Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Design for reuse

Delegation and inheritance

Michael Hilton **Bogdan Vasilescu**

School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 1 graded soon
- Reading assignment due today: Effective Java Items 17 + 50
  - Optional reading due Thursday
  - Required reading due next Tuesday
- Homework 2 due Thursday 11:59 p.m.

Take out a piece of paper…

In 3 minutes:

1. What is your Andrew ID?
2. What is the main point of one of today's reading assignments (e.g., Effective Java 17)?
3. What is the main point of the other of today's reading assignments (e.g., Effective Java 50)?

1. What is your Andrew ID?

+2 any answer

2. and 3. What are the main points of the reading?

+2 Minimize mutability

+2 Make defensive copies

Write "Graded by" and your Andrew ID on the bottom.

Write their score (x/6) near the top of their paper.

Design goals for your Homework 1 solution?

Functional correctness	Adherence of implementation to the specifications
Robustness	Ability to handle anomalous events
Flexibility	Ability to accommodate changes in specifications
Reusability	Ability to be reused in another application
Efficiency	Satisfaction of speed and storage requirements
Scalability	Ability to serve as the basis of a larger version of the application
Security	Level of consideration of application security

Source: Braude, Bernstein, Software Engineering. Wiley 2011

One Homework 1 solution…

```
class Document {
    private final String url;
    public Document(String url) {
        this.url = url;
    }

    public double similarityTo(Document d) {
        … ourText = download(url);
        … theirText = download(d.url);
        … ourFreq = computeFrequencies(ourText);
        … theirFreq = computeFrequencies(theirText);
        return cosine(ourFreq, theirFreq);
    }
    …
}
```

Compare to another Homework 1 solution…

```
class Document {
    private final String url;
    public Document(String url) {
        this.url = url;
    }

    public double s
        … ourText =
        … theirText
        … ourFreq =
        … theirFreq
        return cosi
    }
    …
}
```

```
class Document {
    private final … frequencies;
    public Document(String url) {
        … ourText = download(url);
        frequencies = computeFrequencies(ourText);
    }

    public double similarityTo(Document d) {
        return cosine(frequencies,
                      d.frequencies);
    }
    …
}
```

Using the Document class

```
For each url:
    Construct a new Document

For each pair of Documents d1, d2:
    Compute d1.similarityTo(d2)
    …
```

- What is the running time of this, for n urls?

Latency Numbers Every Programmer Should Know

Jeff Dean, Senior Fellow, Google

PRIMITIVE	LATENCY: ns	us	ms
L1 cache reference	0.5		
Branch mispredict	5		
L2 cache reference	7		
Mutex lock/unlock	25		
Main memory reference	100		
Compress 1K bytes with Zippy	3,000	3	
Send 1K bytes over 1 Gbps network	10,000	10	
Read 4K randomly from SSD*	150,000	150	
Read 1 MB sequentially from memory	250,000	250	
Round trip within same datacenter	500,000	500	
Read 1 MB sequentially from SSD*	1,000,000	1,000	1
Disk seek	10,000,000	10,000	10
Read 1 MB sequentially from disk	20,000,000	20,000	20
Send packet CA->Netherlands->CA	150,000,000	150,000	150

The point

- Constants matter
- Design goals sometimes clearly suggest one alternative

Key concepts from last Thursday

Key concepts from last Thursday

- Testing
- Specifying program behavior: contracts
- Behavioral subtyping
- The `java.lang.Object` contracts

Reminder: Subtype Polymorphism

- A type (e.g. Point) can have many forms (e.g., CartesianPoint, PolarPoint, …)
- Use interfaces to separate expectations from implementation

Creating Objects

```
interface Point {
    int getX();
    int getY();
}
Point p = new Point() {
    int getX() { return 3; }
    int getY() { return -10; }
}
```

Classes as Object Templates

```
interface Point {
    int getX();
    int getY();
}
class CartesianPoint implements Point {
    int x,y;
    Point(int x, int y) {this.x=x; this.y=y;}
    int getX() { return this.x; }
    int getY() { return this.y; }
}
Point p = new CartesianPoint(3, -10);
```

