Principles of Software Construction: Objects, Design, and Concurrency
(Part 1: Designing Classes)

Design for Reuse (class level)

Christian Kästner **Bogdan Vasilescu**

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

- Reading assignment due today
- Homework 2 due Thursday

FindBugs

- We will enable in HW3

- IDE plugin
- build.gradle: `apply plugin: 'findbugs'`

```
43                  }
44
45                  @Override
46                  public void run() {
47                          Lock localLock = new ReentrantLock();
48                          l.lock();
49                          int a = 1;
50                          localLock.lock();
51
52                          if (a == 2) {
53                                  l.unlock();
54                          } else {
55                                  // do nothing
56                          }
57                          return;
58                  }
59          }
```

Proble | Javad | Declar | Search | Consol | Call Hi | Analysi | Debug

0 errors, 12 warnings, 0 others

Description

- Iterator is a raw type. References to generic type Iterator<E> should be parameterized
- Iterator is a raw type. References to generic type Iterator<E> should be parameterized
- No required execution environment has been set
- plugin.ProgramPoint defines equals and uses Object.hashCode() [Troubling(14), High confidence]
- tests.NoUnlock$T3.run() does not release lock on all paths [Troubling(12), High confidence]
- tests.NoUnlock$T4.run() might ignore java.lang.Exception [Troubling(14), High confidence]
- Type safety: Unchecked cast from Object to Map.Entry<String,ProgramPoint.LockState>

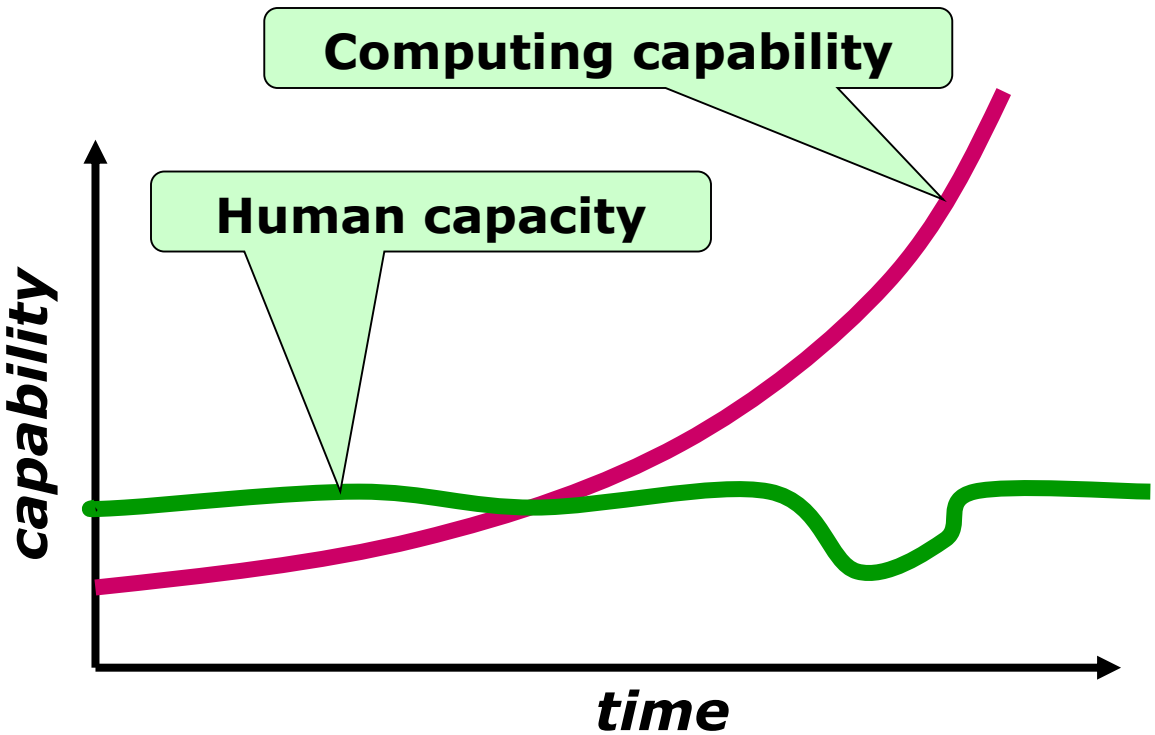
tests.NoUnlock$T3.run() does not release...l paths [Troubling(12), High confidence]

Review

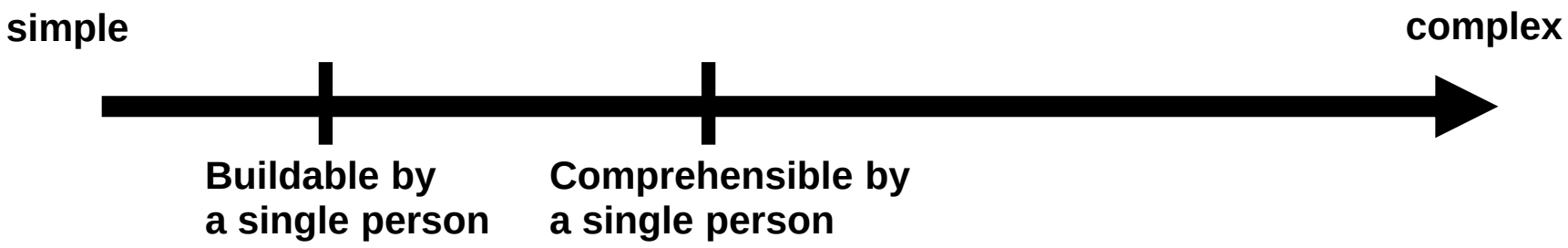
Learning goals for today

- 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
- Identify applicability of and apply the template method design patterns
- Read and write UML interaction diagrams

