

Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Designing classes

Behavioral subtyping

Michael Hilton

Bogdan Vasilescu

Administrivia

- Homework 1 due tonight 11:59 p.m.
 - Everyone must read and sign our collaboration policy
- Reading due Tuesday: Effective Java, Items 17 and 50
- Homework 2 due next Thursday at 11:59 p.m.

Key concepts from Tuesday

Key concepts from Tuesday

- Information hiding: Design for change, design for reuse
 - Encapsulation: Visibility modifiers in Java
 - Interface types vs. class types
- Functional correctness
 - JUnit and friends

Today

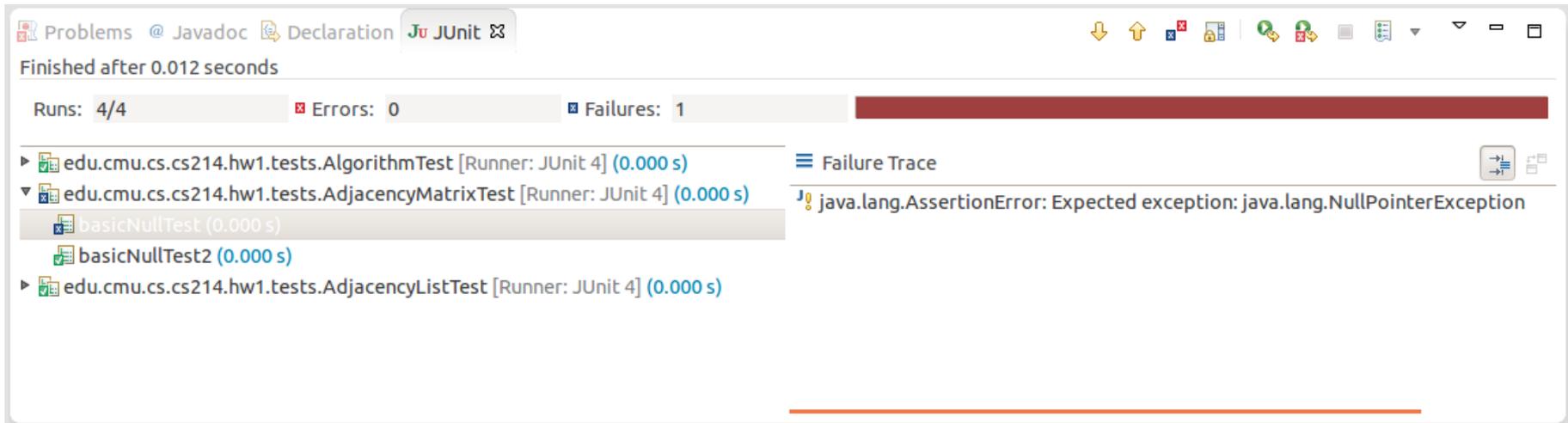
- Functional correctness, continued
- Behavioral subtyping
 - Liskov Substitution Principle
 - The `java.lang.Object` contracts

Unit testing

- Tests for small units: methods, classes, subsystems
 - Smallest testable part of a system
 - Test parts before assembling them
 - Intended to catch local bugs
- Typically written by developers
- Many small, fast-running, independent tests
- Few dependencies on other system parts or environment

JUnit

- A popular, easy-to-use, unit-testing framework for Java



The screenshot shows an IDE's JUnit runner interface. At the top, it says "Finished after 0.012 seconds". Below that, a progress bar shows "Runs: 4/4", "Errors: 0", and "Failures: 1". The test results list includes:

- edu.cmu.cs.cs214.hw1.tests.AlgorithmTest [Runner: JUnit 4] (0.000 s)
- edu.cmu.cs.cs214.hw1.tests.AdjacencyMatrixTest [Runner: JUnit 4] (0.000 s)
 - basicNullTest (0.000 s) - Failed
 - basicNullTest2 (0.000 s) - Passed
- edu.cmu.cs.cs214.hw1.tests.AdjacencyListTest [Runner: JUnit 4] (0.000 s)

The failure trace for the failed test is:

```
java.lang.AssertionError: Expected exception: java.lang.NullPointerException
```

A JUnit example

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class AdjacencyListTest {
    @Test
    public void testSanityTest(){
        Graph g1 = new AdjacencyListGraph(10);
        Vertex s1 = new Vertex("A");
        Vertex s2 = new Vertex("B");
        assertEquals(true, g1.addVertex(s1));
        assertEquals(true, g1.addVertex(s2));
        assertEquals(true, g1.addEdge(s1, s2));
        assertEquals(s2, g1.getNeighbors(s1)[0]);
    }

    @Test
    public void test...

    private int helperMethod...
}
```

Selecting test cases

- Write tests based on the specification, for:
 - Representative cases
 - Invalid cases
 - Boundary conditions
- Write stress tests
 - Automatically generate huge numbers of test cases
- Think like an attacker
- Other tests: performance, security, system interactions, ...

A testing example

```
/**  
 * computes the sum of the first len values of the array  
 *  
 * @param array array of integers of at least length len  
 * @param len number of elements to sum up  
 * @return sum of the first len array values  
 * @throws NullPointerException if array is null  
 * @throws ArrayIndexOutOfBoundsException if len > array.Length  
 * @throws IllegalArgumentException if len < 0  
 */  
int partialSum(int array[], int len);
```

A testing example

```
/**  
 * computes the sum of the first len values of the array  
 *  
 * @param array array of integers of at least length len  
 * @param len number of elements to sum up  
 * @return sum of the first len array values  
 * @throws NullPointerException if array is null  
 * @throws ArrayIndexOutOfBoundsException if len > array.Length  
 * @throws IllegalArgumentException if len < 0  
 */  
int partialSum(int array[], int len);
```

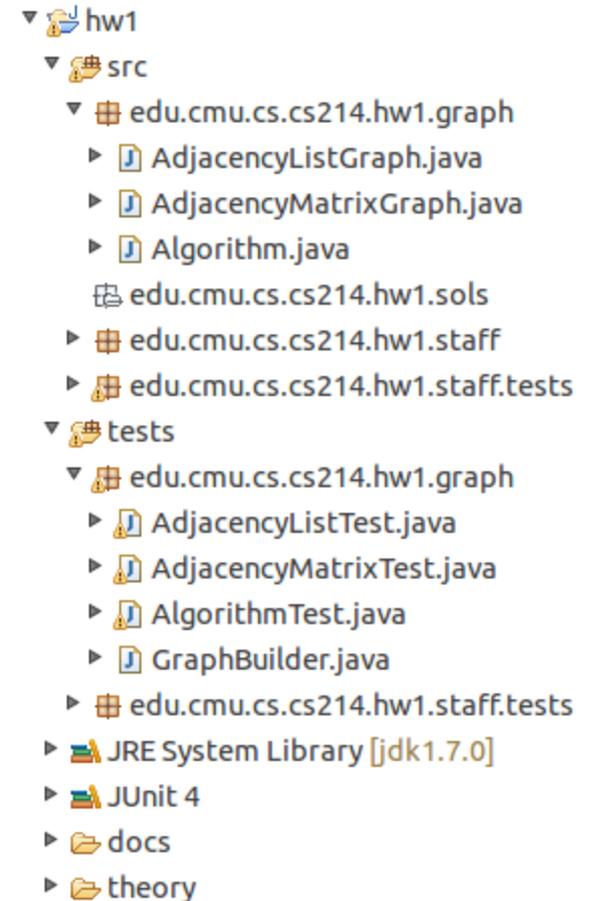
- Test null array
- Test length > array.length
- Test negative length
- Test small arrays of length 0, 1, 2
- Test long array
- Test length == array.length
- Stress test with randomly-generated arrays and lengths

A testing exercise

```
/**  
 * Copies the specified array, truncating or padding with zeros  
 * so the copy has the specified length. For all indices that are  
 * valid in both the original array and the copy, the two arrays will  
 * contain identical values. For any indices that are valid in the  
 * copy but not the original, the copy will contain 0.  
 * Such indices will exist if and only if the specified length  
 * is greater than that of the original array.  
 *  
 * @param original the array to be copied  
 * @param newLength the length of the copy to be returned  
 * @return a copy of the original array, truncated or padded with  
 *         zeros to obtain the specified length  
 * @throws NegativeArraySizeException if newLength is negative  
 * @throws NullPointerException if original is null  
 */  
int [] copyOf(int[] original, int newLength);
```

Test organization conventions

- Have a test class `FooTest` for each public class `Foo`
- Separate source and test directories
 - `FooTest` and `Foo` in the same package



Testable code

- Think about testing when writing code
 - Modularity and testability go hand in hand
- Same test can be used on all implementations of an interface!
- Test-driven development
 - Writing tests before you write the code
 - Tests can expose API weaknesses

Writing testable code