Polar Points

```
interface Point {
    int getX();
    int getY();
}
class PolarPoint implements Point {
    double len, angle;
    PolarPoint(double len, double angle)
        {this.len=len; this.angle=angle;}
    int getX() { return this.len * cos(this.angle);}
    int getY() { return this.len * sin(this.angle); }
  double getAngle() {…}
}
Point p = new PolarPoint(5, .245);
```

Reminder: Subtype Polymorphism (cont'd)

- When invoking a method p.x() the specific implementation of x() from object p is executed
  - The executed method depends on the actual object p, i.e., on the runtime type
  - It does not depend on the static type, i.e., how p is declared

- All implementations of an interface can be used interchangeably

- This allows flexible **change** (modifications, extensions, reuse) later without changing the client implementation, even in unanticipated contexts

Static types vs dynamic types

- Static type: how is a variable declared
- Dynamic type: what type has the object in memory when executing the program (we may not know until we execute the program)

```
Point createZeroPoint() {
       if (new Math.Random().nextBoolean())
              return new CartesianPoint(0, 0);
       else   return new PolarPoint(0,0);
}
Point p = createZeroPoint();
p.getX();
p.getAngle();
```

Behavioral subtyping summary

- When subtyping, design and implement carefully
  - Subtype must be substitutable anywhere the supertype could be used

- 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

Contracts and Interfaces

- All objects implementing an interface must adhere to the interface's contracts
  - Objects may provide different implementations for the same specification
  - Subtype polymorphism: Client only cares about interface, not about the implementation

p.getX() **s.read()**

=> **Design for Change**

Methods common to all `Objects`

- `equals`: returns true if the two objects are “equal”
- `hashCode`: returns an `int` that must be equal for equal objects, and is likely to differ for unequal objects
- `toString`: returns a printable string representation

What does the following code print?

```
public final class Name {
      private final String first, last;

      public Name(String first, String last) {
          if (first == null || last == null)
              throw new NullPointerException();
          this.first = first; this.last = last;

      public boolean equals(Name o) {
          return first.equals(o.first) && last.equals(o.last);
      }

      public int hashCode() {
          return 31 * first.hashCode() + last.hashCode();
      }

      public static void main(String[] args) {
          Set<Name> s = new HashSet<>();
          s.add(new Name("Mickey", "Mouse"));
          System.out.println(
              s.contains(new Name("Mickey", "Mouse")));
      }
}
```

(a) true
(b) false
(c) It varies
(d) None of the above

What does it print?

(a) `true`

(b) `false`

(c) It varies

(d) None of the above

The `Name` class overrides `hashCode` but not `equals`!

The two `Name` instances are thus unequal.

What does the following code print?

```
public final class Name {
      private final String first, last;

      public Name(String first, String last) {
          if (first == null || last == null)
              throw new NullPointerException();
          this.first = first; this.last = last;

      public boolean equals(Name o) {  // Accidental overloading
          return first.equals(o.first) && last.equals(o.last);
      }

      public int hashCode() {
          return 31 * first.hashCode() + last.hashCode();
      }

      public static void main(String[] args) {
          Set<Name> s = new HashSet<>();
          s.add(new Name("Mickey", "Mouse"));
          System.out.println(
              s.contains(new Name("Mickey", "Mouse")));
      }
}
```

A correct `equals` implementation

```
@Override
public boolean equals(Object o) {
    if (!(o instanceof Name))
        return false;
    Name n = (Name) o;
    return n.first.equals(first) && n.last.equals(last);
}
```

Today

- Design for reuse: delegation and inheritance

Recall our earlier sorting example:

Version A:

```
static void sort(int[] list, boolean ascending) {
   …
   boolean mustSwap;
   if (ascending) {
      mustSwap = list[i] < list[j];
   } else {
      mustSwap = list[i] > list[j];
   }
   …
}
```

Version B':

```
interface Comparator {
  boolean compare(int i, int j);
}
final Comparator ASCENDING =  (i, j) -> i < j;
final Comparator DESCENDING = (i, j) -> i > j;

static void sort(int[] list, Comparator cmp) {
  …
  boolean mustSwap =
    cmp.compare(list[i], list[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