The limits of exponentials

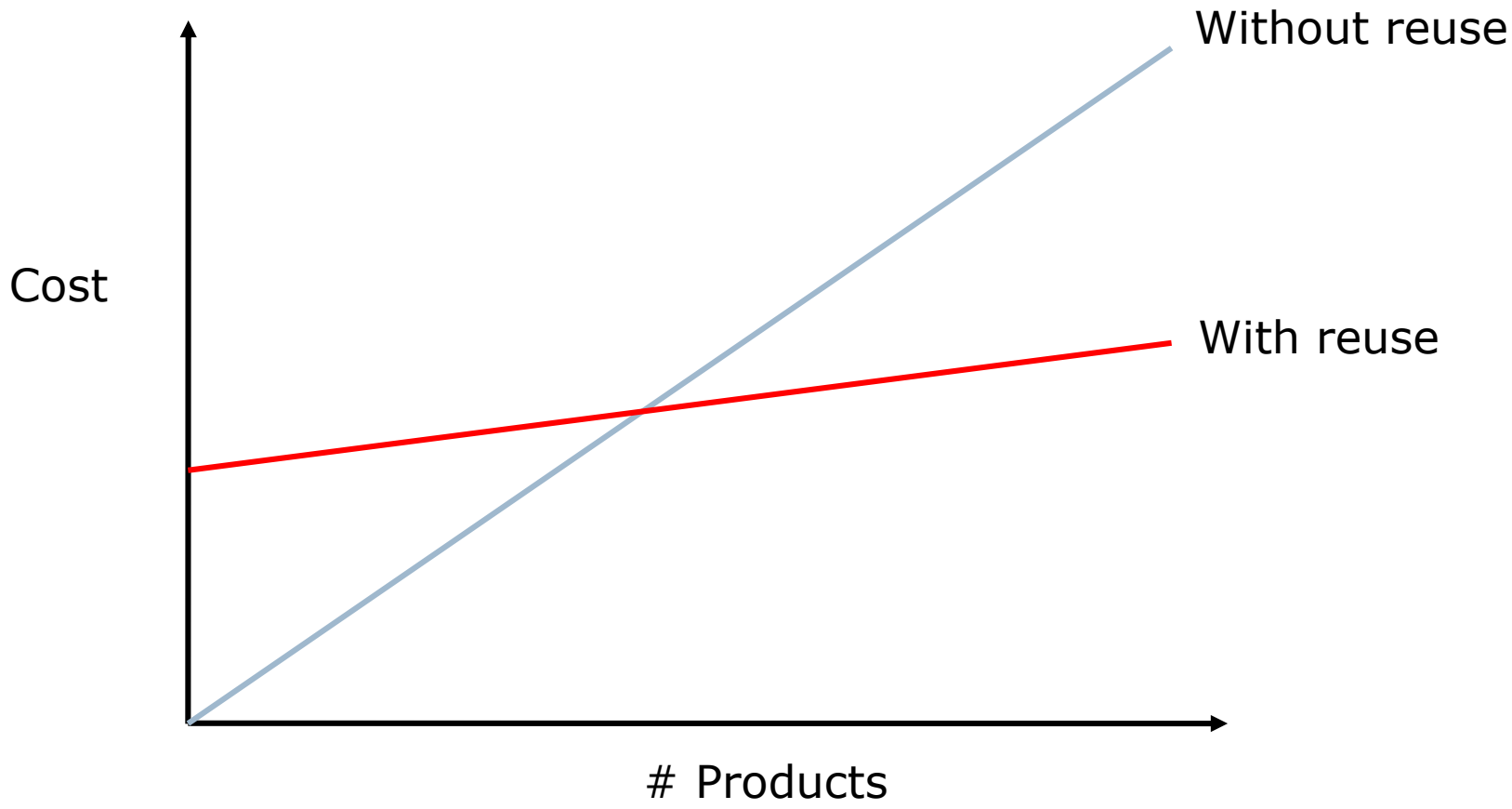


Building complex systems

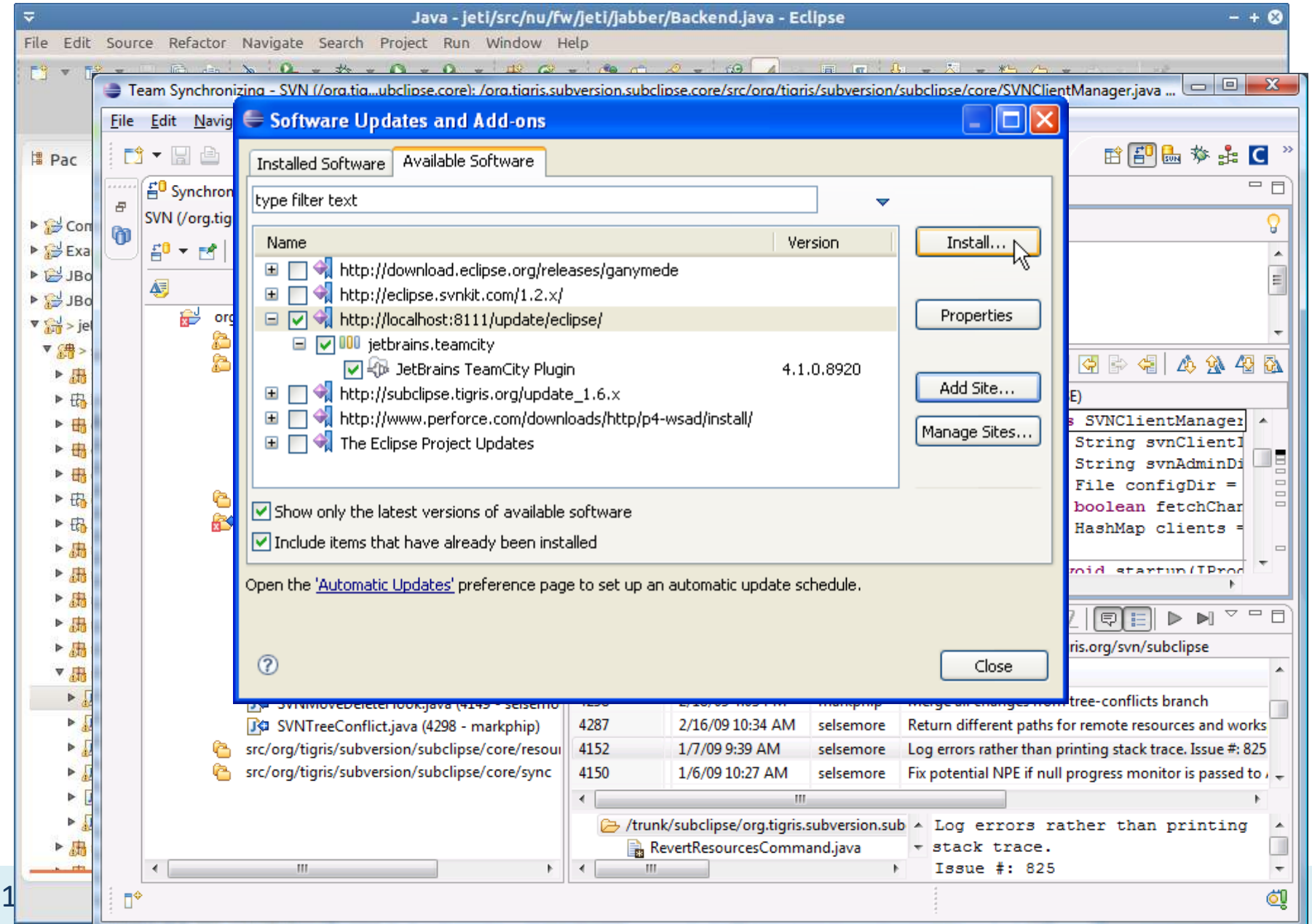


- Division of labor
- Division of knowledge and design effort
- Reuse of existing implementations

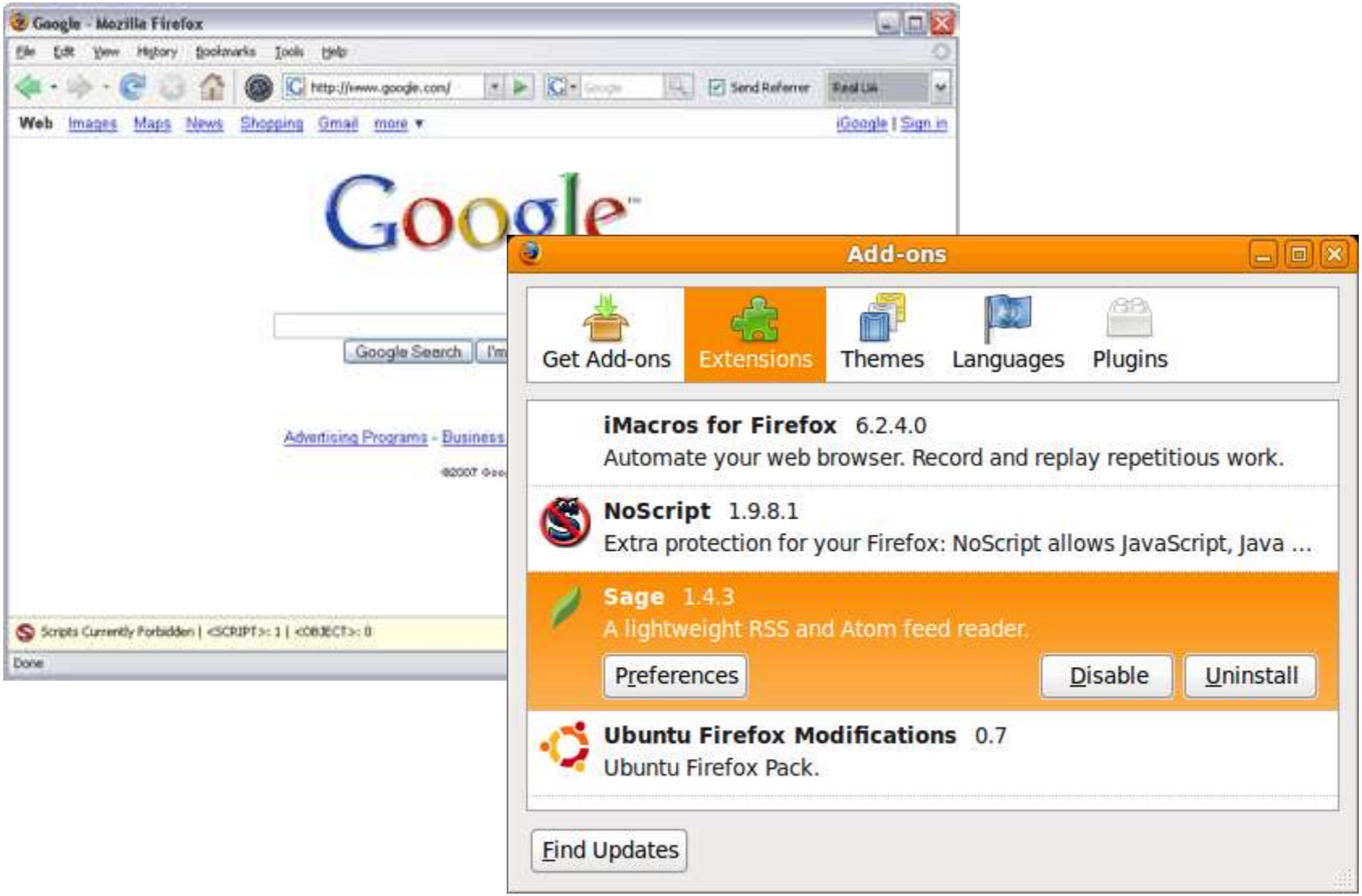
The promise of reuse:



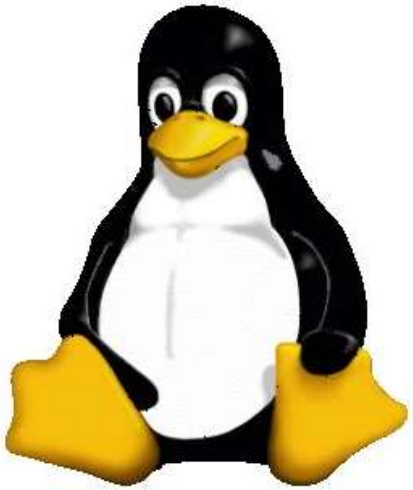
Reuse: Family of development tools



Reuse: Web browser extensions



Reuse and variation: Flavors of Linux



Today: Class-level reuse with delegation and inheritance

- Delegation
- Inheritance
  - Java-specific details for inheritance
- Behavioral subtyping: Liskov's Substitution Principle

- Today and Thursday:
  - Design patterns for improved class-level reuse
- Later in the course:
  - System-level reuse with libraries and frameworks

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
  - e.g. here, the `Sorter` is delegating functionality to some `Comparator`
- Judicious delegation enables code reuse

```
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
  - e.g. here, the `Sorter` is delegating functionality to some `Comparator`
- 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

```
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; }}
```

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 a list that logs its operations to the console…

Using delegation to extend functionality

- One solution:

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

```
public class LoggingList<E> implements List<E> {
  private final List<E> list;

  public LoggingList<E>(List<E> list) { this.list = list; }

  public boolean add(E e) {
      System.out.println("Adding " + e);
      return list.add(e);
  }

  public E remove(int index) {
      System.out.println("Removing at " + index);
      return list.remove(index);
  }
  …
```

Aside: A sequence diagram for the `LoggingList`

Delegation and design

- Small interfaces with clear contracts
- Classes to encapsulate algorithms, behaviors
  - E.g., the `Comparator`

IMPLEMENTATION INHERITANCE AND ABSTRACT CLASSES

Variation in the real world: types of bank accounts

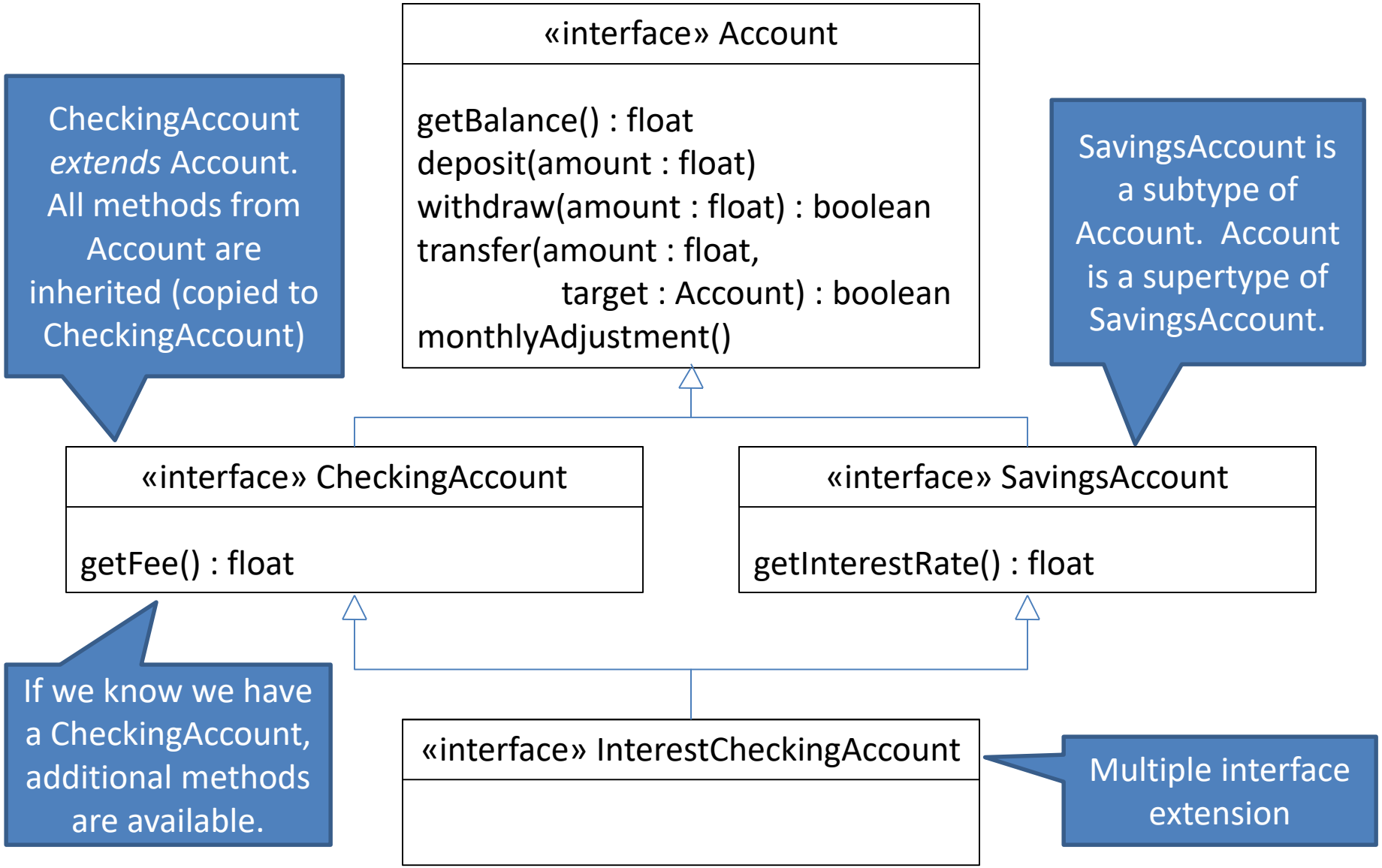
«interface» CheckingAccount
getBalance() : float deposit(amount : float) withdraw(amount : float) : boolean transfer(amount : float, target : Account) : boolean getFee() : float

«interface» SavingsAccount
getBalance() : float deposit(amount : float) withdraw(amount : float) : boolean transfer(amount : float, target : Account) : boolean getInterestRate() : float

Variation in the real world: types of bank accounts

