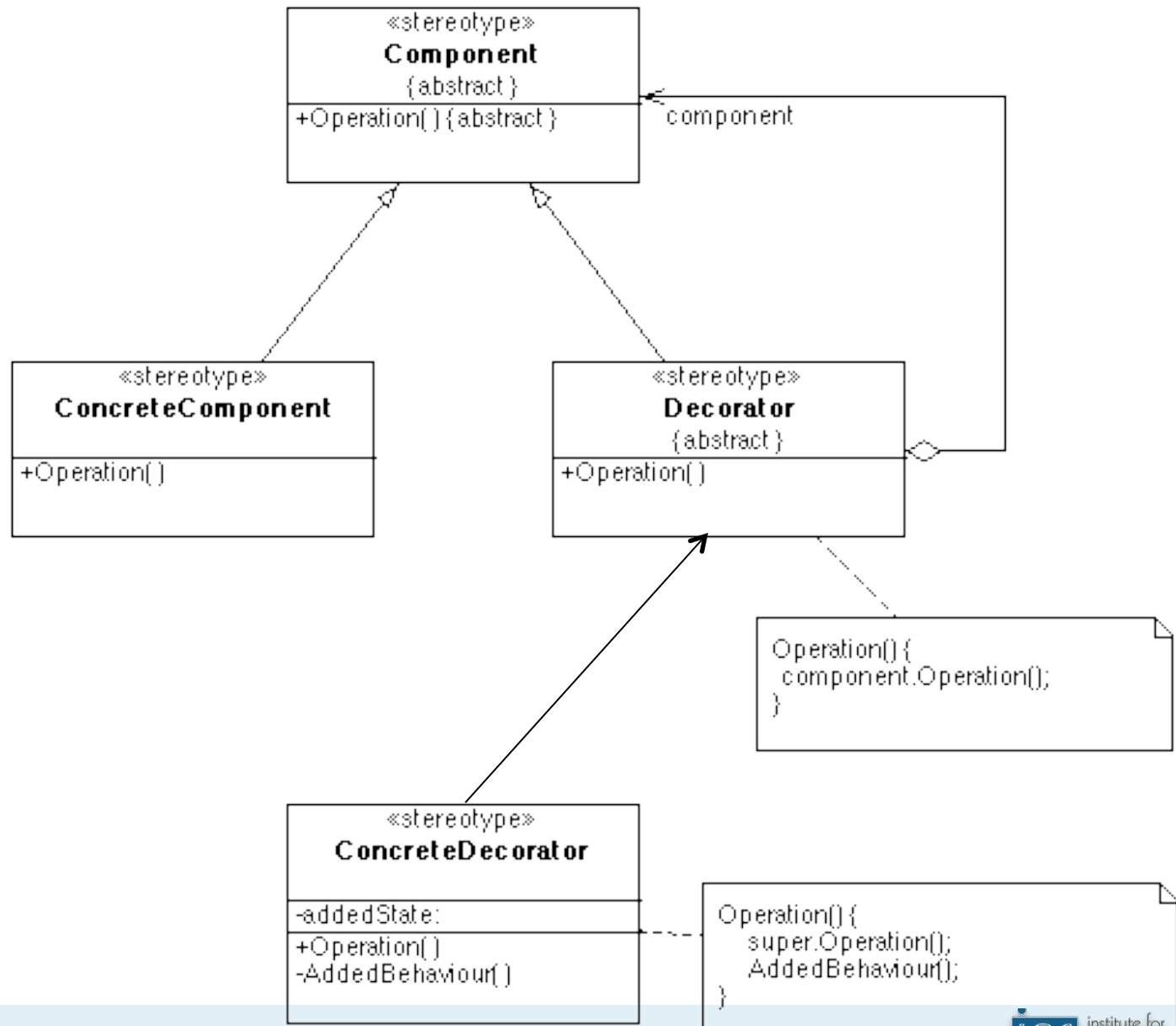
Workarounds?