```
interface Comparator {
  boolean compare(int i, int j);
}
final Comparator ASCENDING =  (i, j) -> i < j;
final Comparator DESCENDING = (i, j) -> i > j;

static void sort(int[] list, Comparator cmp) {
  …
  boolean mustSwap =
    cmp.compare(list[i], list[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
  - `Comparator`s can be reused with arbitrary client code that needs to compare integers

```
interface Comparator {
  boolean compare(int i, int j);
}
final Comparator ASCENDING =  (i, j) -> i < j;
final Comparator DESCENDING = (i, j) -> i > j;

static void sort(int[] list, Comparator cmp) {
  …
  boolean mustSwap =
    cmp.compare(list[i], list[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

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

- One solution:

```
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);
  }
  …
```

Delegation and design

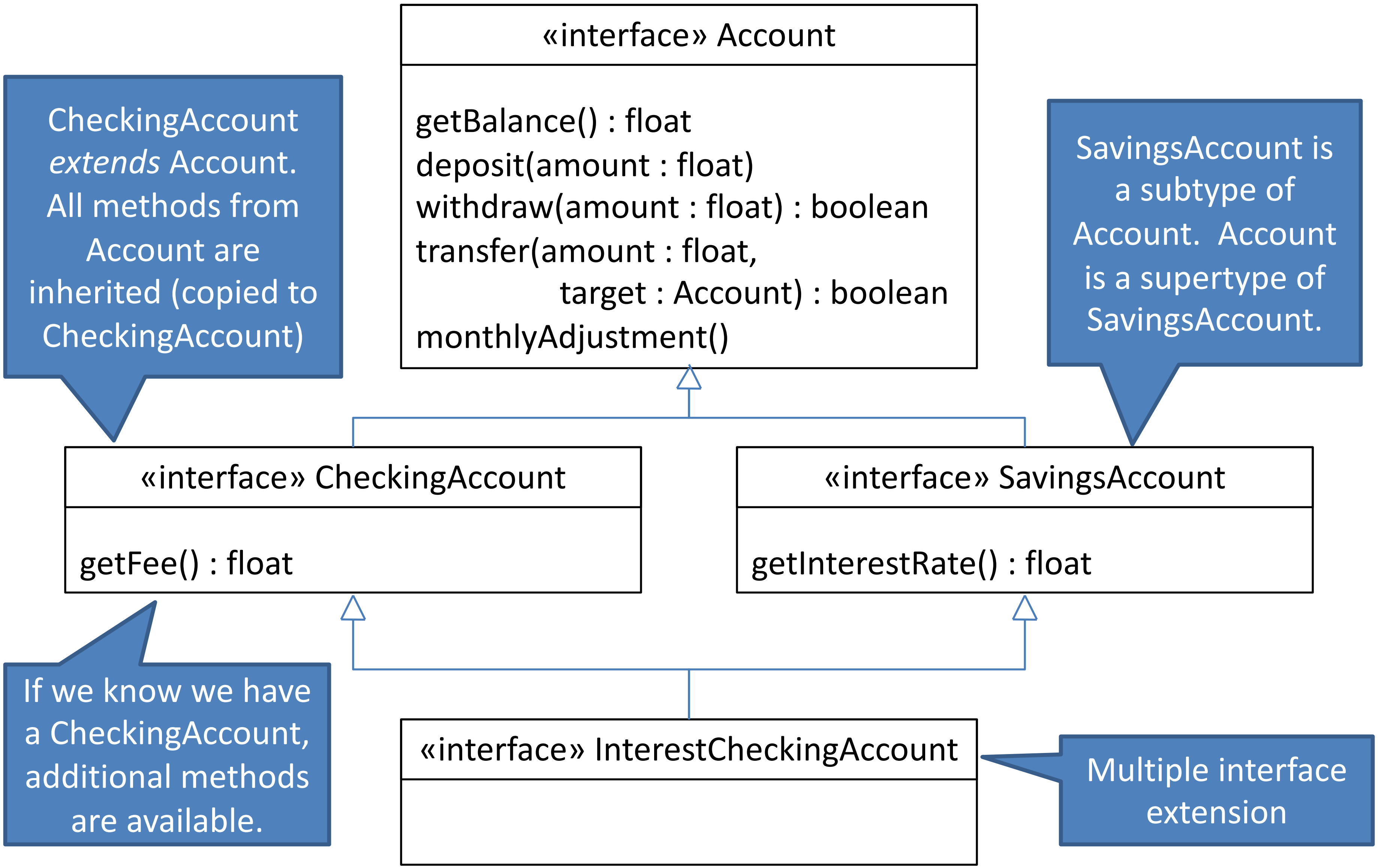
- Small interfaces with clear contracts
- Classes to encapsulate algorithms, behaviors
  - E.g., the `Comparator`

