Principles of Software Construction: Objects, Design and Concurrency

Encapsulation and Inheritance

15-214 toad

Fall 2013

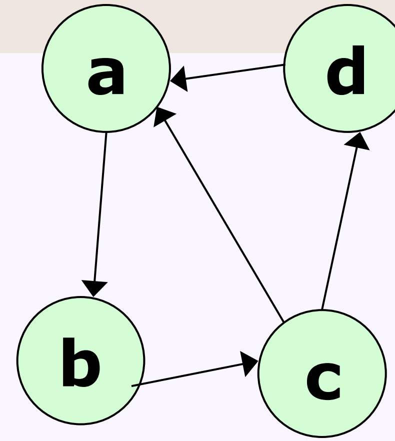
Jonathan Aldrich **Charlie Garrod**

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

- Office hours updates (all times p.m.)
  - Sunday **8 – 10**: Mat in GHC 41xx
  - Monday **8 – 10**: Shannon in GHC 41xx
  - Tuesday 6 – 8: Dan in GHC 41xx
  - Tuesday 8 – 10: Alex in GHC 41xx
  - **Wednesday 2:30 – 3:30: Jonathan in Wean 4216**
  - Wednesday 6 – 8: Bailey in GHC 41xx
  - Thursday 7 – 9: Beth Anne in GHC 41xx
  - Friday 1:30 – 3: Charlie in Wean 5101
- Homework 1 due next Tuesday…