- Combining inheritance hierarchies
 - Combinatorial explosion
 - Massive code replication



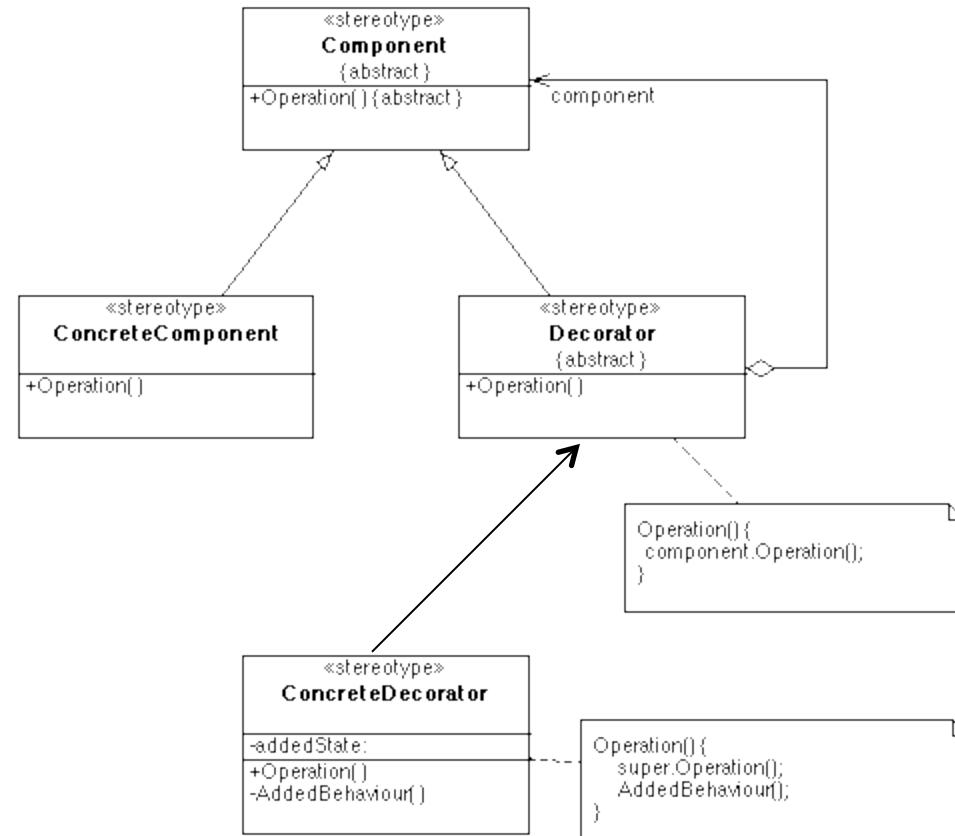
- Multiple inheritance
 - Diamond problem