```
public interface CheckingAccount {
    public long getBalance();
    public void  deposit(long amount);
    public boolean withdraw(long amount);
    public boolean transfer(long amount, Account??? target);
    public long getFee();
}


public interface SavingsAccount {
    public long getBalance();
    public void  deposit(long amount);
    public boolean withdraw(long amount);
    public boolean transfer(long amount, Account??? target);
    public double getInterestRate();
}
```

Better: Interface inheritance for an account type hierarchy



Interface inheritance for an account type hierarchy

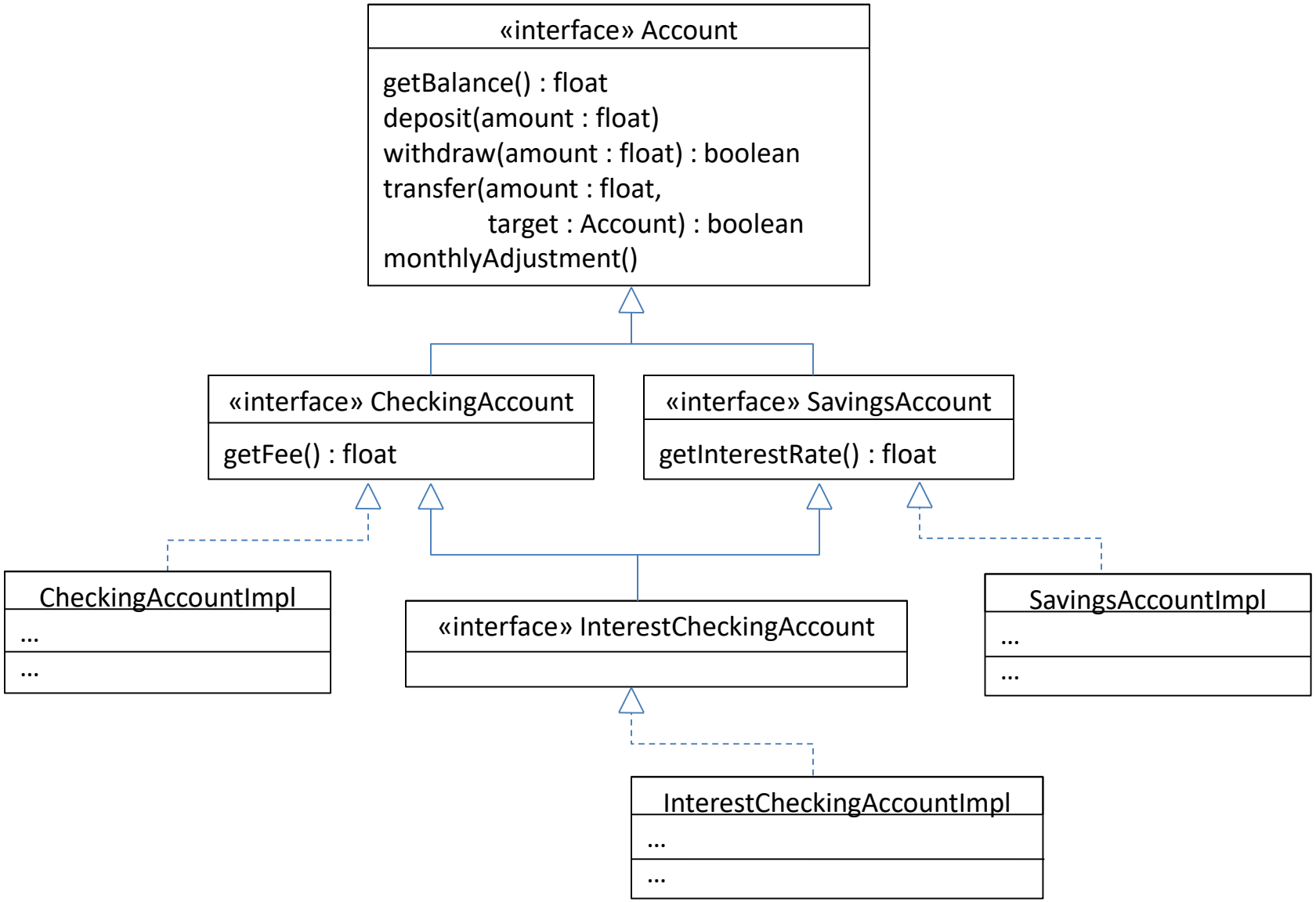
```
public interface Account {
    public long getBalance();
    public void  deposit(long amount);
    public boolean withdraw(long amount);
    public boolean transfer(long amount, Account target);
    public void monthlyAdjustment();
}

public interface CheckingAccount extends Account {
    public long getFee();
}

public interface SavingsAccount extends Account {
    public double getInterestRate();
}

public interface InterestCheckingAccount
                        extends CheckingAccount, SavingsAccount {
}
```

The power of object-oriented interfaces

- Subtype polymorphism
  - Different kinds of objects can be treated uniformly by client code
  - Each object behaves according to its type
    - e.g., if you add new kind of account, client code does not change:

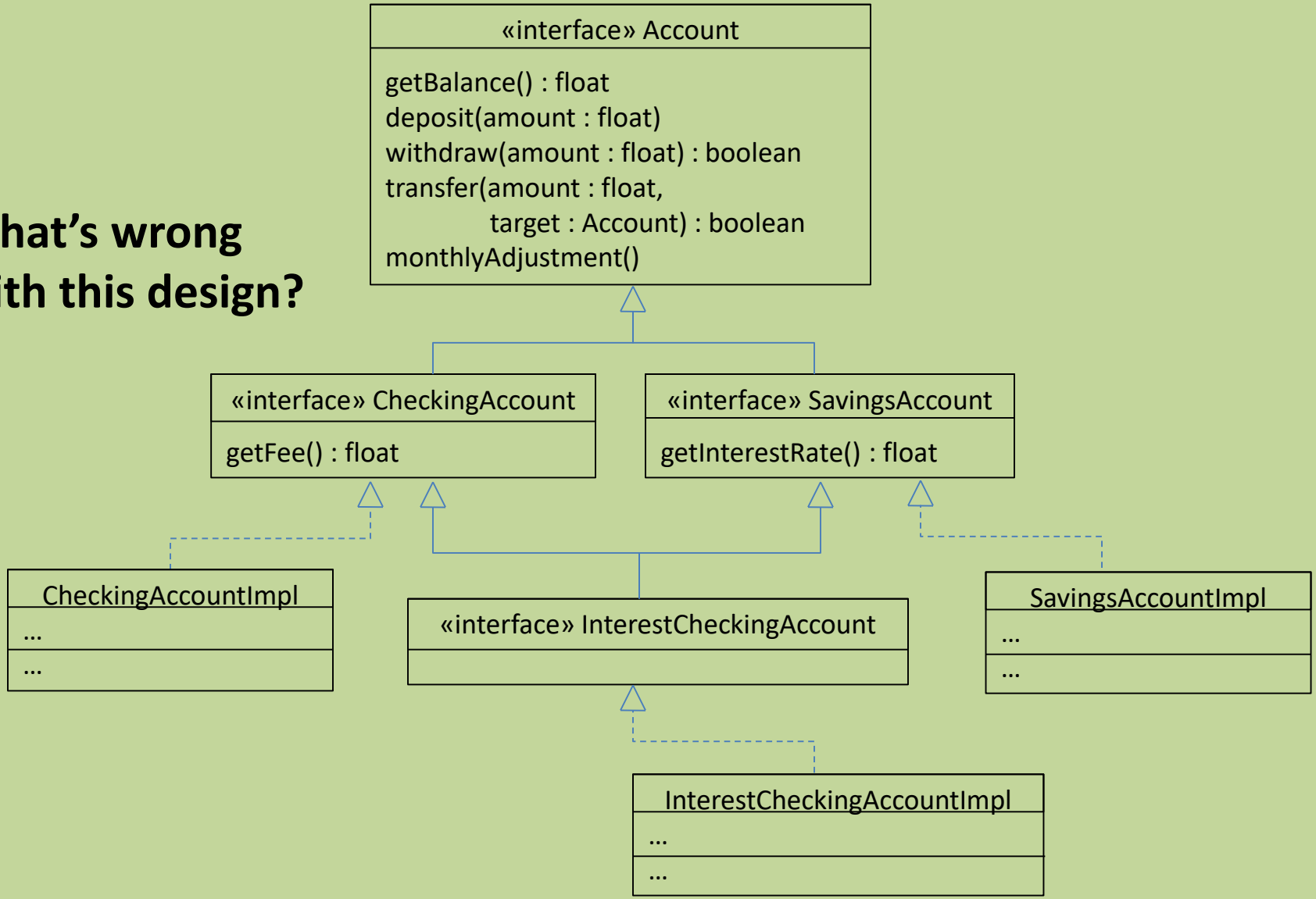
```
If today is the last day of the month:
   For each acct in allAccounts:
      acct.monthlyAdjustment();
```

Implementation inheritance for code reuse



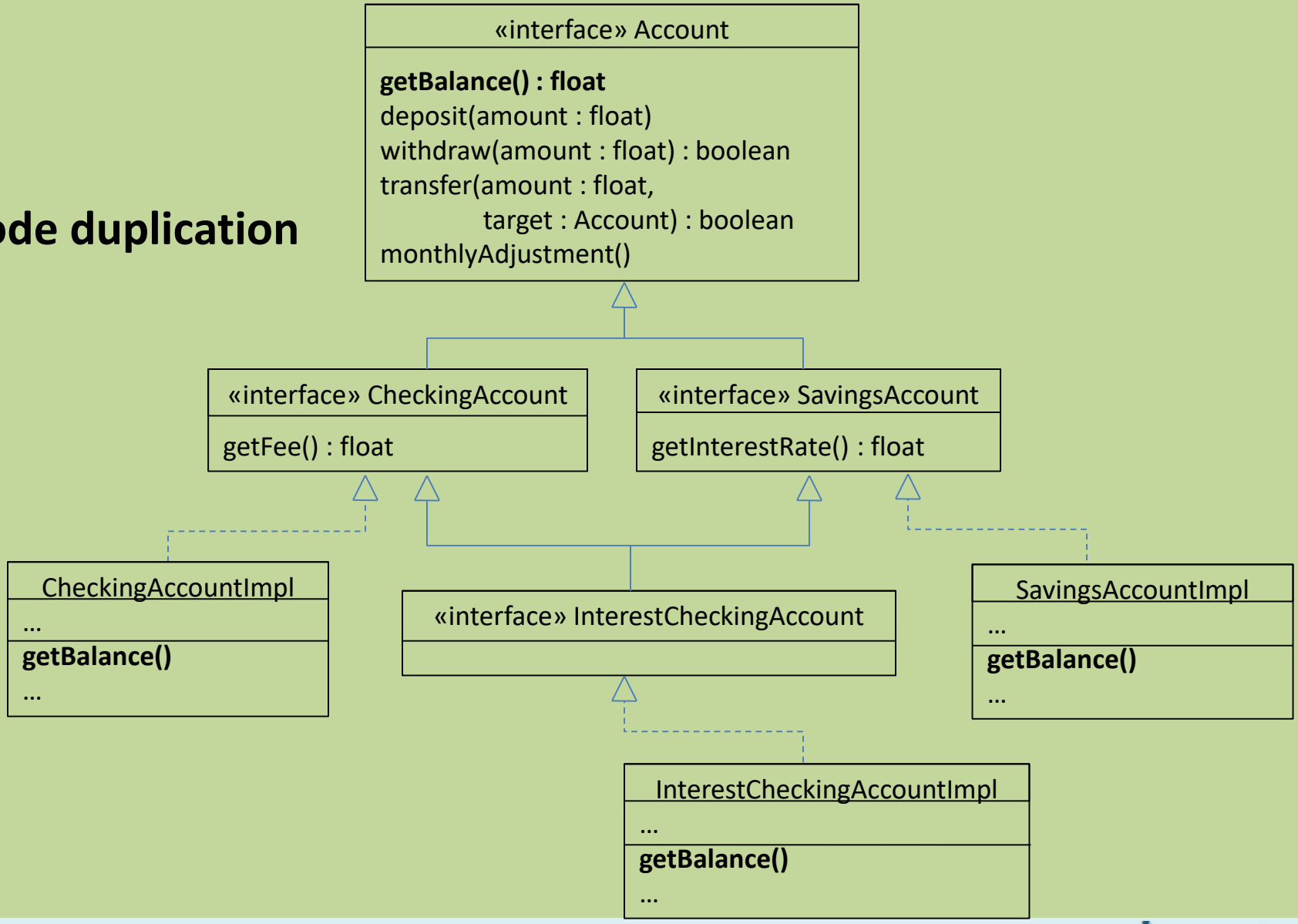
Implementation inheritance for code reuse

What's wrong with this design?



Implementation inheritance for code reuse

Code duplication

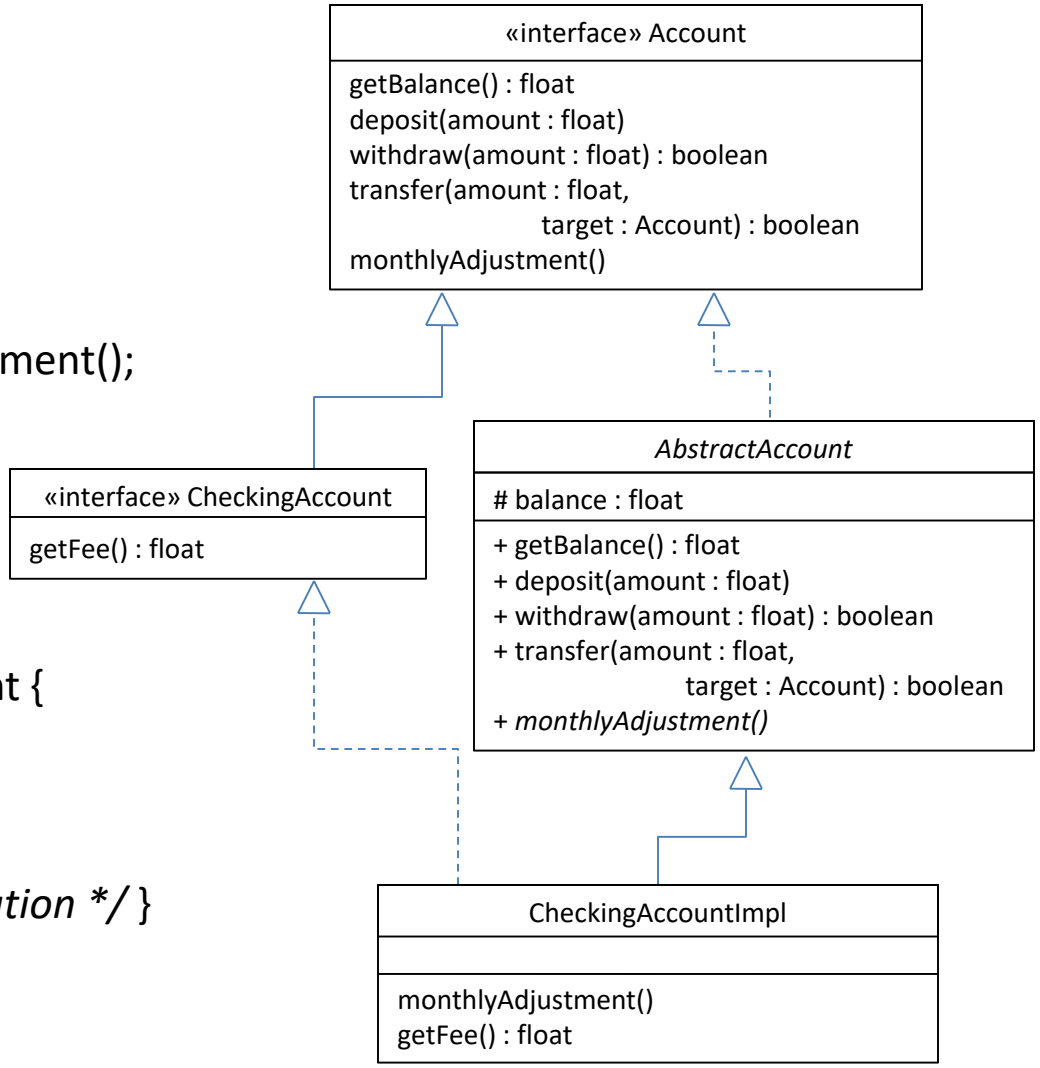


Implementation inheritance for code reuse