Recall: bank accounts example

«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

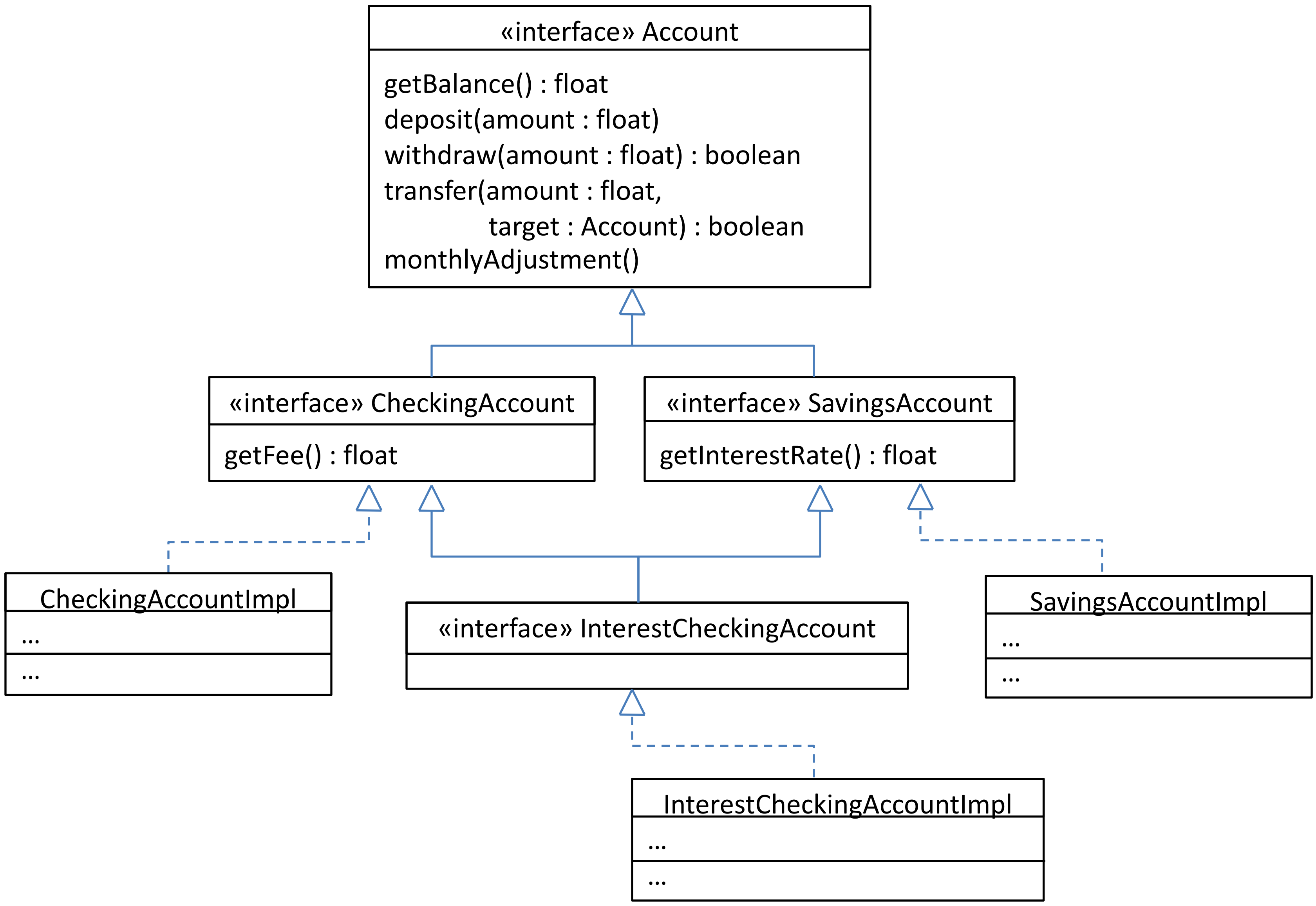
Interface inheritance for an account type hierarchy



Interface inheritance for an account type hierarchy

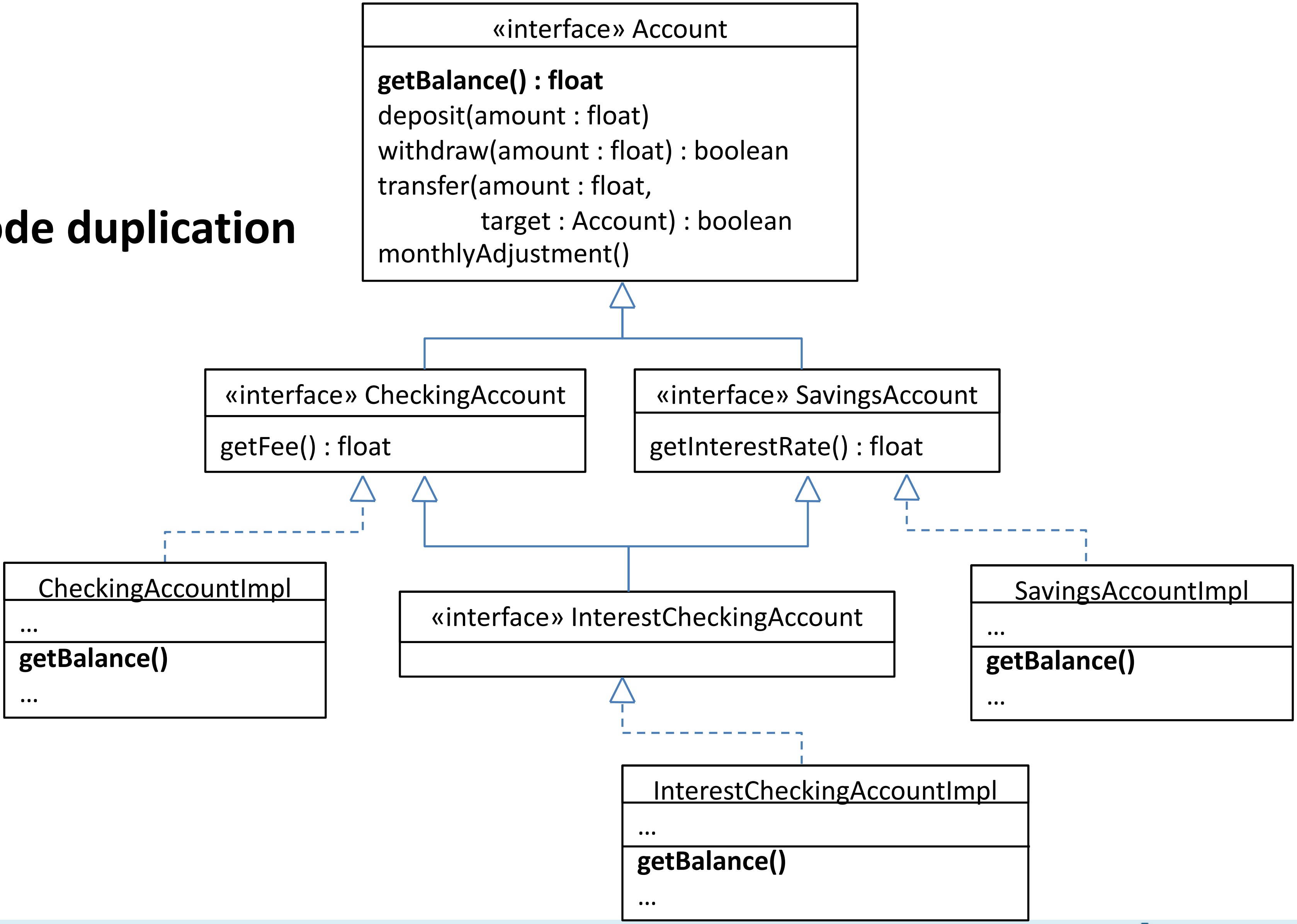
```
public interface Account {
    public long getBalance();
    public void  deposit(long amount);
    public boolean withdraw(long amount);
    public boolean transfer(long amount, Account target);
    public void monthlyAdjustment();
}