The *Decorator* design pattern

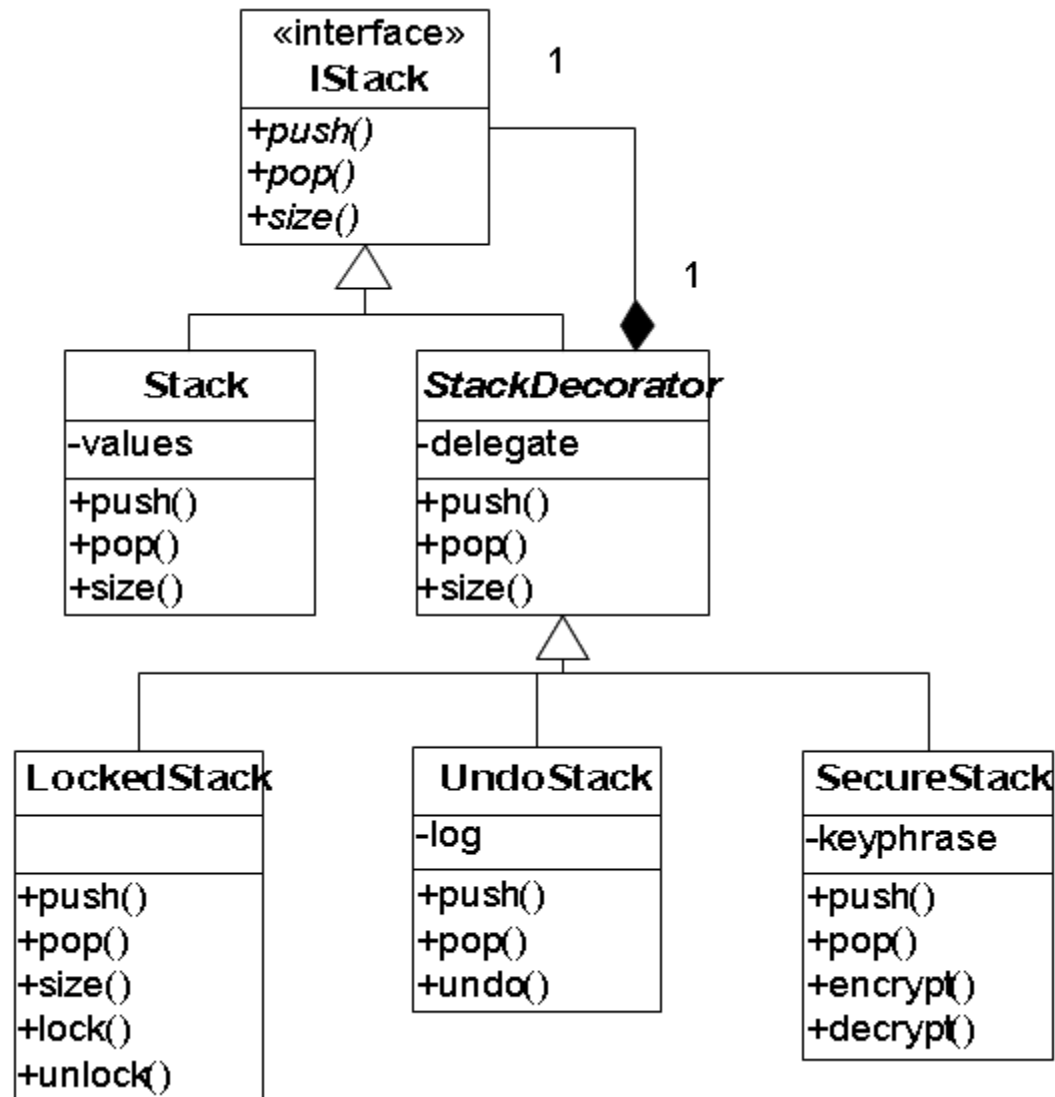


The *Decorator* design pattern

- Applicability
 - To add responsibilities to individual objects dynamically and transparently
 - For responsibilities that can be withdrawn
 - When extension by subclassing is impractical
- Consequences
 - More flexible than static inheritance
 - Avoids monolithic classes
 - Breaks object identity
 - Lots of little objects



Using the Decorator for our Stack example



Decorators from `java.util.Collections`

- Helper methods in `java.util.Collections`:
 - `static List<T> unmodifiableList(List<T> lst);`
 - `static Set<T> unmodifiableSet(Set<T> set);`
 - `static Map<K,V> unmodifiableMap(Map<K,V> map);`
- Turn a mutable list into an immutable list
 - All mutation operations on resulting list throw `UnsupportedOperationException`
- Similar for synchronization:
 - `static List<T> synchronizedList(List<T> lst);`
 - `static Set<T> synchronizedSet(Set<T> set);`
 - `static Map<K,V> synchronizedMap(Map<K,V> map);`

The UnmodifiableCollection (simplified excerpt)

```
public static <T> Collection<T> unmodifiableCollection(Collection<T> c)
    return new UnmodifiableCollection<>(c);
}
class UnmodifiableCollection<E> implements Collection<E>, Serializable
    final Collection<E> c;
    UnmodifiableCollection(Collection<> c) {this.c = c; }
    public int size() {return c.size();}
    public boolean isEmpty() {return c.isEmpty();}
    public boolean contains(Object o) {return c.contains(o);}
    public Object[] toArray() {return c.toArray();}
    public <T> T[] toArray(T[] a) {return c.toArray(a);}
    public String toString() {return c.toString();}
    public boolean add(E e) {throw new UnsupportedOperationException;}
    public boolean remove(Object o) { throw new UnsupportedOperationException;}
        public boolean containsAll(Collection<?> coll) { return
    public boolean addAll(Collection<? extends E> coll) { throw new
    public boolean removeAll(Collection<?> coll) { throw new Unsuppo
    public boolean retainAll(Collection<?> coll) { throw new Unsuppo
    public void clear() { throw new UnsupportedOperationException()}
}
```

The Decorator pattern vs. inheritance

- Composition at runtime vs compile-time
- Inheritance can add methods
- Decorator allows mix and match of extensions
- Can apply decorators multiple times
- Multiple inheritance has conceptual problems
- Inheritance produces single object, decorator consists of multiple collaborating objects
 - Multiple clients: Danger of inconsistent references, possibility of different perspectives

BEHAVIORAL SUBTYPING

Behavioral subtyping

Let $q(x)$ be a property provable about objects x of type T . Then $q(y)$ should be provable for objects y of type S where S is a subtype of T .