```
public abstract class AbstractAccount
         implements Account {
   protected long balance = 0;
   public long getBalance() {
         return balance;
   }
   abstract public void monthlyAdjustment();
   // other methods…
}

public class CheckingAccountImpl
         extends AbstractAccount
         implements CheckingAccount {
   public void monthlyAdjustment() {
         balance -= getFee();
   }
   public long getFee() { … }
}
```

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



Better: Reuse abstract account code

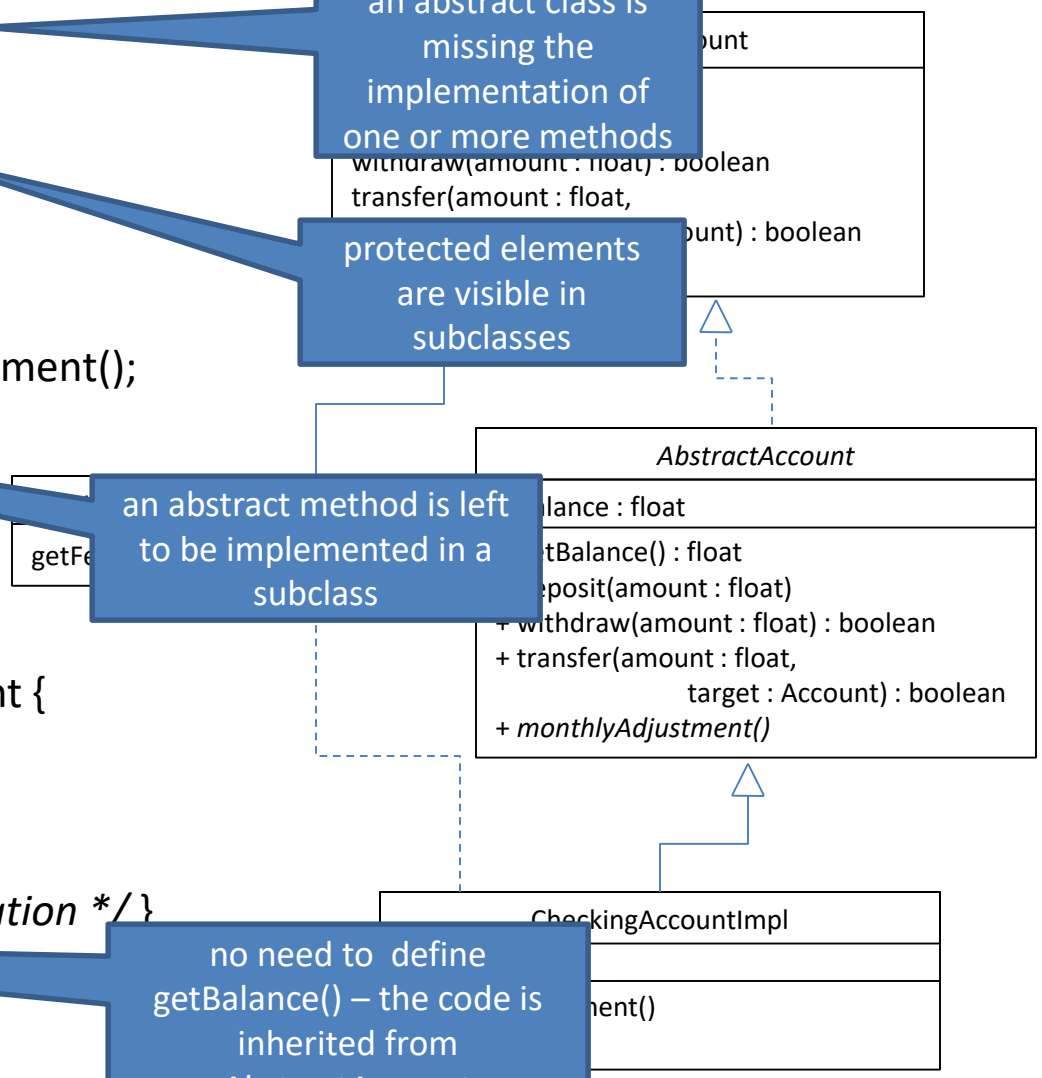
```
public abstract class AbstractAccount
        implements Account {
    protected float balance = 0.0;
    public float getBalance() {
        return balance;
    }
    abstract public void monthlyAdjustment();
    // other methods...
}

public class CheckingAccountImpl
        extends AbstractAcount
        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