```
//700LOC
public boolean foo() {
    try {
        synchronized () {
            if () {
            } else {
            }
            for () {
                if () {
                    if () {
                        if () {
                            if ()?
                            {
                                if () {
                                    for () {
                                    }
                                }
                            }
                        } else {
                            if () {
                                for () {
                                    if () {
                                        } else {
                                        }
                                    } else {
                                        if () {
                                        }
                                    }
                                }
                            }
                            if () {
                                if () {
                                    if () {
                                        for () {
                                        }
                                    }
                                }
                            }
                        } else {
                        }
                    }
                } else {
                }
            }
        }
    }
}
```

Unit testing as a design mechanism:

- Code with low complexity
- Clear interfaces and specifications

Source:
<http://thedailywtf.com/Articles/Coding-Like-the-Tour-de-France.aspx>

Run tests frequently

- Run tests before every commit
 - Do not commit code that fails a test
- If entire test suite becomes too large and slow:
 - Run local package-level tests ("smoke tests") frequently
 - Run all tests nightly
 - Medium sized projects easily have 1000s of test cases
- Continuous integration servers scale testing

Continuous integration: Travis CI

The screenshot shows a web browser window displaying a Travis CI build page for the repository `wyvernlang / wyvern`. The build status is `passing`. The page includes a search bar, navigation links (Blog, Status, Help), and a user profile for Jonathan Aldrich. The main content area shows the build details for `SimpleWyvern-devel`, including the commit hash `fd7be1c`, the duration of 16 seconds, and the fact that it was authored and committed by `potanin`. A yellow banner indicates that the job ran on legacy infrastructure. Below this is a terminal log showing the build process, including system information, git checkout, java version, and ant test steps.

Build #17 - wyvernlang

https://travis-ci.org/wyvernlang/wyvern/builds/79099642

Travis CI Blog Status Help Jonathan Aldrich

Search all repositories

My Repositories +

- wyvernlang/wyvern # 17
- Duration: 16 sec
- Finished: 3 days ago

wyvernlang / wyvern build passing

Current Branches Build History Pull Requests > Build #17 Settings

SimpleWyvern-devel Asserting false (works on Linux, so its OK). # 17 passed

- Commit fd7be1c
- Compare 0e2af1f..fd7b
- ran for 16 sec
- 3 days ago

potanin authored and committed

This job ran on our legacy infrastructure. Please read [our docs on how to upgrade](#)

Remove Log Download Log