Barbara Liskov

- e.g., Compiler-enforced rules in Java:
 - Subtype (subclass, subinterface, object implementing interface) can add, but not remove methods
 - Overriding method must return same type or subtype
 - Overriding method must accept the same parameter types
 - Overriding method may not throw additional exceptions
 - Concrete class must implement all undefined interface methods and abstract methods

Behavioral subtyping

Let $q(x)$ be a property provable about objects x of type T . Then $q(y)$ should be provable for objects y of type S where S is a subtype of T .

Barbara Liskov

- e.g., Compiler-enforced rules in Java:
 - Subtype (subclass, subinterface, object implementing interface) can add, but not remove methods
 - Overriding method must return same type or subtype
 - Overriding method must accept the same parameter types
 - Overriding method may not throw additional exceptions
 - Concrete class must implement all undefined interface methods and abstract methods
- Also applies to behavior: A subclass must fulfill all contracts its superclass does:
 - Same or stronger invariants
 - Same or stronger postconditions for all methods
 - Same or weaker preconditions for all methods

**This is called
the Liskov
Substitution
Principle.**

Behavioral subtyping in a nutshell

- If `Cowboy.draw()` overrides `Circle.draw()` somebody gets hurt!



Behavioral subtyping (Liskov Substitution Principle)

```
abstract class Vehicle {  
    int speed, limit;  
  
    //@ invariant speed < limit;  
  
  
  
  
  
  
    //@ requires speed != 0;  
    //@ ensures speed < \old(speed)  
    void brake();  
}
```

```
class Car extends Vehicle {  
    int fuel;  
    boolean engineOn;  
    //@ invariant speed < limit;  
    //@ invariant fuel >= 0;  
  
    //@ requires fuel > 0 && !engineOn;  
    //@ ensures engineOn;  
    void start() { ... }  
  
    void accelerate() { ... }  
  
    //@ requires speed != 0;  
    //@ ensures speed < \old(speed)  
    void brake() { ... }  
}
```

Subclass fulfills the same invariants (and additional ones)
Overridden method has the same pre and postconditions

Behavioral subtyping (Liskov Substitution Principle)

```
class Car extends Vehicle {  
    int fuel;  
    boolean engineOn;  
    //@ invariant fuel >= 0;  
  
    //@ requires fuel > 0 && !engineOn;  
    //@ ensures engineOn;  
    void start() { ... }  
  
    void accelerate() { ... }  
  
    //@ requires speed != 0;  
    //@ ensures speed < old(speed)  
    void brake() { ... }  
}
```

```
class Hybrid extends Car {  
    int charge;  
    //@ invariant charge >= 0;  
  
    //@ requires (charge > 0 || fuel > 0)  
        && !engineOn;  
    //@ ensures engineOn;  
    void start() { ... }  
  
    void accelerate() { ... }  
  
    //@ requires speed != 0;  
    //@ ensures speed < \old(speed)  
    //@ ensures charge > \old(charge)  
    void brake() { ... }  
}
```

Subclass fulfills the same invariants (and additional ones)
Overridden method start has weaker precondition
Overridden method break has stronger postcondition

Behavioral subtyping (Liskov Substitution Principle)

```
class Rectangle {  
    int h, w;  
  
    Rectangle(int h, int w) {  
        this.h=h; this.w=w;  
    }  
  
    //methods  
}
```

```
class Square extends Rectangle {  
    Square(int w) {  
        super(w, w);  
    }  
}
```

Is this Square a behavioral subtype of Rectangle?