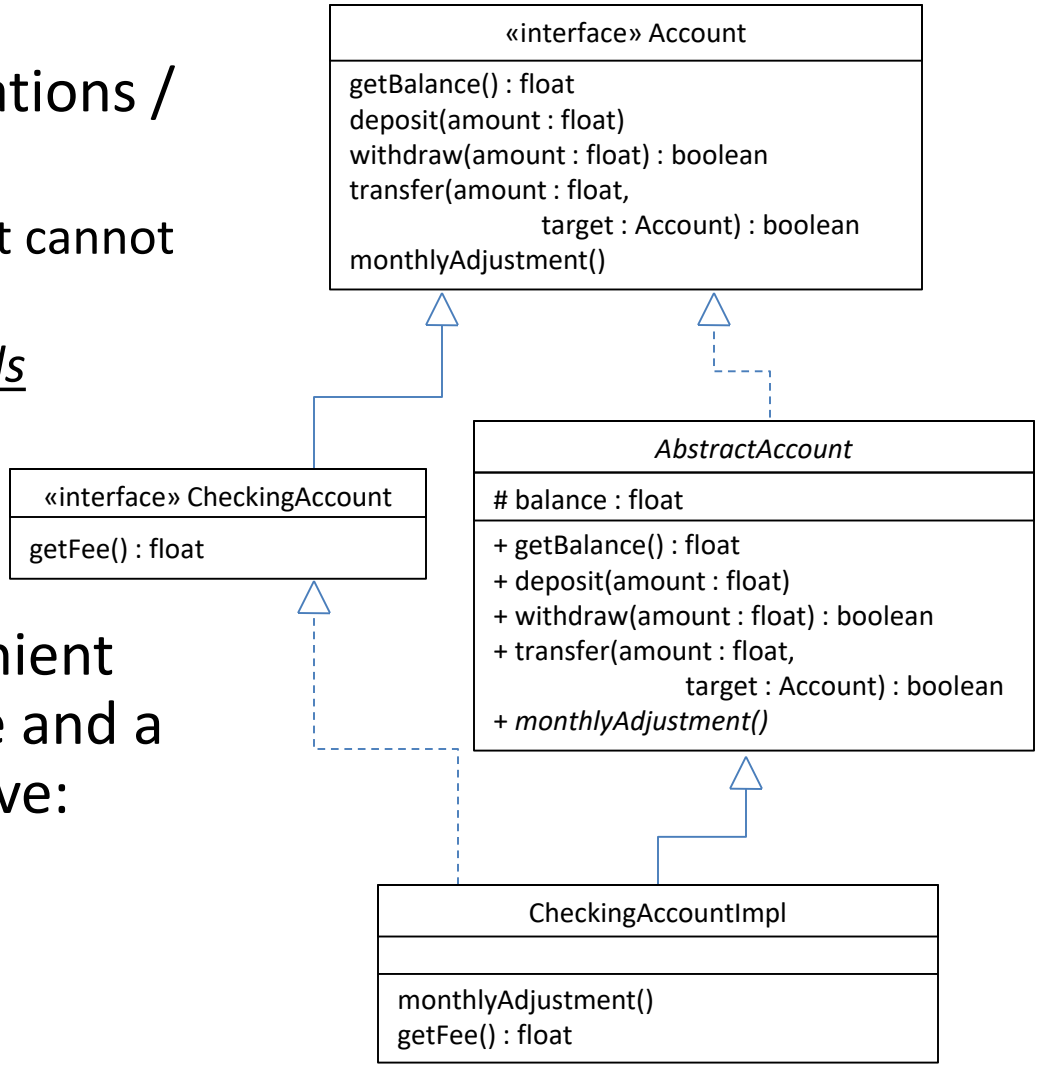
no need to define getBalance() – the code is inherited from AbstractAccount



Interfaces vs Abstract Classes vs Concrete Classes

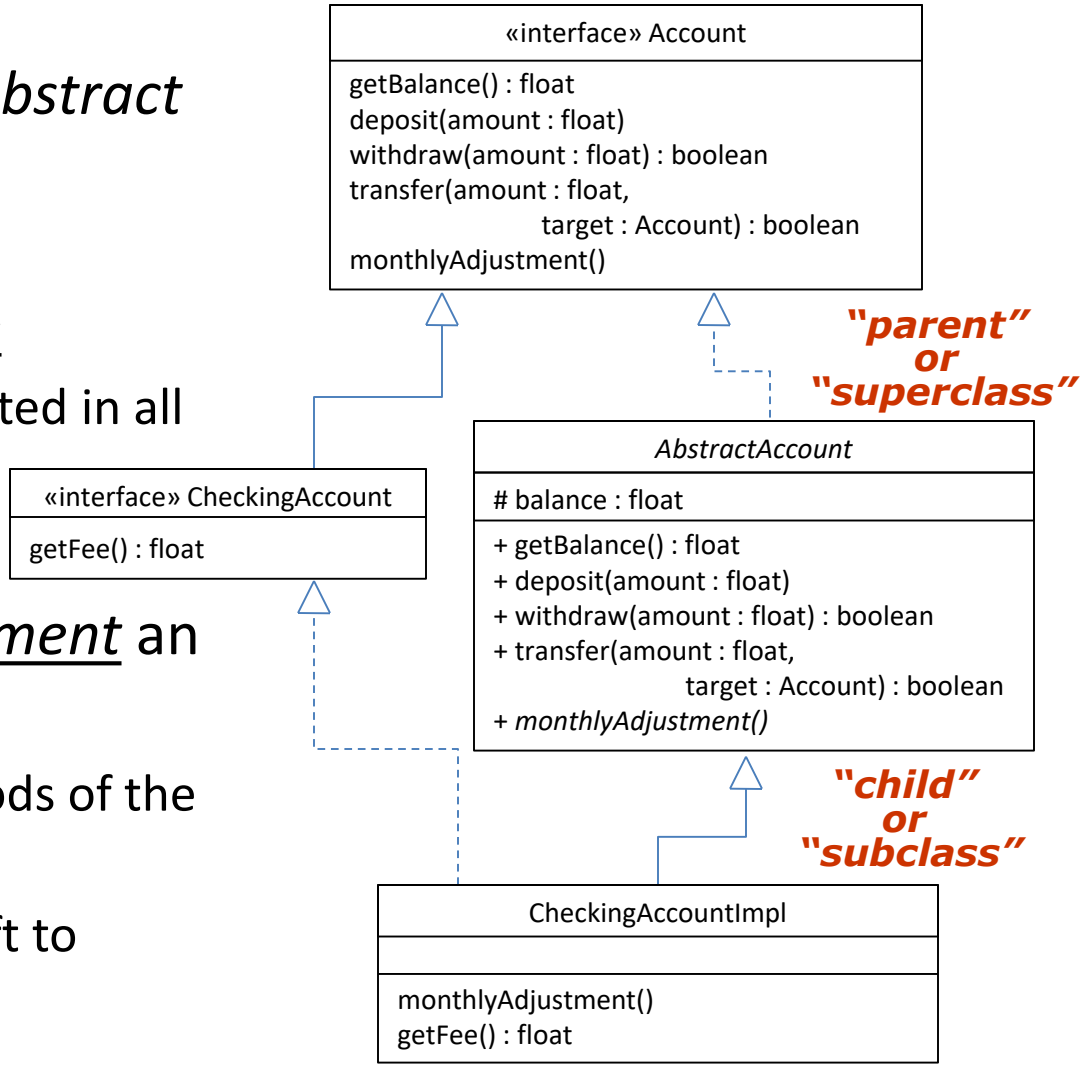
- An *interface* defines expectations / commitment for clients
  - Java: can declare methods but cannot implement them
  - Methods are *abstract methods*

- An *abstract **class*** is a convenient hybrid between an interface and a full implementation. Can have:
  - Abstract methods (no body)
  - Concrete methods (w/ body)
  - Data fields



Interfaces vs Abstract Classes vs Concrete Classes

- Unlike a concrete class, an *abstract class* ...
  - *Cannot be instantiated*
  - *Can declare abstract methods*
    - Which *must* be implemented in all *concrete* subclasses
- An abstract class may *implement* an interface
  - But need not define all methods of the interface
  - Implementation of them is left to subclasses



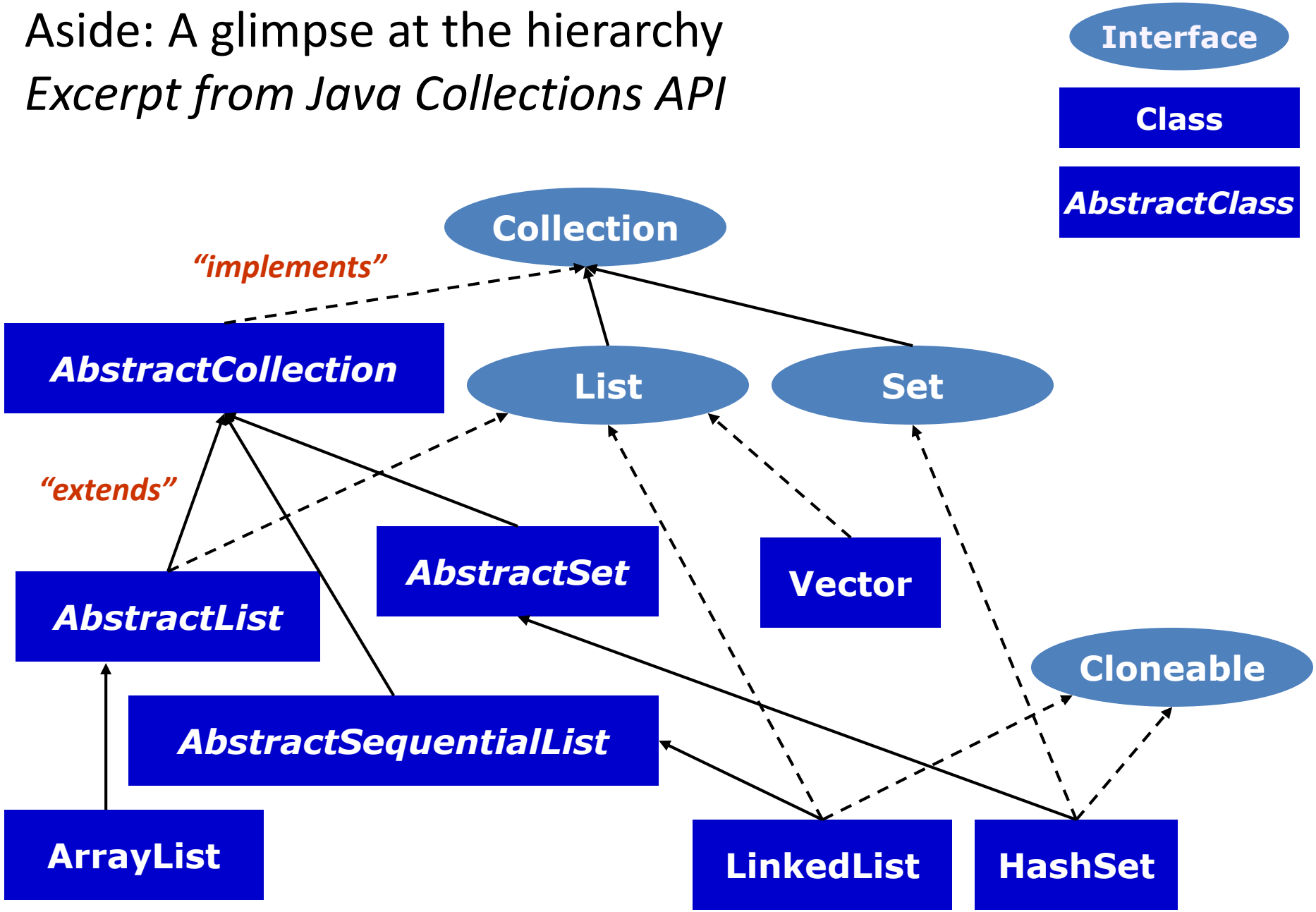
Aside: Inheritance and Class Hierarchies

- All Java classes are arranged in a hierarchy
  - **`Object`** is the *superclass* of all Java classes

- *Inheritance* and hierarchical organization capture idea:
  - One thing is a *refinement* or *extension* of another

Aside: A glimpse at the hierarchy

Excerpt from Java Collections API



Aside: Inheritance and Class Hierarchies

- All Java classes are arranged in a hierarchy
  - **`Object`** is the *superclass* of all Java classes

- *Inheritance* and hierarchical organization capture idea:
  - One thing is a *refinement* or *extension* of another

- Benefits of inheritance:
  - Fundamentally enables *reuse*
  - And *modeling flexibility*

Aside: Inheritance and subtyping

- Inheritance is for *code reuse*
  - Write code once and only once
  - Superclass features implicitly available in subclass