```
1 Using worker: worker-linux-027f0490-1.bb.travis-ci.org:travis-linux-2
2
3 Build system information system_info
67
68 $ git clone --depth=50 --branch=SimpleWyvern-devel git.checkout 0.815
69 $ jdk_switcher use oraclejdk8
70 Switching to Oracle JDK8 (java-8-oracle), JAVA_HOME will be set to /usr/lib/jvm/java-8-oracle
71 $ java -Xmx32m -version
72 java version "1.8.0_31"
73 Java(TM) SE Runtime Environment (build 1.8.0_31-b13)
74 Java HotSpot(TM) 64-Bit Server VM (build 25.31-b07, mixed mode)
75 $ javac -J-Xmx32m -version
76 javac 1.8.0_31
77 $ cd tools 0.815
78
79 The command "cd tools" exited with 0.
80 $ ant test 11.815
81 Buildfile: /home/travis/build/wyvernlang/wyvern/tools/build.xml
82
83 copper-compose-compile:
```

Continuous integration: Travis CI build history

The screenshot shows the Travis CI interface for the repository `wyvernlang / wyvern`. The page displays a list of build history entries, each with a status icon (green checkmark for passed, red X for failed), a commit message, the number of tests passed/failed, the duration, and the time since the build finished. The current build is passing.

Status	Commit Message	Tests	Duration	Time Ago
✓	SimpleWyvern-devel Asserting false (works on L... potanin committed	# 17 passed	16 sec	3 days ago
✓	SimpleWyvern-devel Debugging mac bug. potanin committed	# 16 passed	22 sec	3 days ago
✓	SimpleWyvern-devel Zooming in on Mac's IRBui... potanin committed	# 14 passed	15 sec	4 days ago
✓	SimpleWyvern-devel Zooming in on Mac LLVM b... potanin committed	# 13 passed	16 sec	4 days ago
✓	SimpleWyvern-devel Removed outdated tests Jonathan Aldrich committed	# 7 passed	15 sec	11 days ago
✓	newlexer Merge branch 'master' of https://githu... Jonathan Aldrich committed	# 6 passed	14 sec	11 days ago
✓	master Build with JDK 8 Jonathan Aldrich committed	# 5 passed	13 sec	11 days ago
✗	master fixed Travis build script syntax error Jonathan Aldrich committed	# 4 failed	5 sec	11 days ago

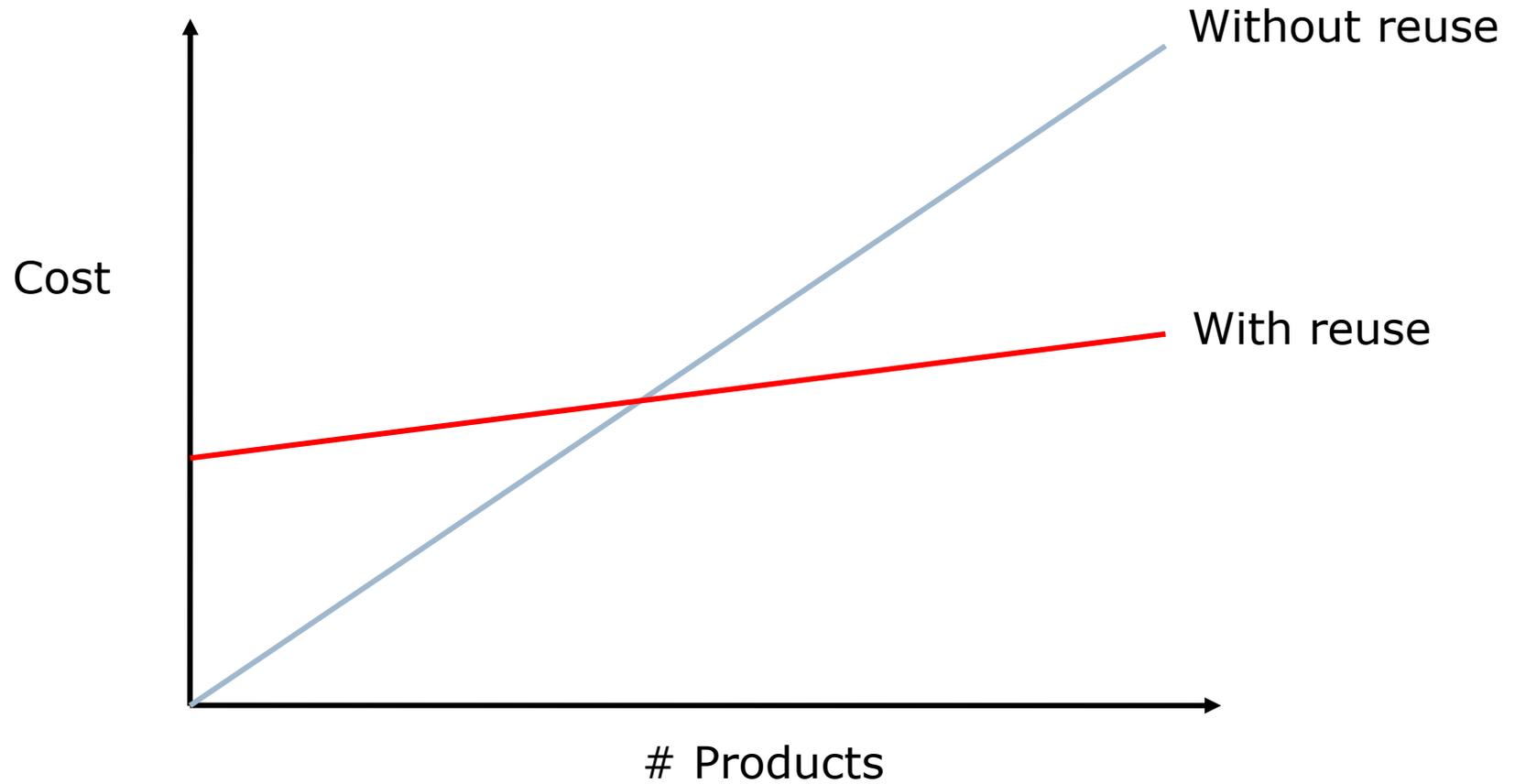
When should you stop writing tests?

When should you stop writing tests?

- When you run out of money...
- When your homework is due...
- When you can't think of any new test cases...
- The *coverage* of a test suite
 - Trying to test all parts of the implementation
 - Statement coverage: percentage of program statements executed
 - Compare to: method coverage, branch coverage, path coverage

DESIGN FOR REUSE

The promise of reuse:



Reuse: Family of development tools

The screenshot shows the Eclipse IDE interface with the 'Software Updates and Add-ons' dialog box open. The dialog has two tabs: 'Installed Software' and 'Available Software'. The 'Available Software' tab is active, showing a list of software updates. The list has columns for 'Name' and 'Version'. The 'Install...' button is highlighted with a mouse cursor.

Name	Version
<input type="checkbox"/> http://download.eclipse.org/releases/ganymede	
<input type="checkbox"/> http://eclipse.svnkit.com/1.2.x/	
<input checked="" type="checkbox"/> http://localhost:8111/update/eclipse/	
<input checked="" type="checkbox"/> jetbrains.teamcity	
<input checked="" type="checkbox"/> JetBrains TeamCity Plugin	4.1.0.8920
<input type="checkbox"/> http://subclipse.tigris.org/update_1.6.x	
<input type="checkbox"/> http://www.perforce.com/downloads/http/p4-wsadm/install/	
<input type="checkbox"/> The Eclipse Project Updates	

Buttons: Install..., Properties, Add Site..., Manage Sites..., Close

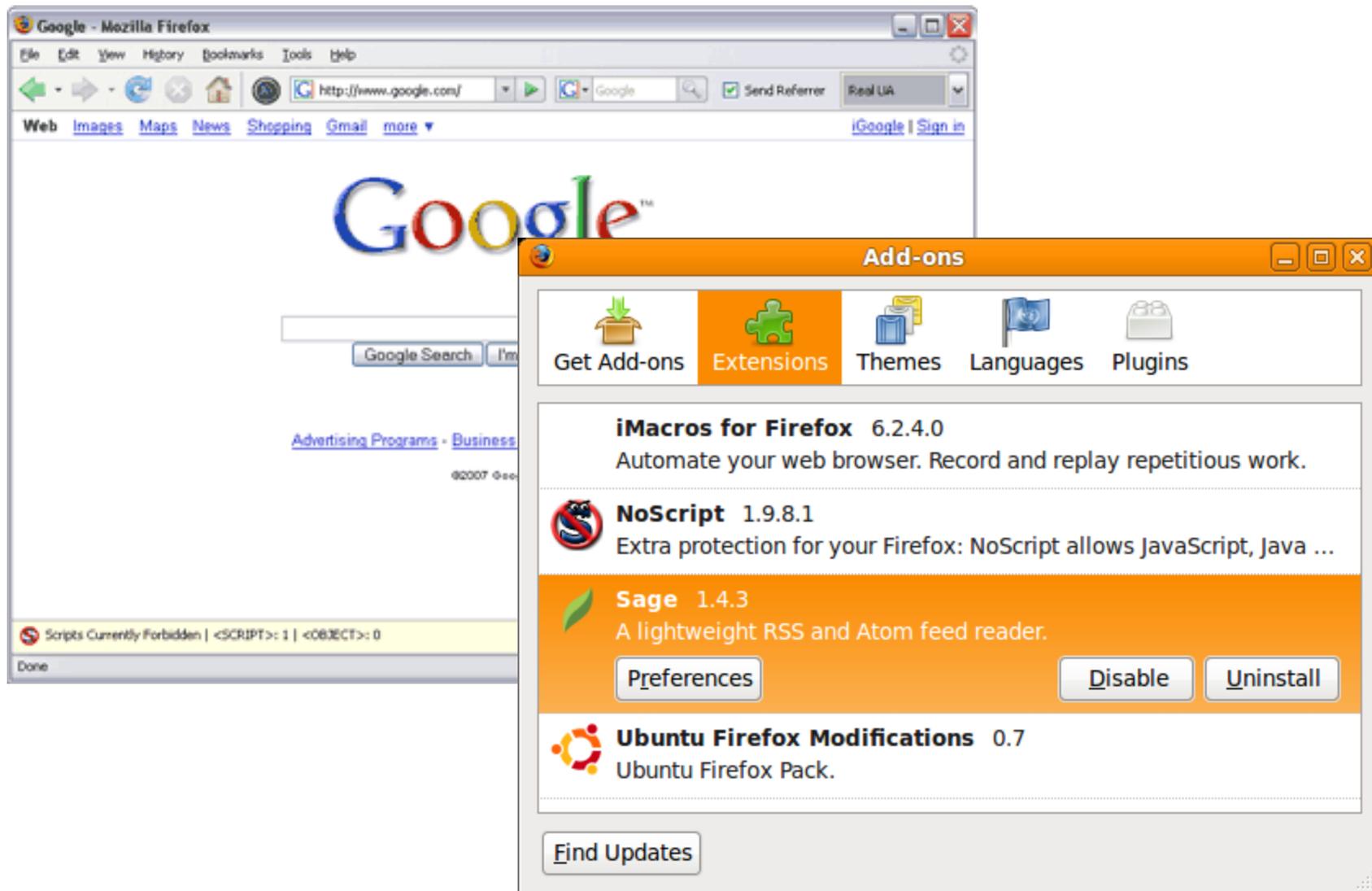
Options:
 Show only the latest versions of available software
 Include items that have already been installed

Open the '[Automatic Updates](#)' preference page to set up an automatic update schedule.

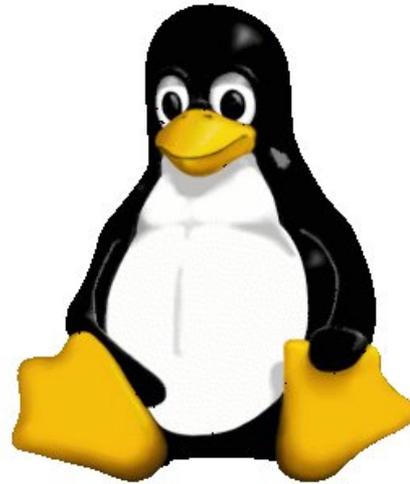
Background IDE elements:
- File: Java - jeti/src/nu/fw/jeti/jjabber/Backend.java - Eclipse
- Team Synchronizing - SVN (/org.tigris.subclipse.core): /org.tigris.subversion.subclipse.core/src/org/tigris/subversion/subclipse/core/SVNClientManager.java ...
- SVNClientManager.java code snippet:

```
String svnClientI  
String svnAdminDi  
File configDir =  
boolean fetchChar  
HashMap clients =  
void startup(IProd
```

Reuse: Web browser extensions



Reuse and variation: Flavors of Linux



Today: Class-level reuse with inheritance

- Inheritance
 - Java-specific details for inheritance
- Behavioral subtyping: Liskov's Substitution Principle
- Next week:
 - Delegation
 - Design patterns for improved class-level reuse
- Later in the course:
 - System-level reuse with libraries and frameworks

IMPLEMENTATION INHERITANCE AND ABSTRACT CLASSES

Variation in the real world: types of bank accounts

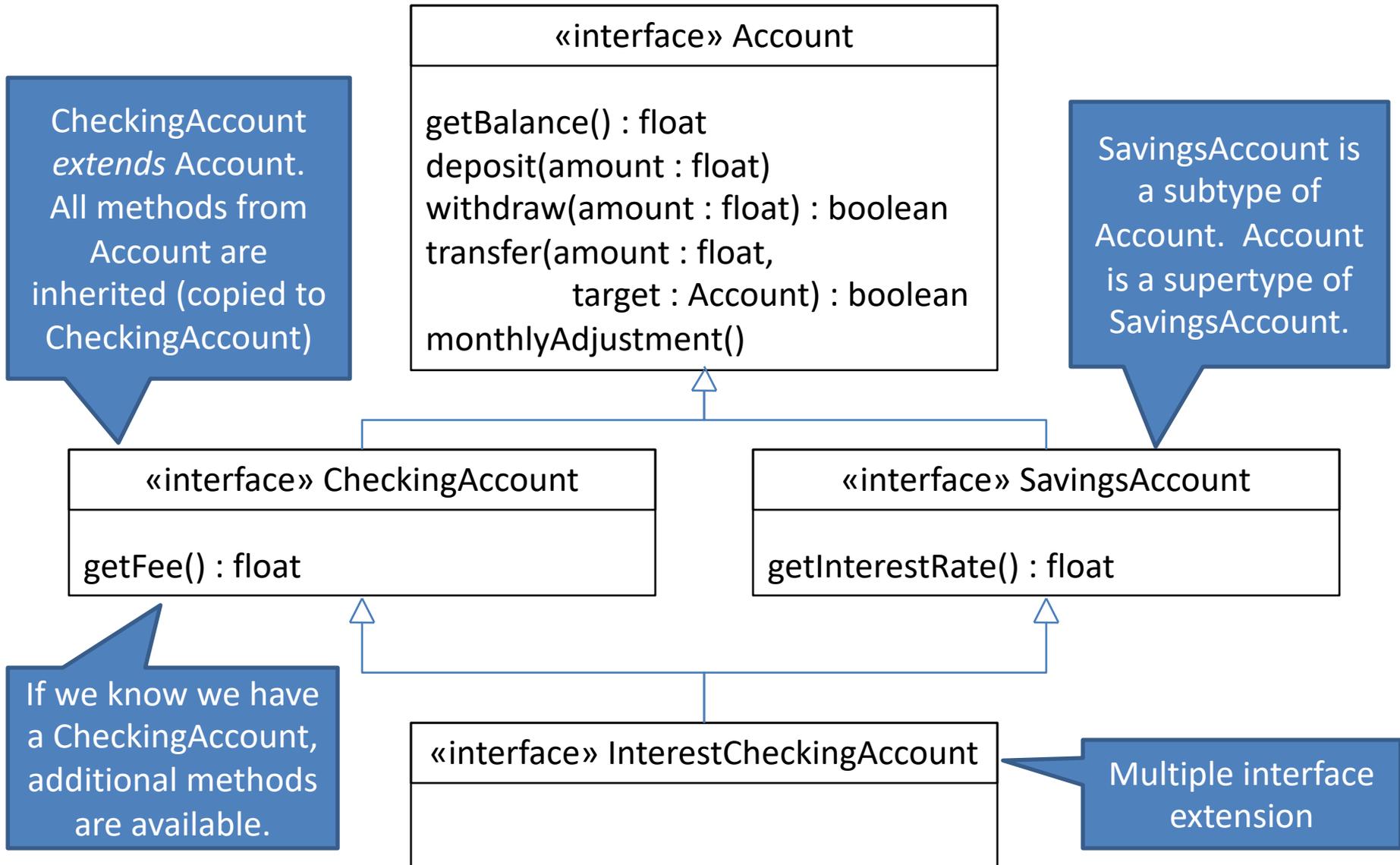
«interface» CheckingAccount

```
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
```

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();
}

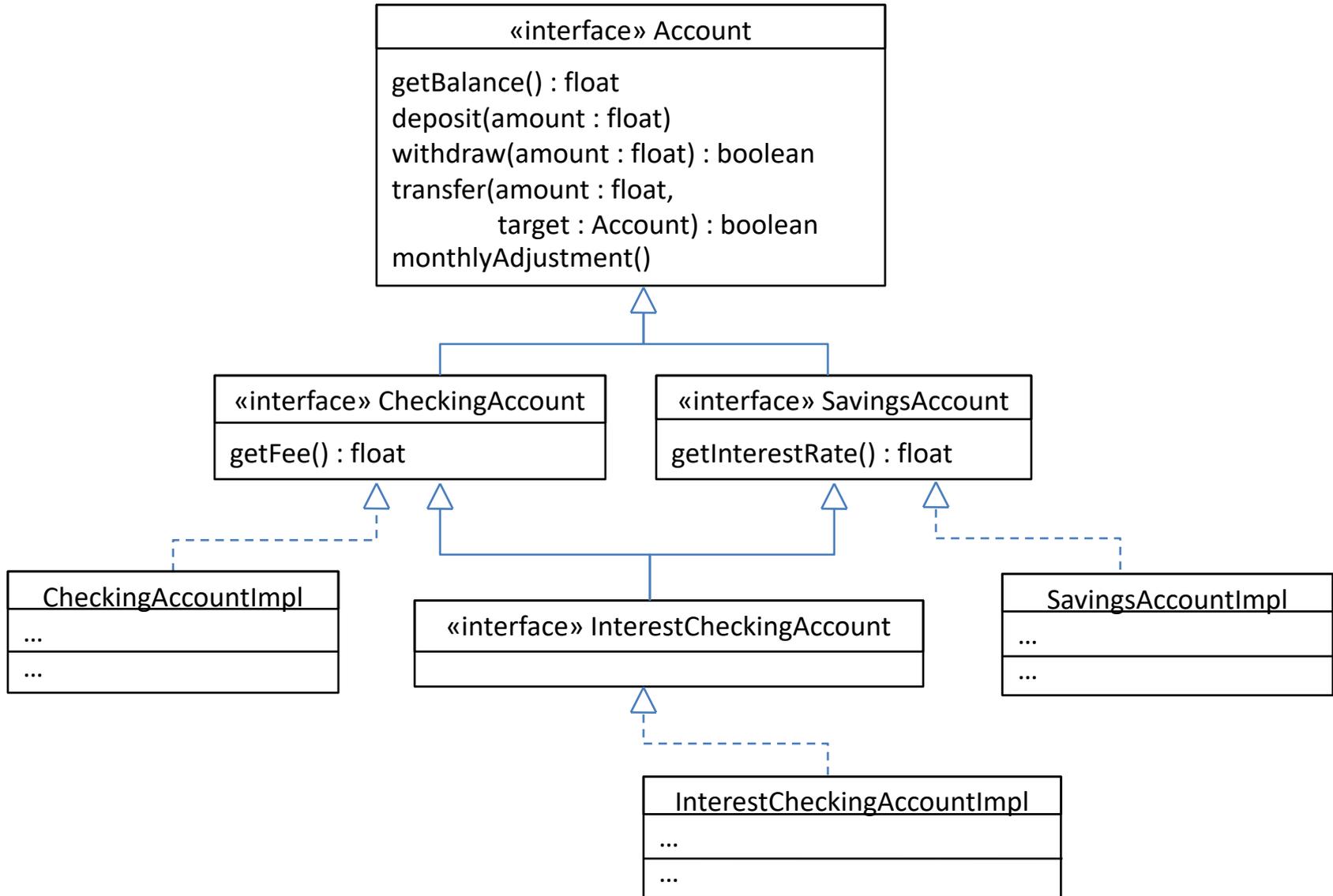
public interface InterestCheckingAccount
    extends CheckingAccount, SavingsAccount {
}
```

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:

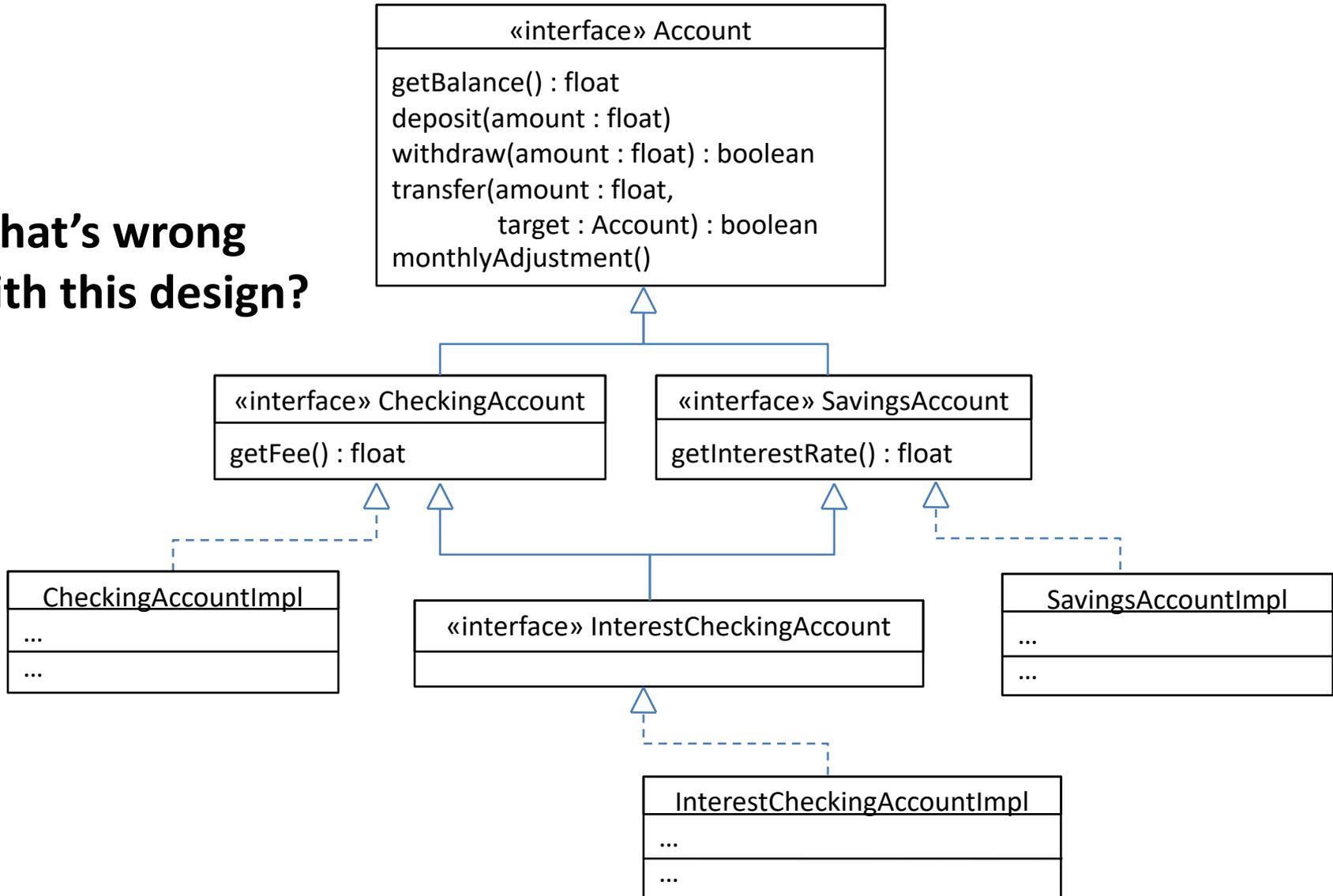
```
If today is the last day of the month:  
  For each acct in allAccounts:  
    acct.monthlyAdjustment();
```

Implementation inheritance for code reuse



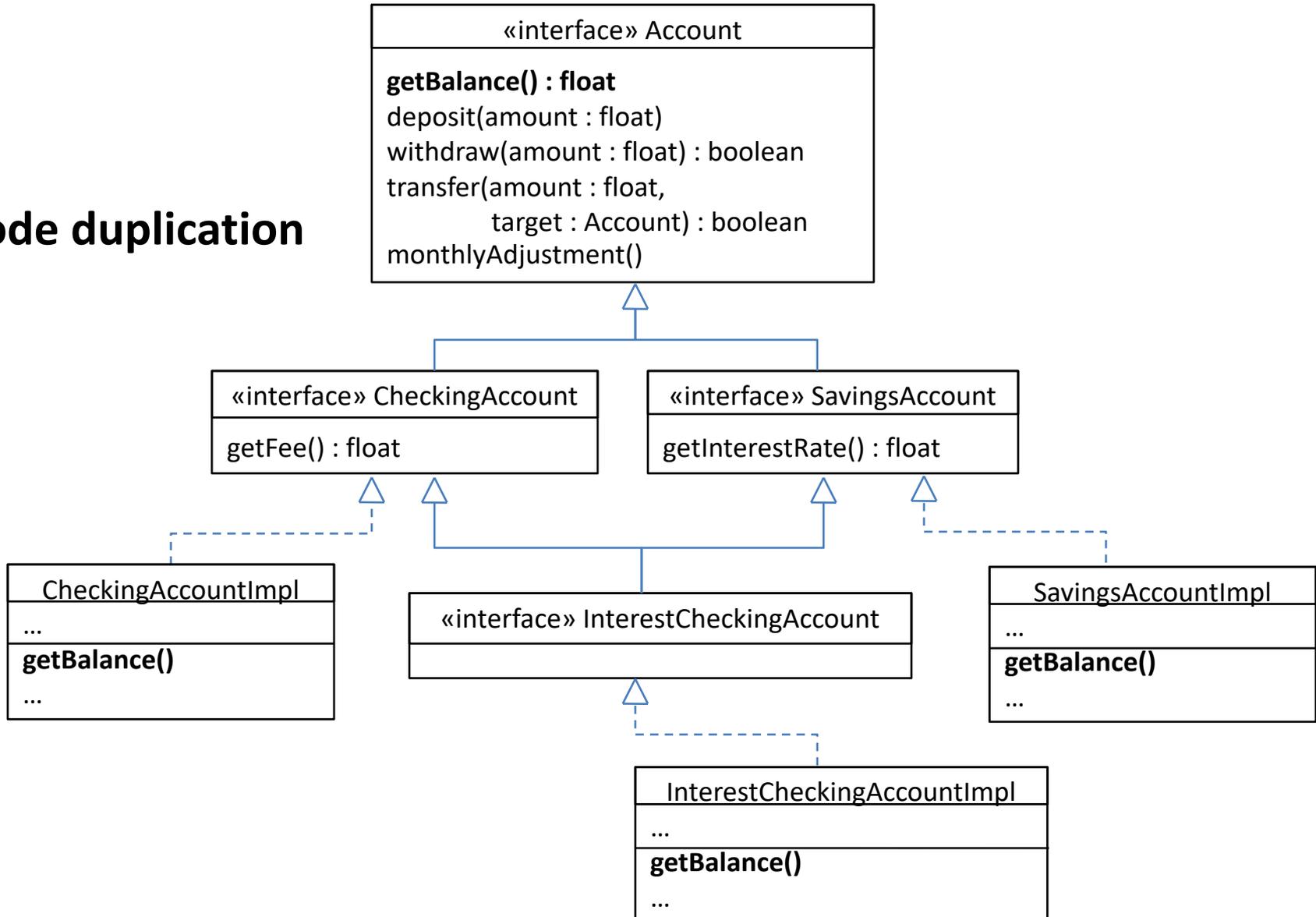
Implementation inheritance for code reuse

What's wrong with this design?