Behavioral subtyping (Liskov Substitution Principle)

```
class Rectangle {  
    int h, w;  
  
    Rectangle(int h, int w) {  
        this.h=h; this.w=w;  
    }  
  
    //methods  
}
```

```
class Square extends Rectangle {  
    Square(int w) {  
        super(w, w);  
    }  
}
```

**Is this Square a behavioral subtype of Rectangle?
(Yes.)**

Behavioral subtyping (Liskov Substitution Principle)

```
class Rectangle {  
    //@ invariant h>0 && w>0;  
    int h, w;  
  
    Rectangle(int h, int w) {  
        this.h=h; this.w=w;  
    }  
  
    //methods  
}
```

```
class Square extends Rectangle {  
    //@ invariant h>0 && w>0;  
    //@ invariant h==w;  
    Square(int w) {  
        super(w, w);  
    }  
}
```

Is this Square a behavioral subtype of Rectangle?

Behavioral subtyping (Liskov Substitution Principle)

```
class Rectangle {  
    //@ invariant h>0 && w>0;  
    int h, w;  
  
    Rectangle(int h, int w) {  
        this.h=h; this.w=w;  
    }  
  
    //methods  
}
```

```
class Square extends Rectangle {  
    //@ invariant h>0 && w>0;  
    //@ invariant h==w;  
    Square(int w) {  
        super(w, w);  
    }  
}
```

**Is this Square a behavioral subtype of Rectangle?
(Yes.)**

Behavioral subtyping (Liskov Substitution Principle)

```
class Rectangle {  
    //@ invariant h>0 && w>0;  
    int h, w;  
  
    Rectangle(int h, int w) {  
        this.h=h; this.w=w;  
    }  
  
    //@ requires factor > 0;  
    void scale(int factor) {  
        w=w*factor;  
        h=h*factor;  
    }  
}
```

```
class Square extends Rectangle {  
    //@ invariant h>0 && w>0;  
    //@ invariant h==w;  
    Square(int w) {  
        super(w, w);  
    }  
}
```

Is this Square a behavioral subtype of Rectangle?

Behavioral subtyping (Liskov Substitution Principle)

```
class Rectangle {  
    //@ invariant h>0 && w>0;  
    int h, w;  
  
    Rectangle(int h, int w) {  
        this.h=h; this.w=w;  
    }  
  
    //@ requires factor > 0;  
    void scale(int factor) {  
        w=w*factor;  
        h=h*factor;  
    }  
}
```

```
class Square extends Rectangle {  
    //@ invariant h>0 && w>0;  
    //@ invariant h==w;  
    Square(int w) {  
        super(w, w);  
    }  
}
```

Is this Square a behavioral subtype of Rectangle?
(Yes.)

Behavioral subtyping (Liskov Substitution Principle)

```
class Rectangle {  
    //@ invariant h>0 && w>0;  
    int h, w;  
  
    Rectangle(int h, int w) {  
        this.h=h; this.w=w;  
    }  
  
    //@ requires factor > 0;  
    void scale(int factor) {  
        w=w*factor;  
        h=h*factor;  
    }  
    //@ requires neww > 0;  
    void setWidth(int neww) {  
        w=neww;  
    }  
}
```

```
class Square extends Rectangle {  
    //@ invariant h>0 && w>0;  
    //@ invariant h==w;  
    Square(int w) {  
        super(w, w);  
    }  
}
```

Is this Square a behavioral subtype of Rectangle?

Behavioral subtyping (Liskov Substitution Principle)

```
class Rectangle {  
    //@ invariant h>0 && w>0;  
    int h, w;  
  
    Rectangle(int h, int w) {  
        this.h=h; this.w=w;  
    }  
  
    void scale(int factor) {  
        w=w*factor;  
        h=h*factor;  
    }  
  
    void setWidth(int neww) {  
        w=neww;  
    }  
}
```

```
class Square extends Rectangle {  
    //@ invariant h>0 && w>0;  
    //@ invariant h==w;  
    Square(int w) {  
        super(w, w);  
    }  
}
```

```
class GraphicProgram {  
    void scaleW(Rectangle r, int factor) {  
        r.setWidth(r.getWidth() * factor);  
    }  
}
```

void setWidth(int neww) { ← **Invalidates stronger invariant (w==h) in subclass**

Maybe? (If so, it's not a square...)