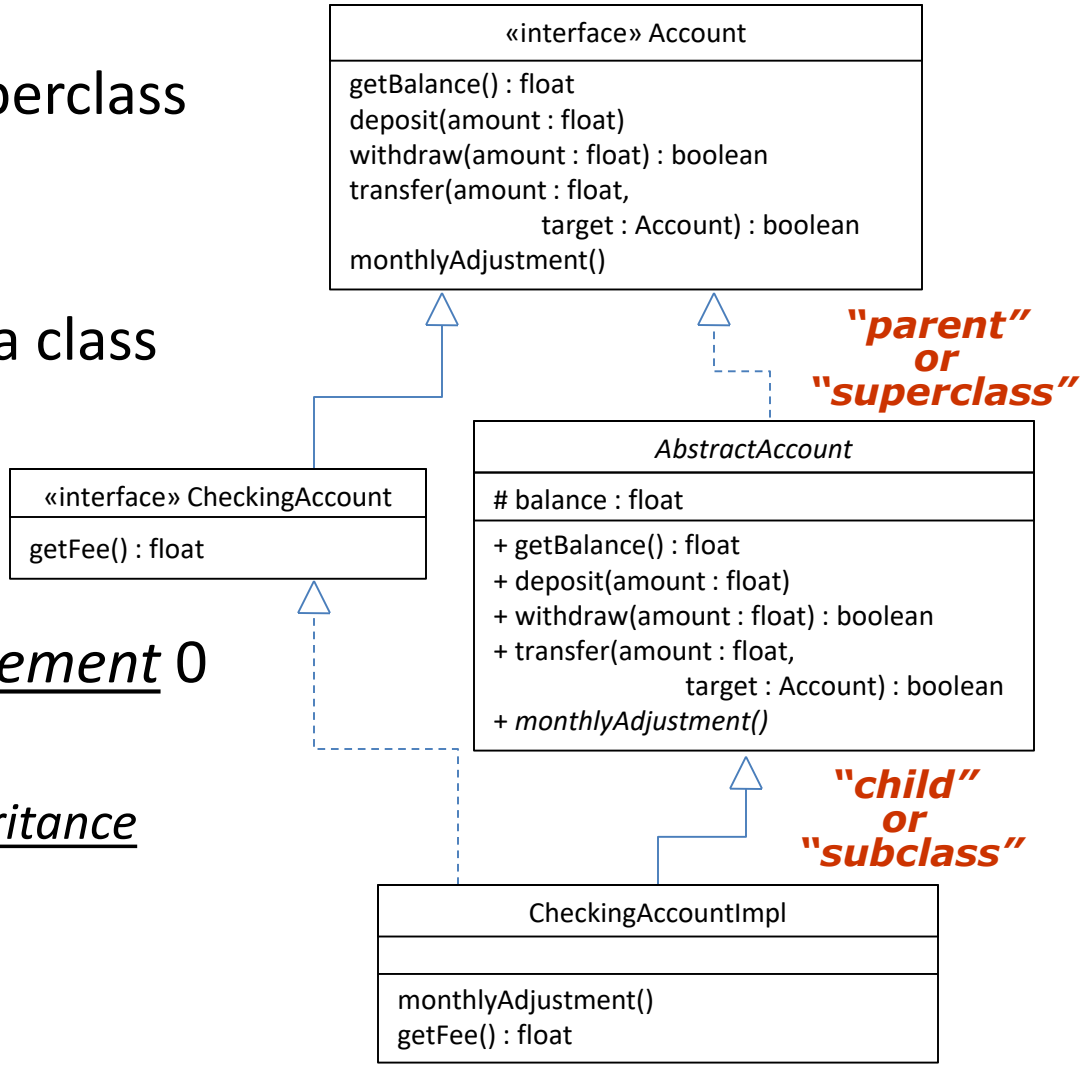
```
class A extends B
```

- Subtyping is for *polymorphism*
  - Accessing objects the same way, but getting different behavior
  - Subtype is substitutable for supertype

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

Interfaces vs Abstract Classes vs Concrete Classes

- A class can *extend* 0 or 1 superclass
  - Called *single inheritance*
- An interface cannot extend a class at all
  - (Because it is not a class)
- A class or interface can *implement* 0 or more interfaces
  - Closest thing to *multiple inheritance*



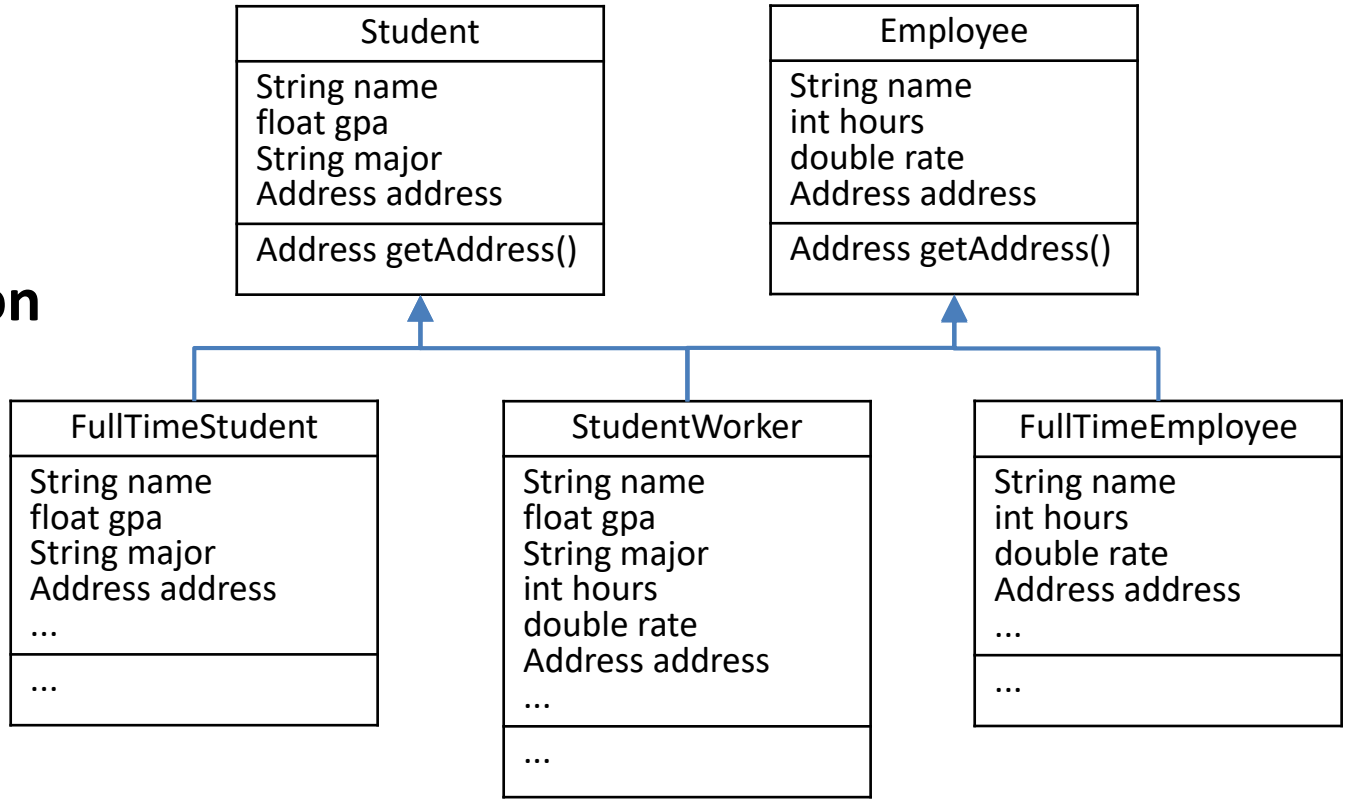
Multiple Inheritance, Multiple Interfaces, and Delegation

- *Multiple inheritance:* the ability to *extend* more than one class
- Multiple inheritance ...
  - Is difficult to implement efficiently

Multiple Inheritance, Multiple Interfaces, and Delegation

- *Multiple inheritance:* the ability to *extend* more than one class
- Multiple inheritance ...
  - Is difficult to implement efficiently

Illegal situation



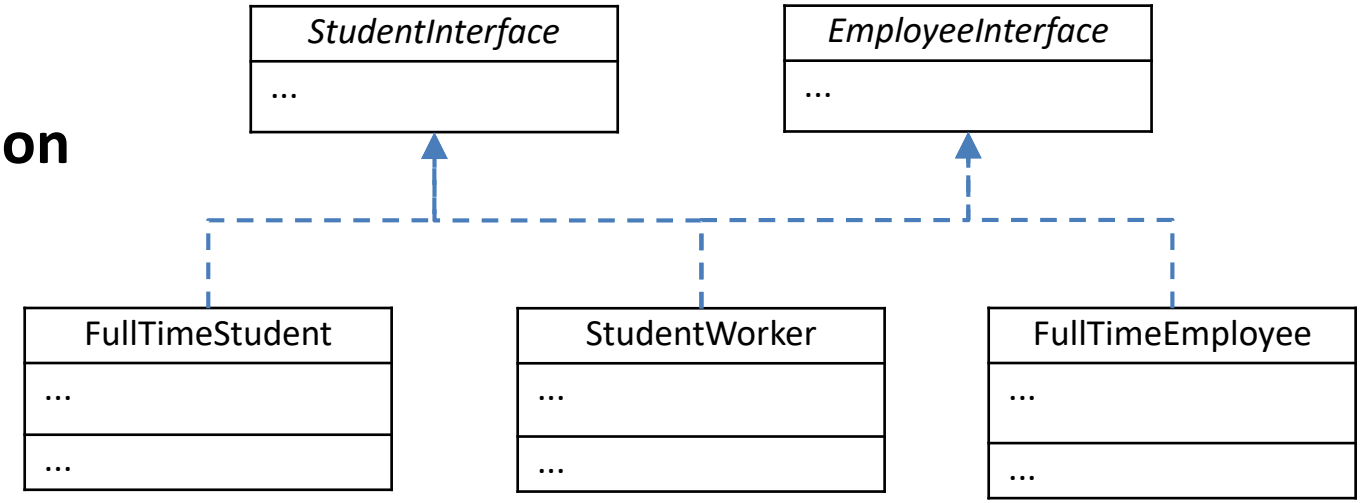
Multiple Inheritance, Multiple Interfaces, and Delegation

- *Multiple inheritance:* the ability to *extend* more than one class
- Multiple inheritance ...
  - Is difficult to implement efficiently
  - Can lead to ambiguity:
    - If two parents implement the same method, which to use?
  - Java does not allow a class to extend more than one class

Multiple Interfaces can Emulate Multiple Inheritance

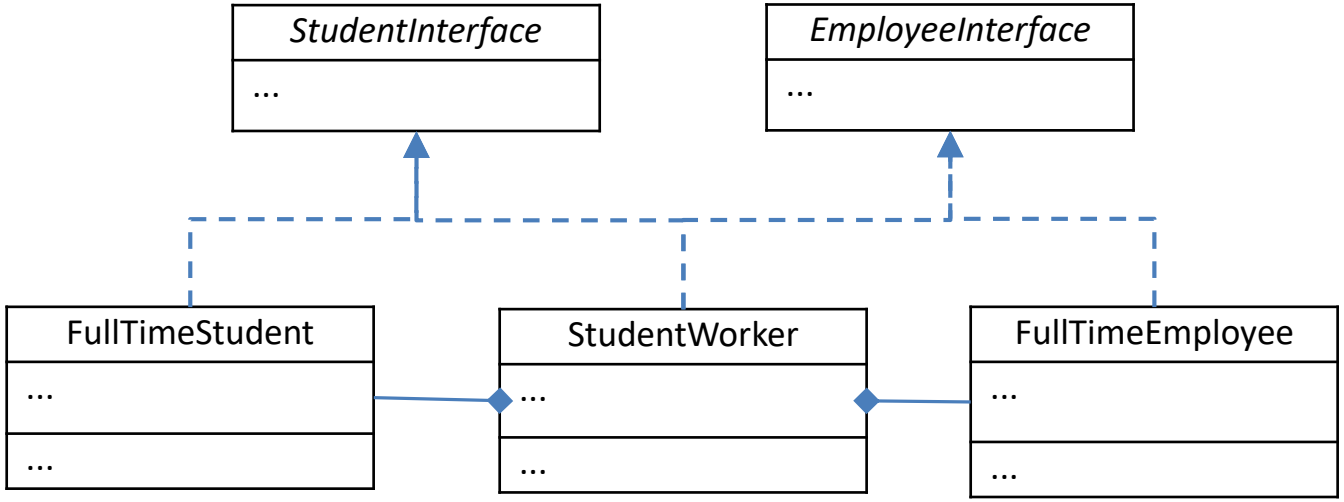
- A class can implement two or more interfaces
- Multiple interfaces emulate multiple inheritance