Implementation inheritance for code reuse

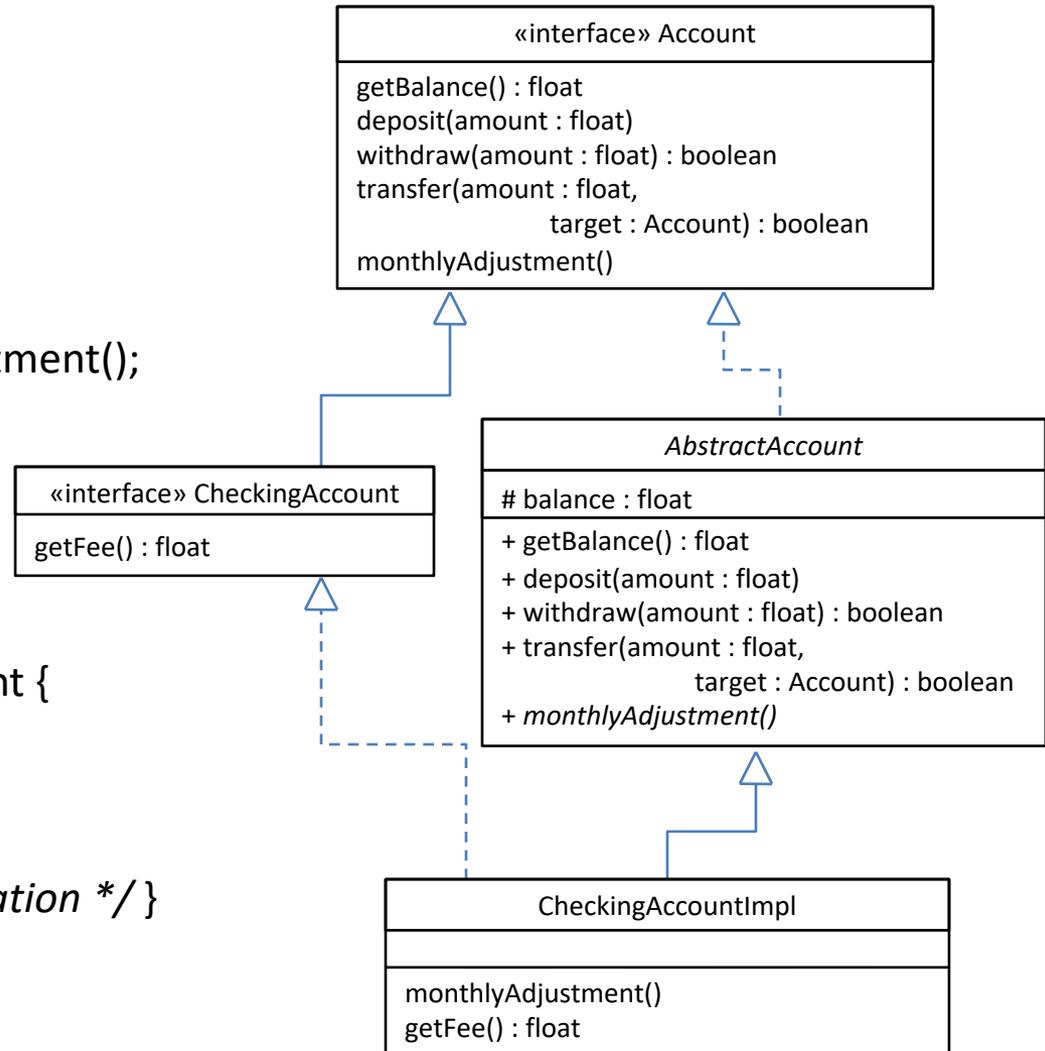
Code duplication



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...
}
```

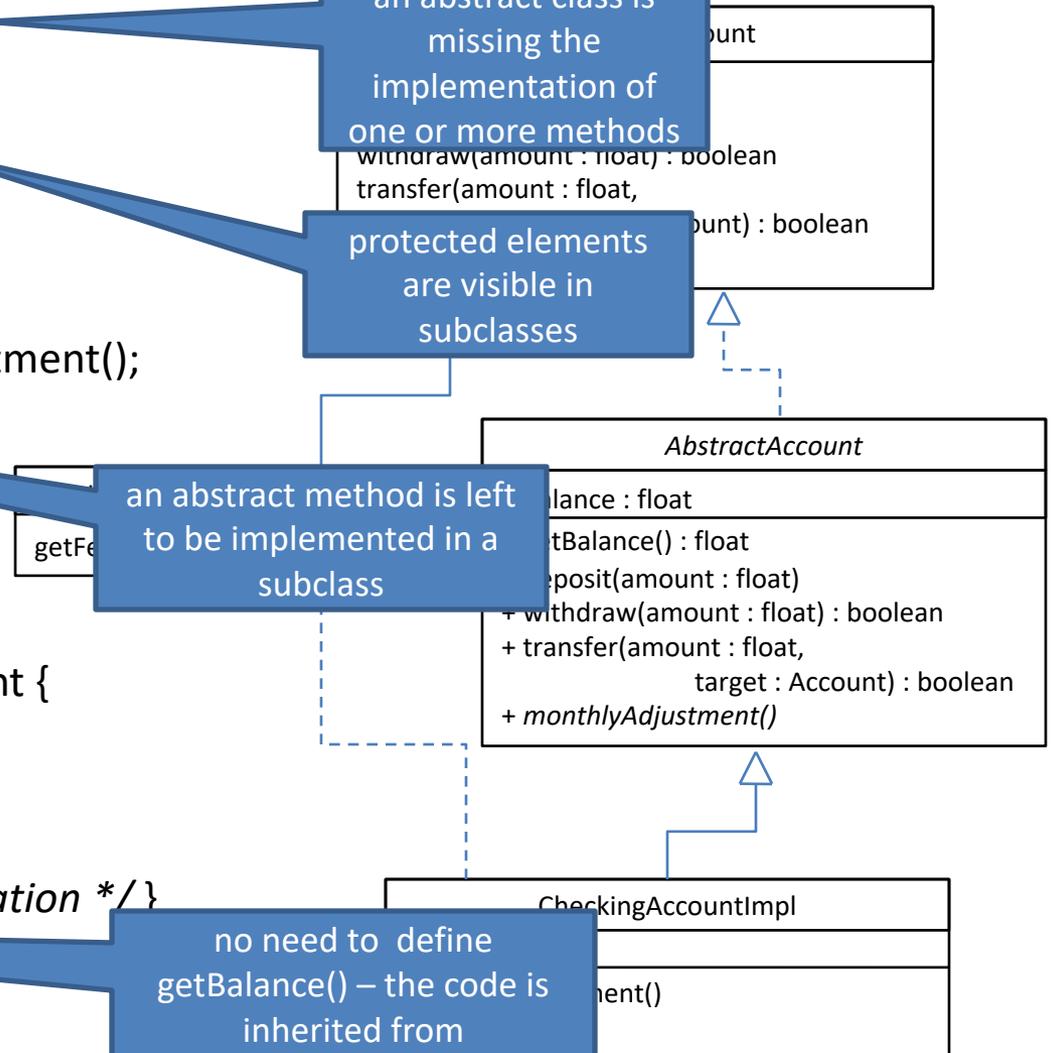
```
public class CheckingAccountImpl
    extends AbstractAccount
    implements CheckingAccount {
    public void monthlyAdjustment() {
        balance -= getFee();
    }
    public float getFee() { /* fee calculation */ }
}
```

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

protected elements are visible in subclasses

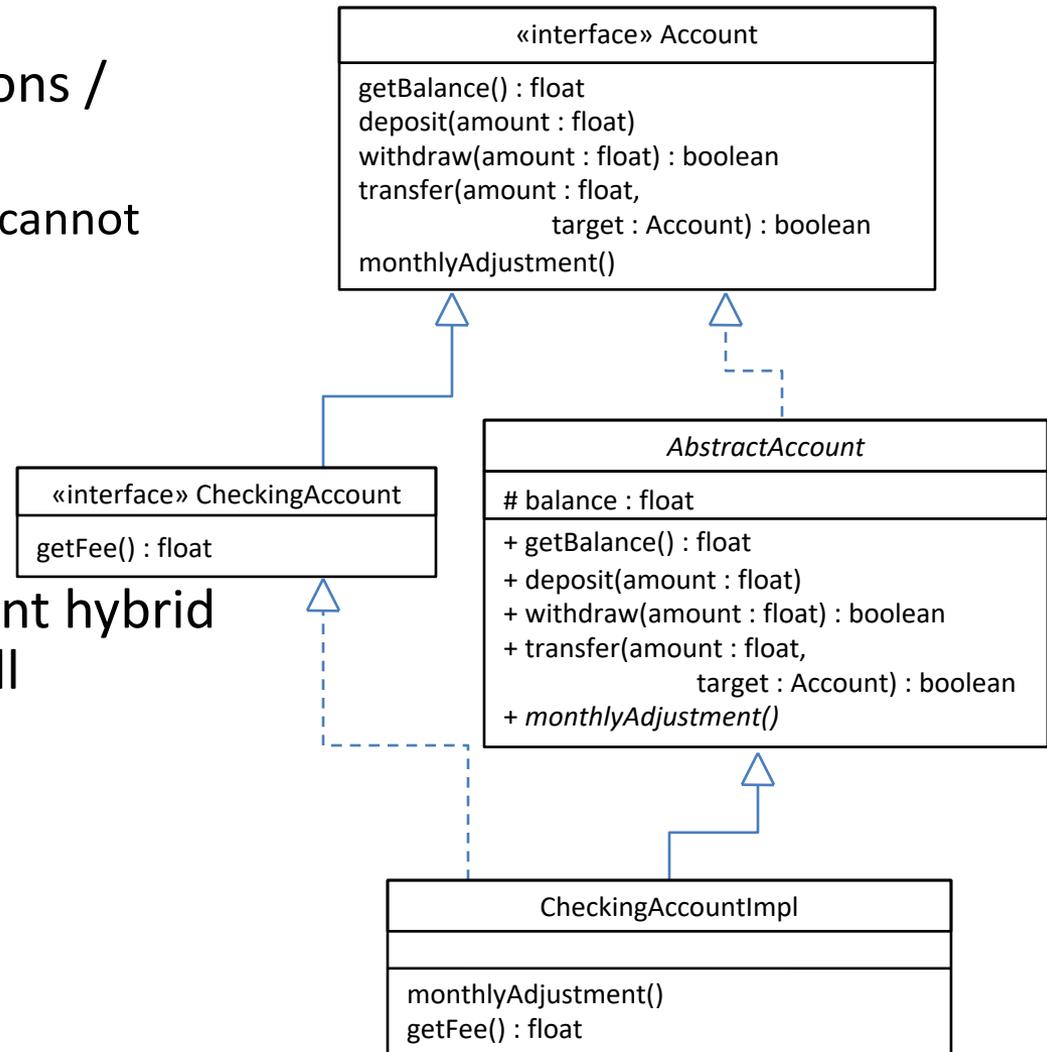