Homework 1: Representing graphs



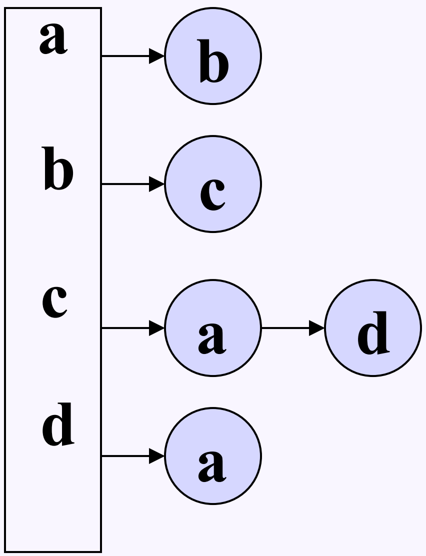
Two common representations

- *Adjacency matrix*
- *Adjacency list*

Adjacency matrix

source \ target	a	b	c	d
a	0	1	0	0
b	0	0	1	0
c	1	0	0	1
d	1	0	0	0

Adjacency list

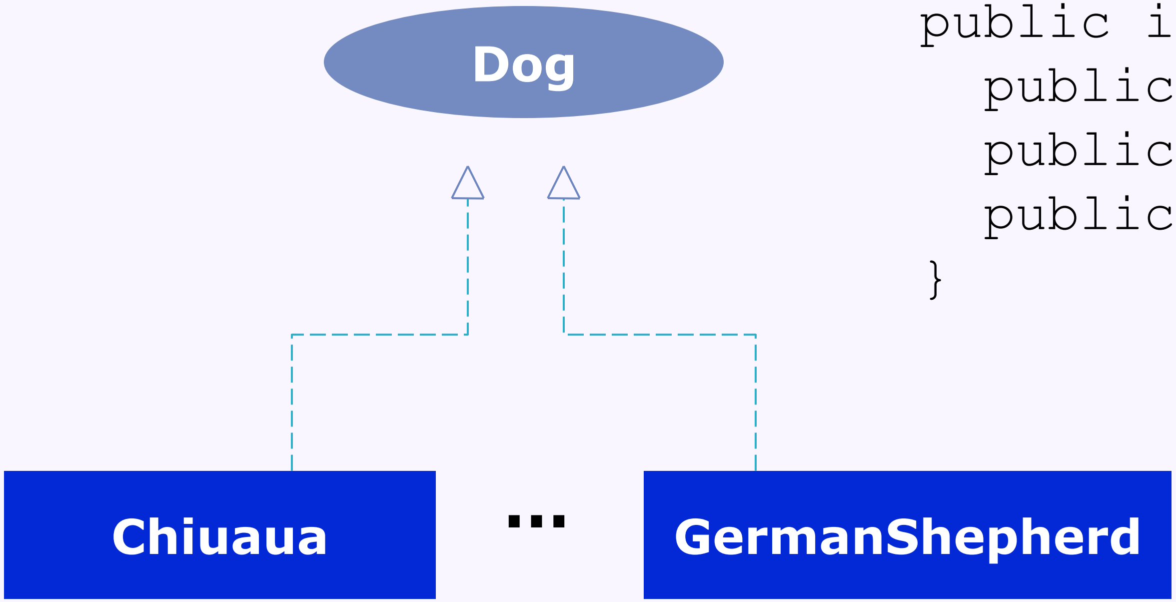


Key concepts from Thursday

Key concepts from Thursday

- Objects, classes, and references
- Encapsulation and visibility
- Polymorphism
  - Interfaces
  - Introduction to method dispatch
- Object equality

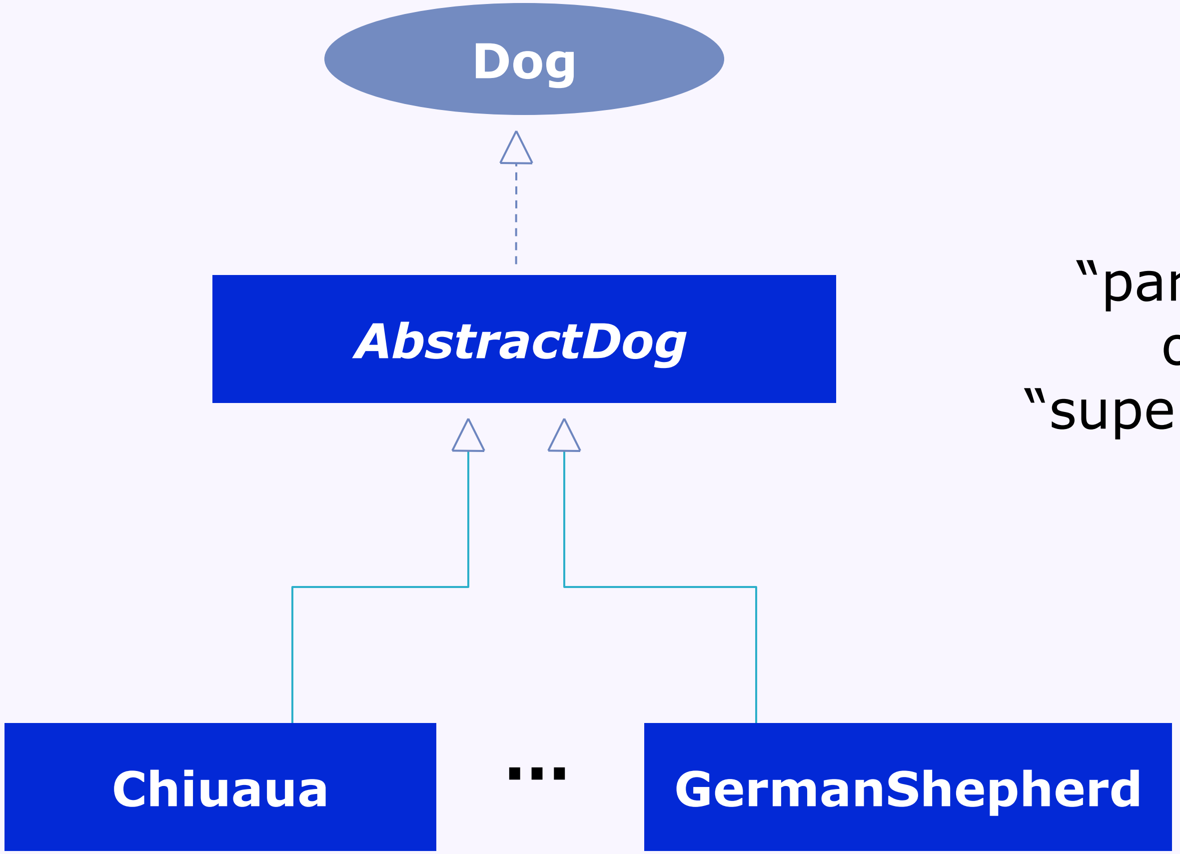
E.g., a Dog interface




```
public interface Dog {
  public String getName();
  public String getBreed();
  public void bark();
}
```

```
public class Chiuaua implements Dog {
  public String getName()  { return "Bob"; }
  public String getBreed() { return "Chiuaua"; }
  public void bark()       { /* How do I bark? */ }
}
```