Legal situation

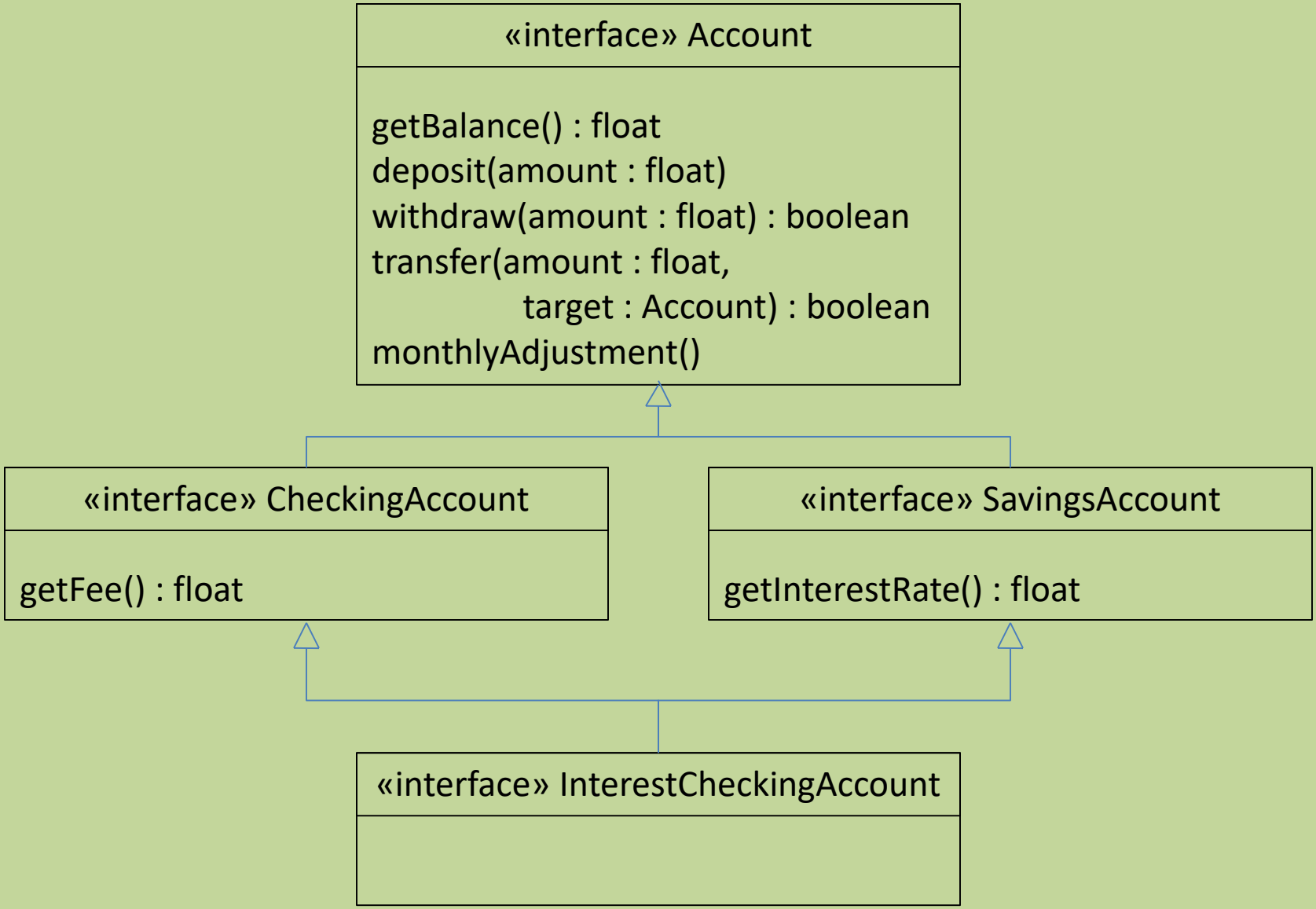


Supporting Reuse Using *Delegation*

- Reduce "cut and paste polymorphism": copied code
- *Idea:* Object of another class does the work
- *Delegation:* original object ***delegates*** to the other

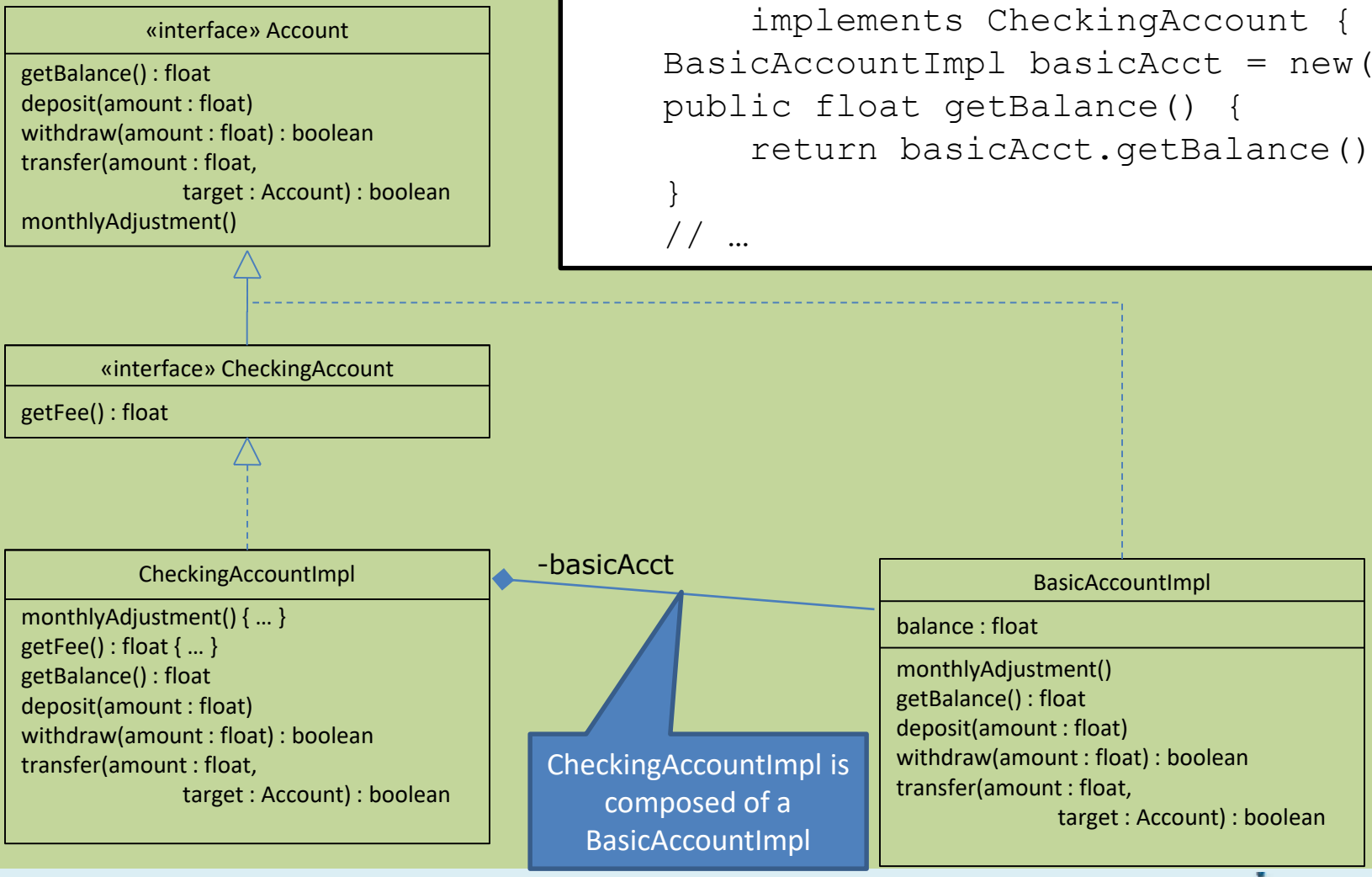


Challenge: Can we get good code reuse without inheritance?



Yes! (Reuse via composition and delegation)

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



JAVA ASIDE: SUPER, THIS, FINAL, INSTANCEOF

Java details: extended reuse with super

```
public abstract class AbstractAccount implements Account {
   protected long balance = 0;
   public boolean withdraw(long amount) {
       // withdraws money from account (code not shown)
   }
}

public class ExpensiveCheckingAccountImpl
      extends AbstractAccount implements CheckingAccount {
   public boolean withdraw(long 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 long fee;

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

   public CheckingAccountImpl(long initialBalance) {
      this(initialBalance, 500);
   }
   /* other methods… */ }
```

Invokes a constructor of the superclass. Must be the first statement of the constructor.

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.,
```
double 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;
long 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) {
    long adj = 0;
    if (acct instanceof CheckingAccount) {
        checkingAcct = (CheckingAccount) acct;
        adj = checkingAcct.getFee();
    } else if (acct instanceof SavingsAccount) {
        savingsAcct = (SavingsAccount) acct;
        adj = savingsAcct.getInterest();
    }
    …
}
```

Warning: This code is bad.

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

Note: `instanceof`

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

```
public void doSomething(Account acct) {
    long adj = 0;
    if (acct instanceof CheckingAccount) {
        checkingAcct = (CheckingAccount) acct;
        adj = checkingAcct.getFee();
    } else if (acct instanceof SavingsAccount) {
        savingsAcct = (SavingsAccount) acct;
        adj = savingsAcct.getInterest();
    } else if (acct instanceof InterestCheckingAccount) {
        icAccount = (InterestCheckingAccount) acct;
        adj = icAccount.getInterest();
        adj -= icAccount.getFee();
    }
    …
}
```

Warning: This code is bad.