JAVA.LANG.OBJECT BEHAVIORAL CONTRACTS

The `java.lang.Object`

- All Java objects inherit from `java.lang.Object`
- Commonly-used/overridden public methods:

<code>String</code>	<code>toString()</code>
<code>boolean</code>	<code>equals(Object obj)</code>
<code>int</code>	<code>hashCode()</code>
<code>Object</code>	<code>clone()</code>

Overriding java.lang.Object's .equals

- The default Object.equals:

```
public class Object {  
    public boolean equals(Object obj) {  
        return this == obj;  
    }  
}
```

- An aside: Do you like:

```
public class CheckingAccountImpl  
    implements CheckingAccount {  
    @Override  
    public boolean equals(Object obj) {  
        return false;  
    }  
}
```

The `.equals(Object obj)` contract

- An equivalence relation
 - Reflexive: $\forall x \quad x.equals(x)$
 - Symmetric: $\forall x, y \quad x.equals(y)$ if and only if $y.equals(x)$
 - Transitive: $\forall x, y, z \quad x.equals(y)$ and $y.equals(z)$ implies $x.equals(z)$
- Consistent
 - Invoking `x.equals(y)` repeatedly returns the same value unless `x` or `y` is modified
- `x.equals(null)` is always false
- `.equals()` always terminates and is side-effect free

The `.hashCode()` contract

- Consistent
 - Invoking `x.hashCode()` repeatedly returns same value unless `x` is modified
- `x.equals(y)` implies `x.hashCode() == y.hashCode()`
 - The reverse implication is not necessarily true:
 - `x.hashCode() == y.hashCode()` does not imply `x.equals(y)`
- Advice: Override `.equals()` if and only if you override `.hashCode()`

The `.clone()` contract

- Returns a *deep copy* of an object
- Generally (but not required!):
 - `x.clone() != x`
 - `x.clone().equals(x)`

Conforming to behavioral contracts

- Complete to support object equality checks:

```
public class Person {  
    private String firstName;  
    private String lastName;  
    public Person(String name) {  
        this.firstName = name.split(" ")[0];  
        this.lastName = name.split(" ")[1];  
    }  
}
```

}

A lesson in equality

```
public class Point {  
    private final int x;  
    private final int y;  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

Recall: The `java.lang.Object`

- All Java objects inherit from `java.lang.Object`
- Commonly-used/overridden public methods:
 - `String` `toString()`
 - `boolean` `equals(Object obj)`
 - `int` `hashCode()`
 - `Object` `clone()`

Complete to support equality-checking for the `Point` class.

A tempting but incorrect solution

```
public class Point {  
    private final int x;  
    private final int y;  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

```
public boolean equals(Point p) {  
    return x == p.x && y == p.y;  
}
```

Types must match

Recall: The java.lang.Object

- All Java objects inherit from java.lang.Object
- Commonly-used/overridden public methods:
 - String toString()
 - boolean equals(Object obj)
 - int hashCode()
 - Object clone()

boolean equals(Point p) does not override
boolean equals(Object obj)

A correct solution

```
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

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

    public int hashCode() {
        return 31*x + y;
    }
}
```

The `.equals(Object obj)` contract

- An equivalence relation

- Reflexive: $\forall x \quad x.equals(x)$
- Symmetric: $\forall x, y \quad x.equals(y)$ if and only if $y.equals(x)$
- Transitive: $\forall x, y, z \quad x.equals(y)$ and $y.equals(z)$ implies $x.equals(z)$

- Consistent

- Invoking `x.equals(y)` repeatedly returns the same value unless `x` or `y` is modified

- `x.equals(null)` is always false

A new challenge

```
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

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

```
public class ColorPoint
    extends Point {
    private final Color color;

    public ColorPoint(int x,
                      int y,
                      Color color) {
        super(x, y);
        this.color = color;
    }
}
```

Implement `.equals` for the `ColorPoint` class.
You may assume `Color` correctly implements `.equals`

A tempting solution

```
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

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

```
public class ColorPoint
    extends Point {
    private final Color color;

    public ColorPoint(int x,
                      int y,
                      Color color) {
        super(x, y);
        this.color = color;
    }

    public boolean equals(Object obj) {
        if (!(obj instanceof ColorPoint))
            return false;
        ColorPoint cp = (ColorPoint) obj;
        return super.equals(cp) &&
            color.equals(cp.color);
    }
}
```

A tempting solution

```
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

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