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

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

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

Recall: Implementation inheritance for code reuse



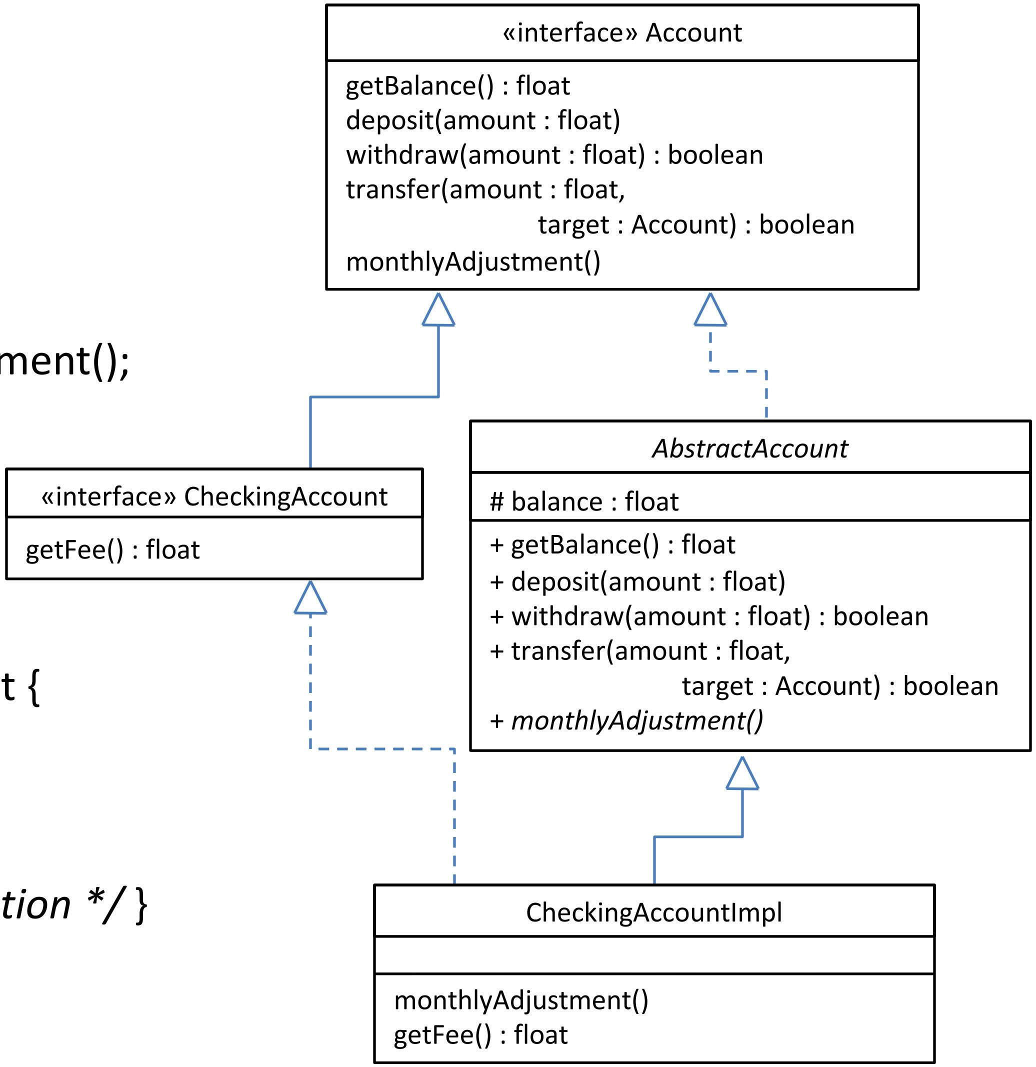
Implementation inheritance for code reuse