Avoiding `instanceof`

```
public interface Account {
    …
    public long getMonthlyAdjustment();
}

public class CheckingAccount implements Account {
    …
    public long getMonthlyAdjustment() {
        return getFee();
    }
}

public class SavingsAccount implements Account {
    …
    public long getMonthlyAdjustment() {
        return getInterest();
    }
}
```

Avoiding `instanceof`

```
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();
   }
   …
}
```

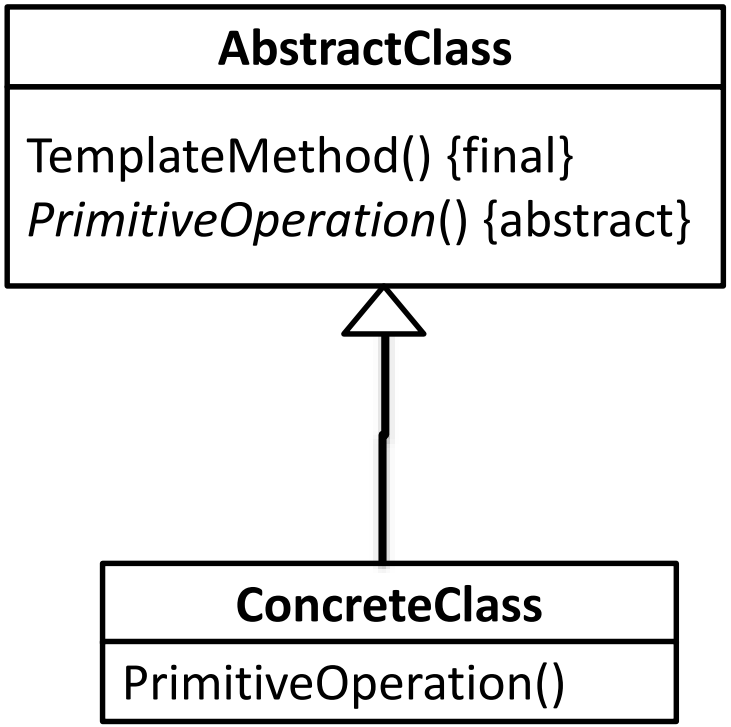
Instead:

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

THE TEMPLATE-METHOD DESIGN PATTERN

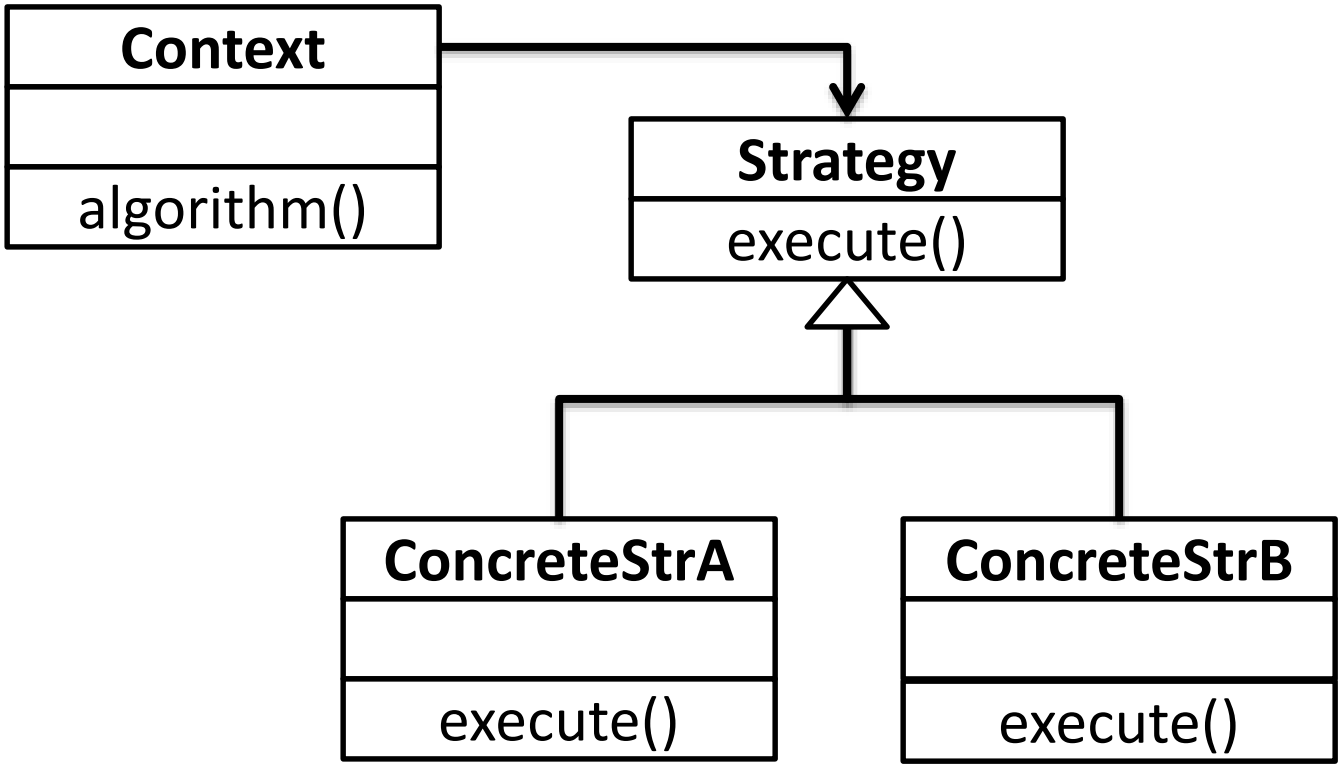
Template method design pattern

- **Problem**: An algorithm consists of customizable parts and invariant parts
- **Solution**: Implement the invariant parts of the algorithm in an abstract class, with abstract (unimplemented) primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations
- **Consequences**
  - Code reuse for the invariant parts of algorithm
  - Customization is restricted to the primitive operations
  - Inverted (Hollywood-style) control for customization: don't call us, we'll call you
    - Ensures the invariant parts of the algorithm are not changed by subclasses



The Strategy Design Pattern

Review



Template method vs. the strategy pattern

- Both support variation in a larger context
- *Template method* uses inheritance + an overridable method
- *Strategy* uses an interface and polymorphism (via composition)
  - Strategy objects are reusable across multiple classes
  - Multiple strategy objects are possible per class

ASIDE:
SPECIFICATION OF CLASS INVARIANTS

Data Structure Invariants (cf. 122)

```
struct list {
    elem data;
    struct list* next;
};
struct queue {
    list front;
    list back;
};

bool is_queue(queue Q) {
    if (Q == NULL) return false;
    if (Q->front == NULL || Q->back == NULL) return false;
    return is_segment(Q->front, Q->back);
}
```

Data Structure Invariants (cf. 122)

- Properties of the Data Structure
- Should always hold before and after method execution
- May be invalidated temporarily during method execution

```
void enq(queue Q, elem s)
//@requires is_queue(Q);
//@ensures is_queue(Q);
{ … }
```

Class Invariants

- Properties about the fields of an object
- Established by the constructor
- Should always hold before and after execution of public methods
- May be invalidated temporarily during method execution

Class Invariants

- Properties about the fields of an object
- Established by the constructor
- Should always hold before and after execution of public methods

```
public class SimpleSet {

    int contents[];
    int size;

    //@ ensures sorted(contents);
    SimpleSet(int capacity) { … }

    //@ requires sorted(contents);
    //@ ensures sorted(contents);
    boolean add(int i) { … }

    //@ requires sorted(contents);
    //@ ensures sorted(contents);
    boolean contains(int i) { … }
}
```

```
public class SimpleSet {

    int contents[];
    int size;

    //@invariant sorted(contents);

    SimpleSet(int capacity) { … }

    boolean add(int i) { … }

    boolean contains(int i) { … }
}
```

BEHAVIORAL SUBTYPING

Behavioral subtyping (Liskov Substitution Principle)

> 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:
  - Subtypes can add, but not remove methods
  - Concrete class must implement all undefined methods
  - Overriding method must return same type or subtype
  - Overriding method must accept the same parameter types
  - Overriding method may not throw additional exceptions

Behavioral subtyping (Liskov Substitution Principle)

> 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:
  - Subtypes can add, but not remove methods
  - Concrete class must implement all undefined methods
  - Overriding method must return same type or subtype
  - Overriding method must accept the same parameter types
  - Overriding method may not throw additional exceptions
- Also applies to specified behavior:
  - 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 brake 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;
        }
}
```

← **Invalidates stronger invariant (w==h) in subclass**

```
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);
    }
}
```

Maybe? (If so, it's not a square…)

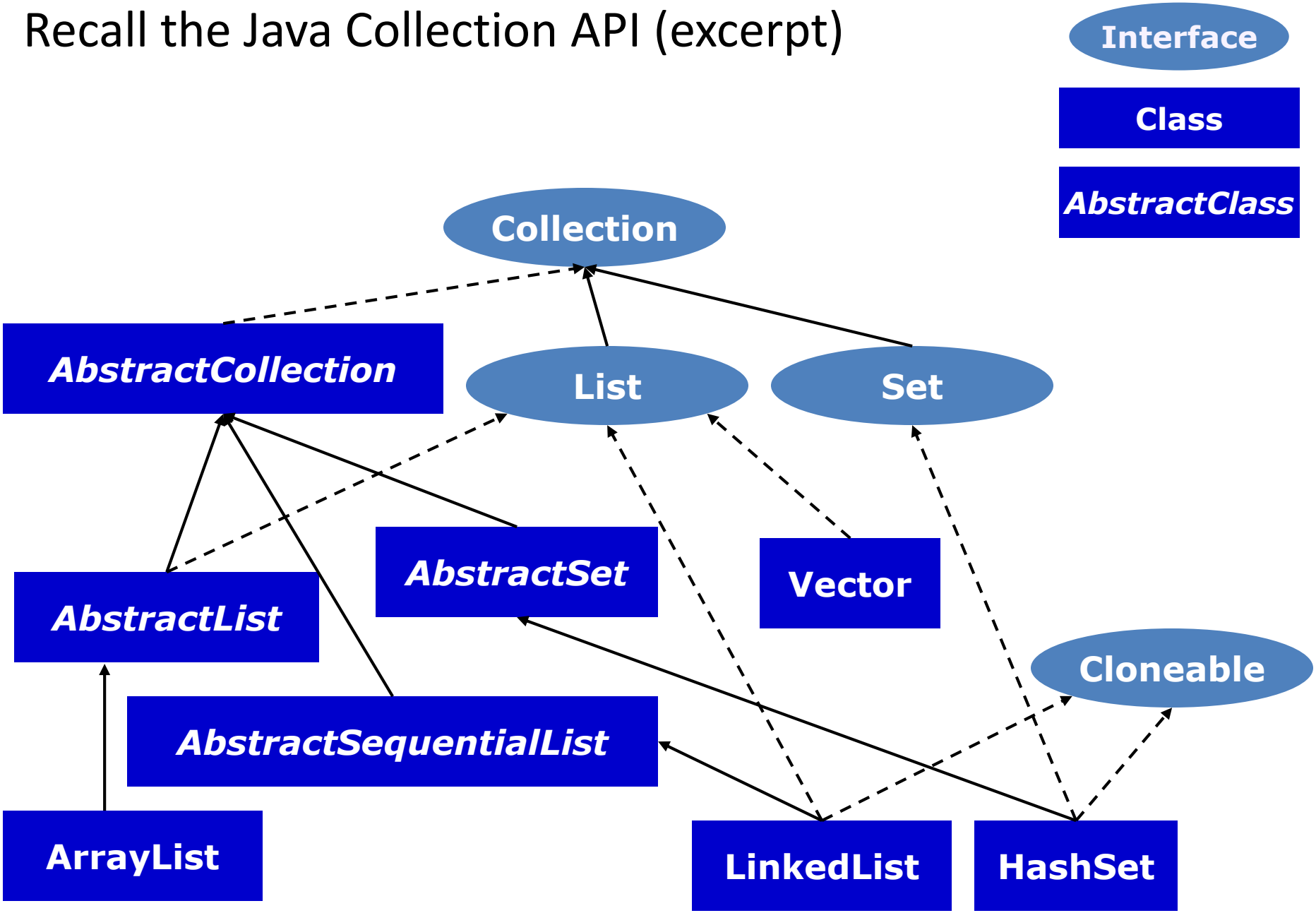
Summary: Designing reusable classes

- Favor composition over inheritance
  - Inheritance violates information hiding
- Design and document for inheritance, or prohibit it
  - Document requirements for overriding any method

- Reusable implementations with simple, clear contracts
- Inheritance for reuse, its pitfalls, and its alternatives
- Liskov's Substitution Principle for behavioral subtyping

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