```
public class ColorPoint
    extends Point {
    private final Color color;

    public ColorPoint(int x,
                      int y,
                      Color color) {
        super(x, y);
        this.color = color;
    }

    public boolean equals(Object obj) {
        if (!(obj instanceof ColorPoint))
            return false;
        ColorPoint cp = (ColorPoint) obj;
        return super.equals(cp) &&
            color.equals(cp.color);
    }
}
```

A problem: `p.equals(cp)`
but `!cp.equals(p)`:

```
Point p = new Point(2, 42);
ColorPoint cp = new ColorPoint(2, 42, Color.BLUE);
```

More problems

```
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

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

```
public class ColorPoint
    extends Point {
    private final Color color;

    public ColorPoint(int x,
                      int y,
                      Color color) {
        super(x, y);
        this.color = color;
    }

    public boolean equals(Object obj) {
        if (!(obj instanceof Point))
            return false;
        if (!(obj instanceof ColorPoint))
            return super.equals(obj);
        ColorPoint cp = (ColorPoint) obj;
        return super.equals(cp) &&
            color.equals(cp.color);
    }
}
```

Consider:

```
Point p = new Point(2, 42);
ColorPoint cp1 = new ColorPoint(2, 42, Color.BLUE);
ColorPoint cp2 = new ColorPoint(2, 42, Color.MAUVE);
```

An abstract solution

```
public abstract class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

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

```
public class ColorPoint
    extends Point {
    private final Color color;

    public ColorPoint(int x,
                      int y,
                      Color color) {
        super(x, y);
        this.color = color;
    }

    public boolean equals(Object obj) {
        if (!(obj instanceof ColorPoint))
            return false;
        ColorPoint cp = (ColorPoint) obj;
        return super.equals(cp) &&
            color.equals(cp.color);
    }
}
```

```
public class PointImpl extends Point {
    public PointImpl(int x, int y) { super(x,y); }
    public boolean equals(Object obj) {
        if (!(obj instanceof PointImpl))
            return false;
        return super.equals(obj);
    }
}
```


The lesson

- Conforming to behavioral contracts can be difficult
- Advice:
 - Don't allow equality between distinct types
 - Be careful when inheriting from a concrete class

"Overriding the equals method seems simple, but there are many ways to get it wrong and the consequences can be dire." -- Josh Bloch

The lesson

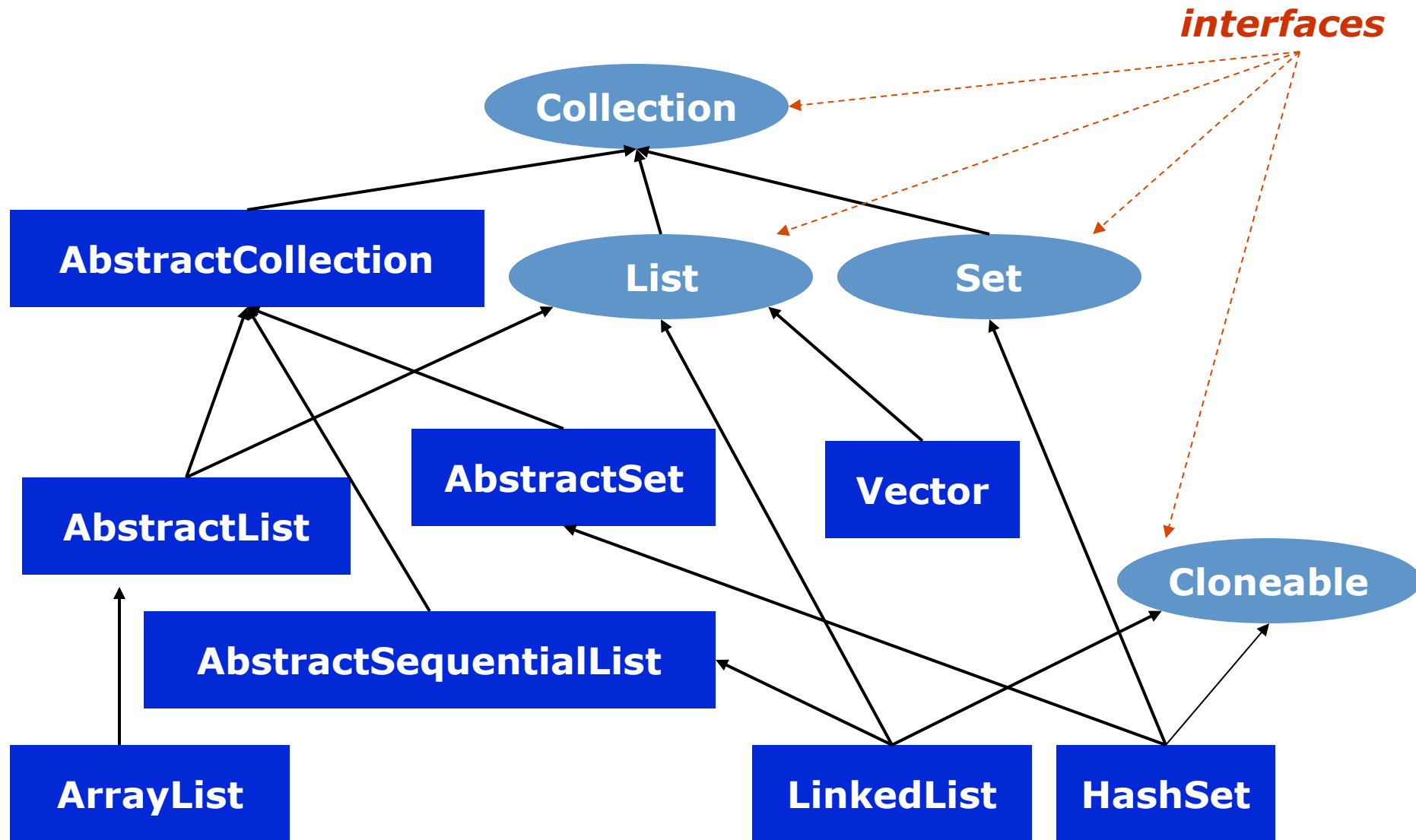
- Conforming to behavioral contracts can be difficult
- Advice:
 - Don't allow equality between distinct types
 - Be careful when inheriting from a concrete class
- Symmetry kills:

```
public class EvilButTrue {  
    public boolean equals(Object obj) {  
        return obj != null;  
    }  
    public int hashCode() {  
        return 0;  
    }  
}
```

"Overriding the equals method seems simple, but there are many ways to get it wrong and the consequences can be dire." -- Josh Bloch

PARAMETRIC POLYMORPHISM (GENERICS)

Recall the Java Collection API (excerpt)



Consider the `java.util.Stack`

```
public class Stack {  
    public void push(Object obj) { ... }  
    public Object pop() { ...}  
}
```

- Some possible client code?:

```
Stack stack = new Stack();  
String s = "Hello!";  
stack.push(s);  
String t = stack.pop();
```

Consider the `java.util.Stack`

```
public class Stack {  
    public void push(Object obj) { ... }  
    public Object pop() { ...}  
}
```

- Some possible client code:

```
Stack stack = new Stack();  
String s = "Hello!";  
stack.push(s);  
String t = (String) stack.pop();
```



**To fix the type error
with a downcast
(urgs!)**

Parametric polymorphism via Java Generics

- *Parametric polymorphism* is the ability to define a type generically to allow static type-checking without fully specifying types
- The `java.util.Stack` instead
 - A stack of some type T :

```
public class Stack<T> {  
    public void push(T obj) { ... }  
    public T pop() { ... }  
}
```
- Improves typechecking, simplifies(?) client code:

```
Stack<String> stack = new Stack<String>();  
String s = "Hello!";  
stack.push(s);  
String t = stack.pop();
```

Many Java Generics details

- Can have multiple type parameters
 - e.g., `Map<Integer,String>`
- Wildcards
 - e.g., `ArrayList<?>` or `ArrayList<? extends Animal>`
- Subtyping
 - `ArrayList<String>` is a subtype of `List<String>`
 - `ArrayList<String>` is not a subtype of `ArrayList<Object>`
- Cannot create Generic arrays
`List<String>[] foo = new List<String>[42];` // won't compile
- Type erasure
 - Generic type info is compile-time only
 - Cannot use `instanceof` to check generic type
 - But shouldn't use `instanceof` anyway

Summary: Designing Reusable Classes

- Reuse implementations with clear contracts (information hiding)
- Provide variation points in classes for reuse
 - Strategy Pattern + Composition
 - Overriding / Template Method Pattern + Inheritance
 - Decorator Pattern <- Adding behavior through delegation
 - Type parameters (generics)