Code duplication

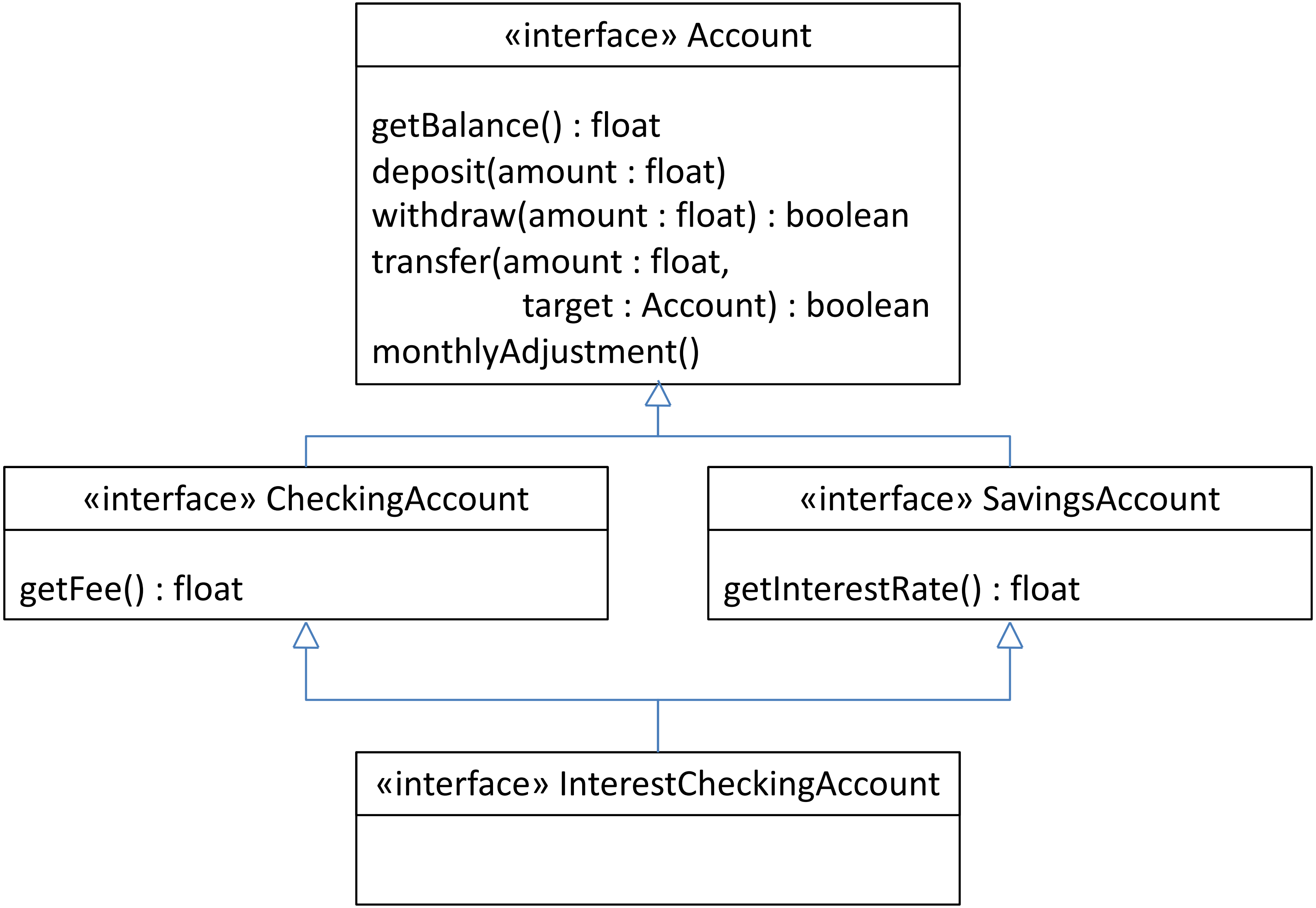


Better: Reuse abstract account code

```
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 */ }
}
```