an abstract method is left to be implemented in a subclass

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



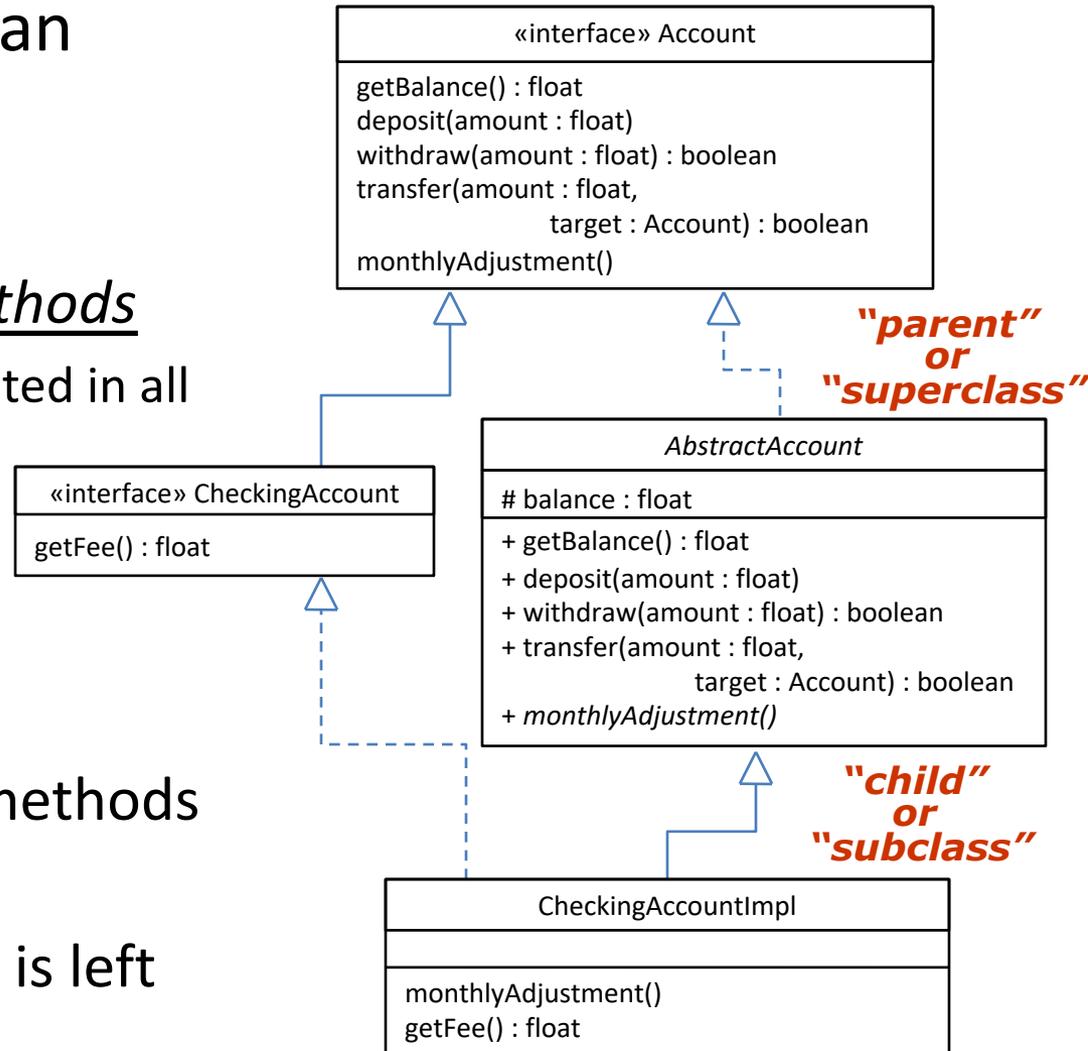
Interfaces vs Abstract Classes vs Concrete Classes

- 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 subtyping

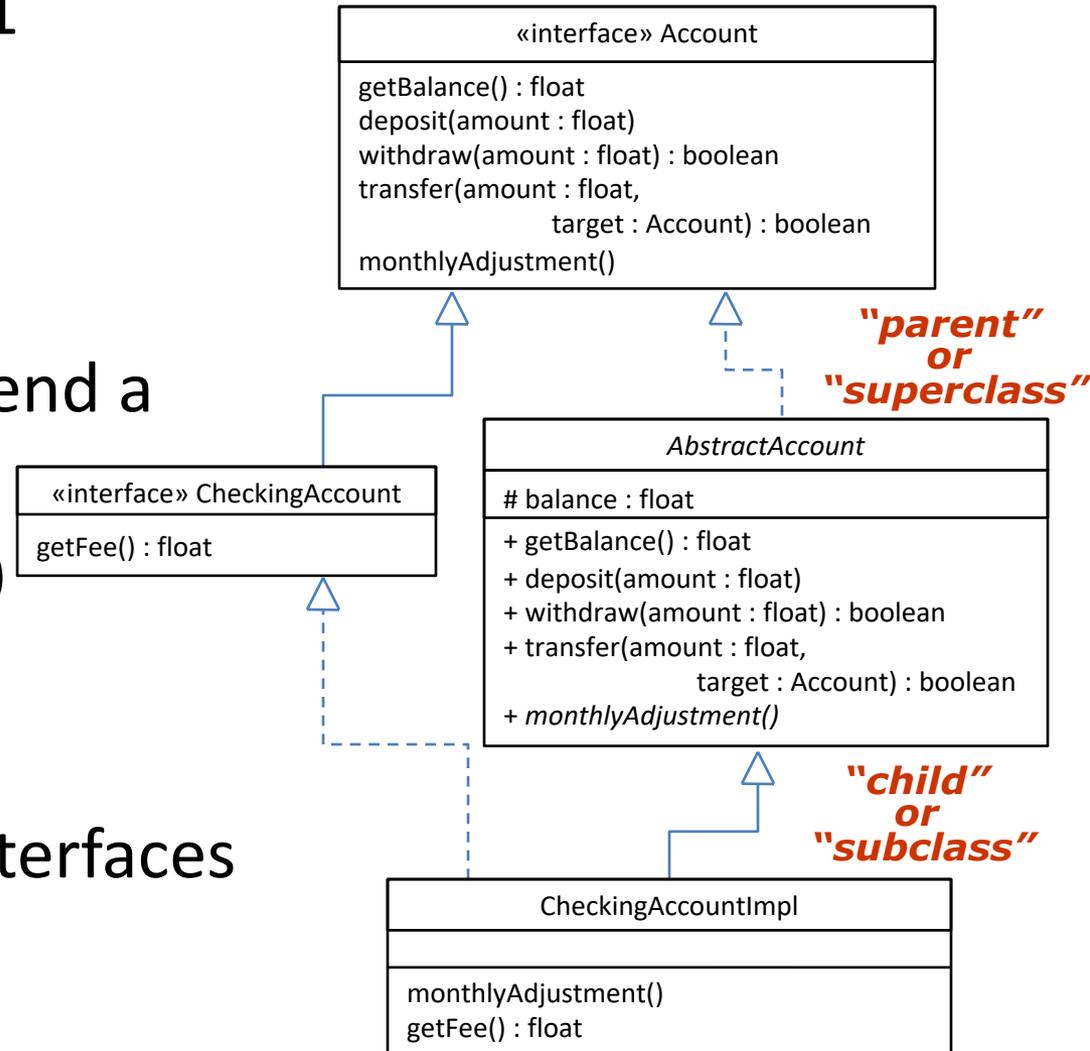
- Inheritance is for code reuse
 - Write code once and only once
 - Superclass features implicitly available in subclass
- Subtyping is for polymorphism
 - Accessing objects the same way, but getting different behavior
 - Subtype is substitutable for supertype

```
class A extends B
```

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

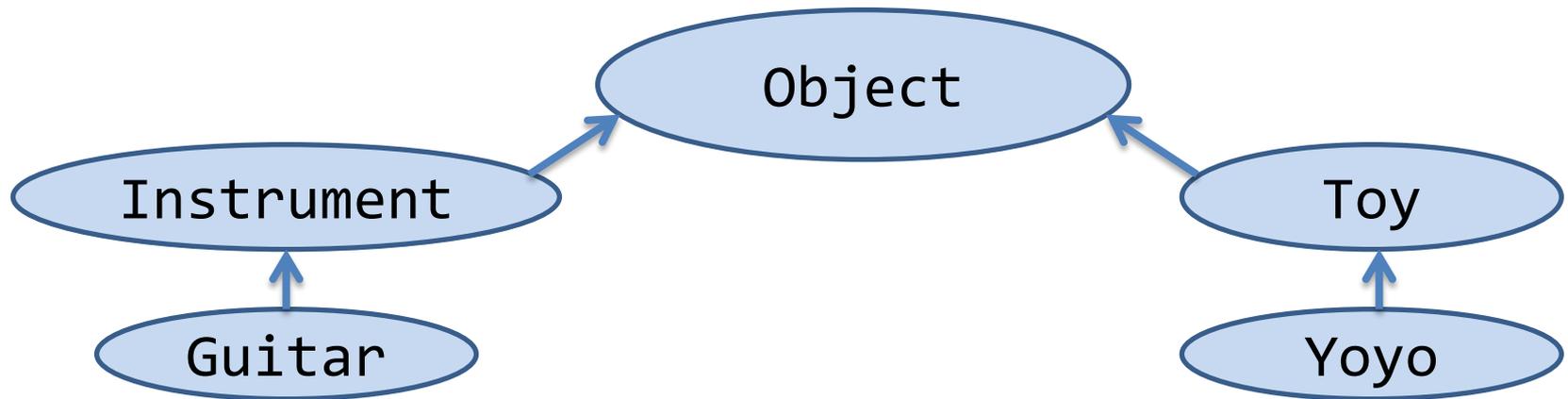
Interfaces vs Abstract Classes vs Concrete Classes

- 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



The class hierarchy

- The root is `Object` (all non-primitives are `Objects`)
- All classes except `Object` have one parent class
 - Specified with an `extends` clause:
`class Guitar extends Instrument { ... }`
 - If `extends` clause is omitted, defaults to `Object`
- A class is an instance of all its superclasses



CLASS INVARIANTS

Recall: 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);
}
```

Recall: 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 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

“SHOULD I BE INHERITING FROM THIS TYPE?”

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

- Applies to specified behavior:
 - Same or stronger invariants
 - Same or weaker preconditions for all methods
 - Same or stronger postconditions for all methods
- 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

This is called the *Liskov Substitution Principle*.

Barbara Liskov

- First Woman to earn PhD in CS in US
- Turing Award (2008)
- MIT Professor



Car is a behavioral subtype of Vehicle

```
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**

Hybrid is a behavioral subtype of Car

```
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() { ... }

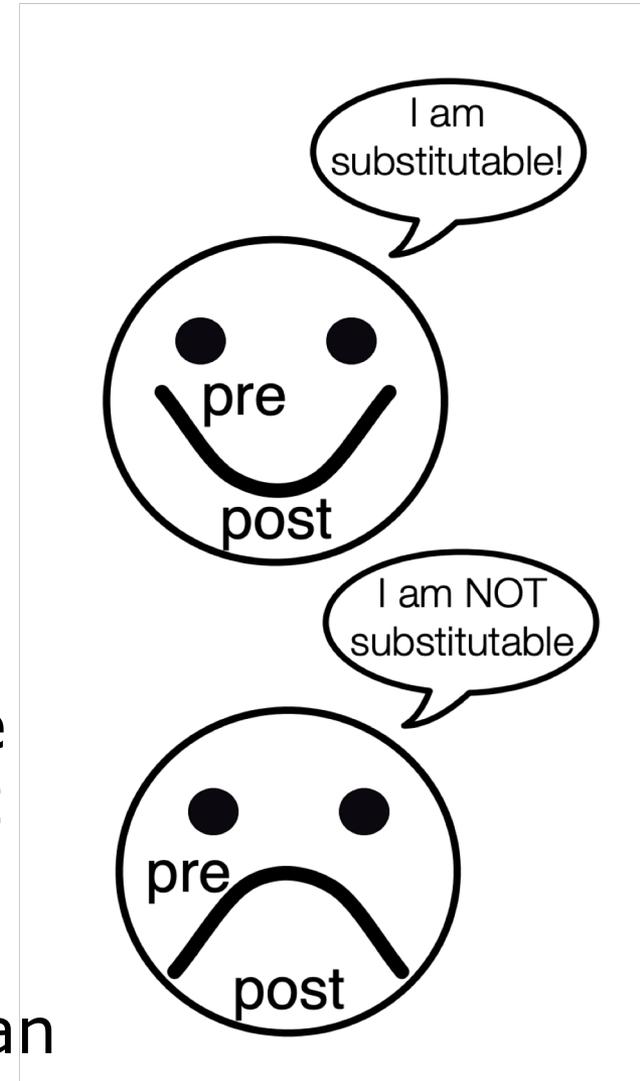
    void accelerate() { ... }

    //@ 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**

Happy and Sad

- 1) An operation is happy if it can be substituted for its super type's method and sad if it cannot.
- 2) A smile is wider at the top than at the bottom; a frown is the opposite.
- 3) Preconditions are at the top of the smile/frown, postconditions are at the bottom of the smile/frown.
- 4) Ergo: Preconditions can be wider (looser/weaker), postconditions can be narrower (tighter/stronger).



Is this Square a behavioral subtype of Rectangle?

```
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?

```
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);  
  }  
}
```

(Yes.)

Is this Square a behavioral subtype of Rectangle?

```
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?

```
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);  
  }  
}
```

(Yes.)

Is this Square a behavioral subtype of Rectangle?

```
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?

```
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);
  }
}
```

(Yes.)

Is this Square a behavioral subtype of Rectangle?

```
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?

```
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);
  }
}
```

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

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

(Yes! But the Square is not a square...)

This Square is *not* a behavioral subtype of Rectangle

```
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;
  //@ ensures w==neww
  && h==old.h;
  void setWidth(int neww) {
    w=neww;
  }
}
```

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

  //@ requires neww > 0;
  //@ ensures w==neww
  && h==neww;
  @Override
  void setWidth(int neww) {
    w=neww;
    h=neww;
  }
}
```

Today

- Functional correctness, continued
- Behavioral subtyping
 - Liskov Substitution Principle
 - The `java.lang.Object` contracts

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

The built-in `java.lang.Object` implementations

- Provide identity semantics:
 - `equals(Object o)`: returns true if `o` refers to this object
 - `hashCode()`: returns a near-random `int` that never changes
 - `toString()`: returns a string consisting of the type and hash code
 - For example: `java.lang.Object@659e0bfd`

The toString() specification

- Returns a concise, but informative textual representation
- Advice: Always override toString(), e.g.:

```
final class PhoneNumber {  
    private final short areaCode;  
    private final short prefix;  
    private final short lineNumber;  
    ...  
    @Override public String toString() {  
        return String.format("(%03d) %03d-%04d",  
            areaCode, prefix, lineNumber);  
    }  
}
```

```
Number jenny = ...;  
System.out.println(jenny);  
Prints: (707) 867-5309
```

The equals(Object) specification

- Must define an equivalence relation:
 - Reflexive: For every object x , $x.equals(x)$ is always true
 - Symmetric: If $x.equals(y)$, then $y.equals(x)$
 - Transitive: If $x.equals(y)$ and $y.equals(z)$, then $x.equals(z)$
- Consistent: Equal objects stay equal, unless mutated
- "Non-null": $x.equals(null)$ is always false

An equals(Object) example

```
public final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof PhoneNumber)) // Does null check
            return false;
        PhoneNumber pn = (PhoneNumber) o;
        return pn.lineNumber == lineNumber
            && pn.prefix == prefix
            && pn.areaCode == areaCode;
    }

    ...
}
```

The hashCode() specification

- Equal objects must have equal hash codes
 - If you override equals you must override hashCode
- Unequal objects should usually have different hash codes
 - Take all value fields into account when constructing it
- Hash code must not change unless object is mutated

A hashCode() example

```
public final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    @Override public int hashCode() {
        int result = 17; // Nonzero is good
        result = 31 * result + areaCode; // Constant must be odd
        result = 31 * result + prefix; // " " " "
        result = 31 * result + lineNumber; // " " " "
        return result;
    }

    ...
}
```

An Object method exercise

Provide all code needed for a reasonable equals method:

```
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;
    }
    ...
}
```

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);  
}
```

Summary

- Please complete the course reading assignments
- Test early, test often!
- Subtypes must fulfill behavioral contracts
- Always override `hashCode` if you override `equals`
- Always use `@Override` if you intend to override a method
 - Or let your IDE generate these methods for you...