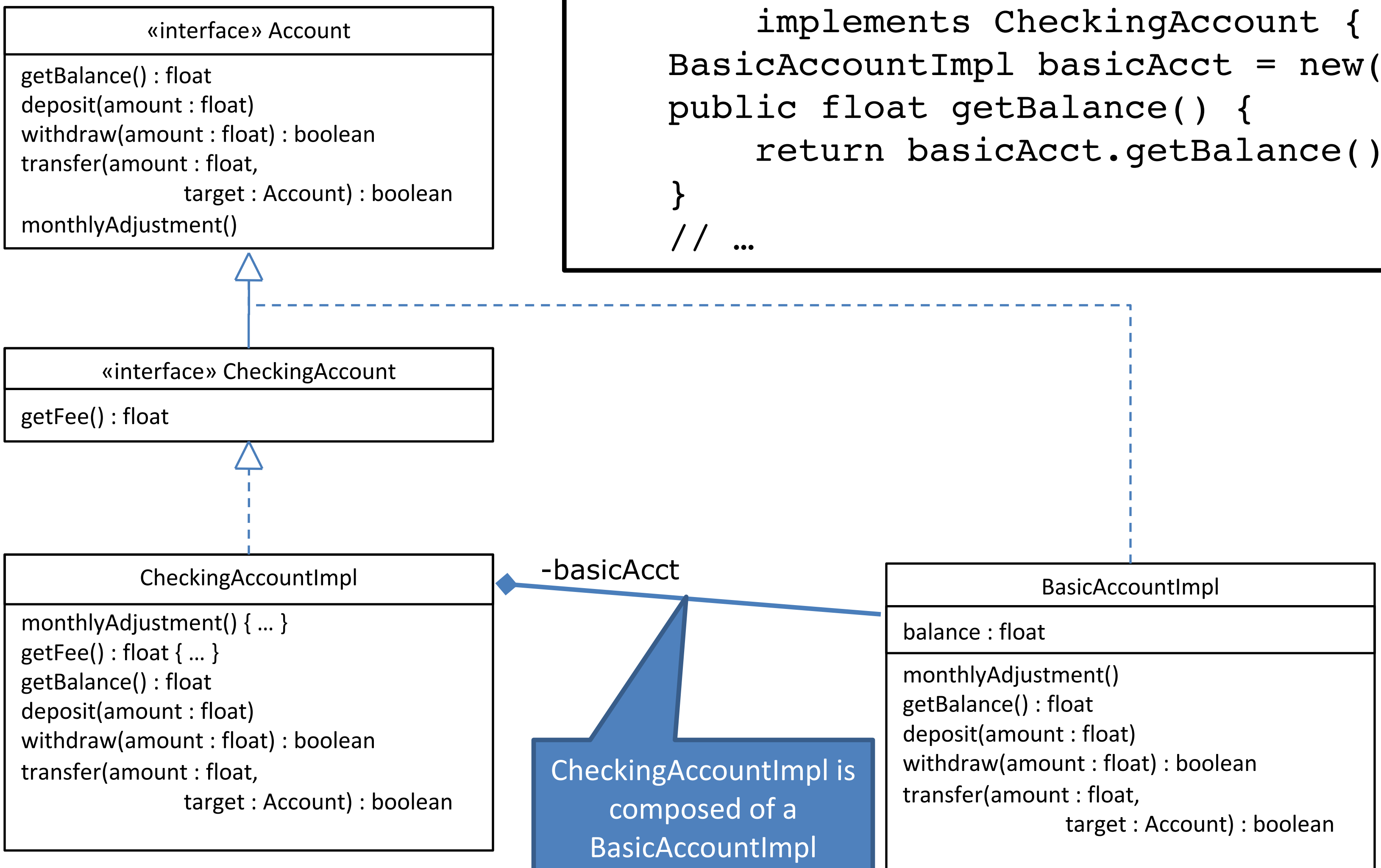


Challenge: Can we get good code reuse without inheritance?



Yes! (Reuse via composition and delegation)

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



Design alternatives: delegation vs. inheritance

```
class BasicAccount
        implements Account {
   private long balance = 0;
   public long getBalance() {
        return balance;
   }
   // other methods…
}

public class CheckingAccountImpl
        implements CheckingAccount {
   private BasicAccount account;
   public long getBalance() {
        return account.getBalance();
   }
   public void monthlyAdjustment() {
        account.setBalance(
            account.getBalance() - getFee());
   }
   public long getFee() { … }
}
```

Design discussion: Delegation vs. inheritance

- Inheritance can improve modeling flexibility
  - `protected` hooks / helper methods
  - Test with subclasses
- But usually, favor composition/delegation over inheritance
  - Inheritance violates information hiding
  - Delegation supports information hiding
- Design and document for inheritance, or prohibit it
  - Document requirements for overriding any method
- **See (excellent) optional reading for Thursday**