A preview of inheritance



"parent"
or
"superclass"

"child"
or
"subclass"

Key concepts for today

- Encapsulation, revisited
  - Packages
    - Name and visibility management
    - Qualified names
  - General design principles
- Inheritance and polymorphism
  - For maximal code re-use
  - Diagrams to show the relationships between classes
  - Polymorphism and its alternatives
  - Types and type-checking
  - Method dispatch, revisited
  - Etc.

Programming languages: a complex view

	Small-scale	Larger-scale
Data	Primitives Arrays Structures	Objects Heaps
Control	Basic (if, while, ;) Function/method calls	Method dispatch Concurrency
Naming and Reference	Local variables Parameters	Package, imports Visibility Qualification

Java packages

- Packages divide the Java namespace to organize related classes

```
package edu.cmu.cs.cs214.geo;

class Point {
  private int x, y;
  public int getX() { return x; } // a method; getY() is similar
  public Point(int px, int py) { x = px; y = py; } // …
}
class Rectangle {
  private Point origin;
  private int width, height;
  public Point getOrigin() { return origin; }
  public int getWidth() { return width; }
  // …
}
```

Packages are another mechanism of encapsulation

- Visibility of names:
  - `public`: visible everywhere
  - `protected`: visible within package and also to subclasses
  - default (no modifier): visible only within package
  - `private`: visible only within class

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N

Packages and qualified names

- E.g., three ways to refer to a `java.util.Queue`:
  - Use the full name:

```
java.util.Queue q = …;
q.add(…);
```

  - Import `java.util.Queue`, then use the unqualified name:

```
import java.util.Queue;
Queue q = …;
```

  - Import the entire `java.util` package:

```
import java.util.*;
Queue q = …;
```

- Compiler will warn about ambiguous references
  - Must then use qualified name to disambiguate

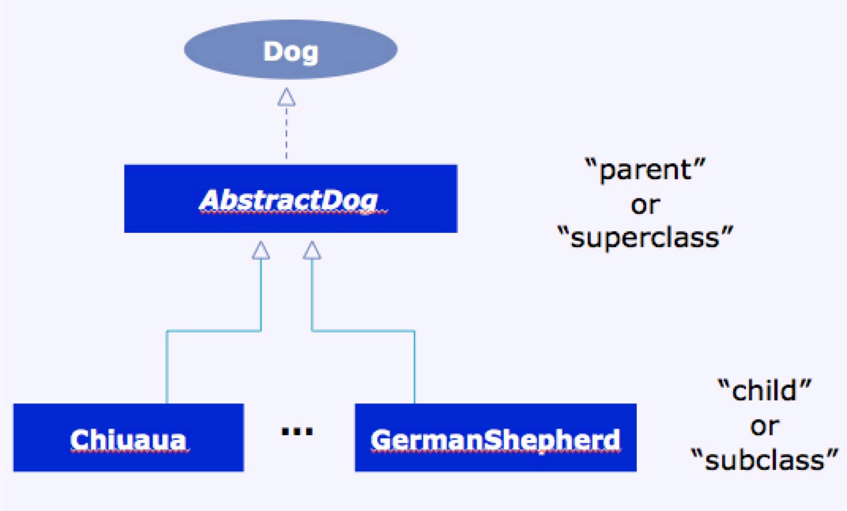
Encapsulation design principles

- Restrict accessibility as much as possible
  - Make data and methods `private` unless you have a reason to make it more visible

"The single most important factor that distinguishes a well-designed module from a poorly designed one is the degree to which the module hides its internal data and other implementation details." -- Josh Bloch

An introduction to inheritance

- A dog of an example:
  - Dog.java
  - AbstractDog.java
  - Chiuaua.java
  - GermanShepherd.java

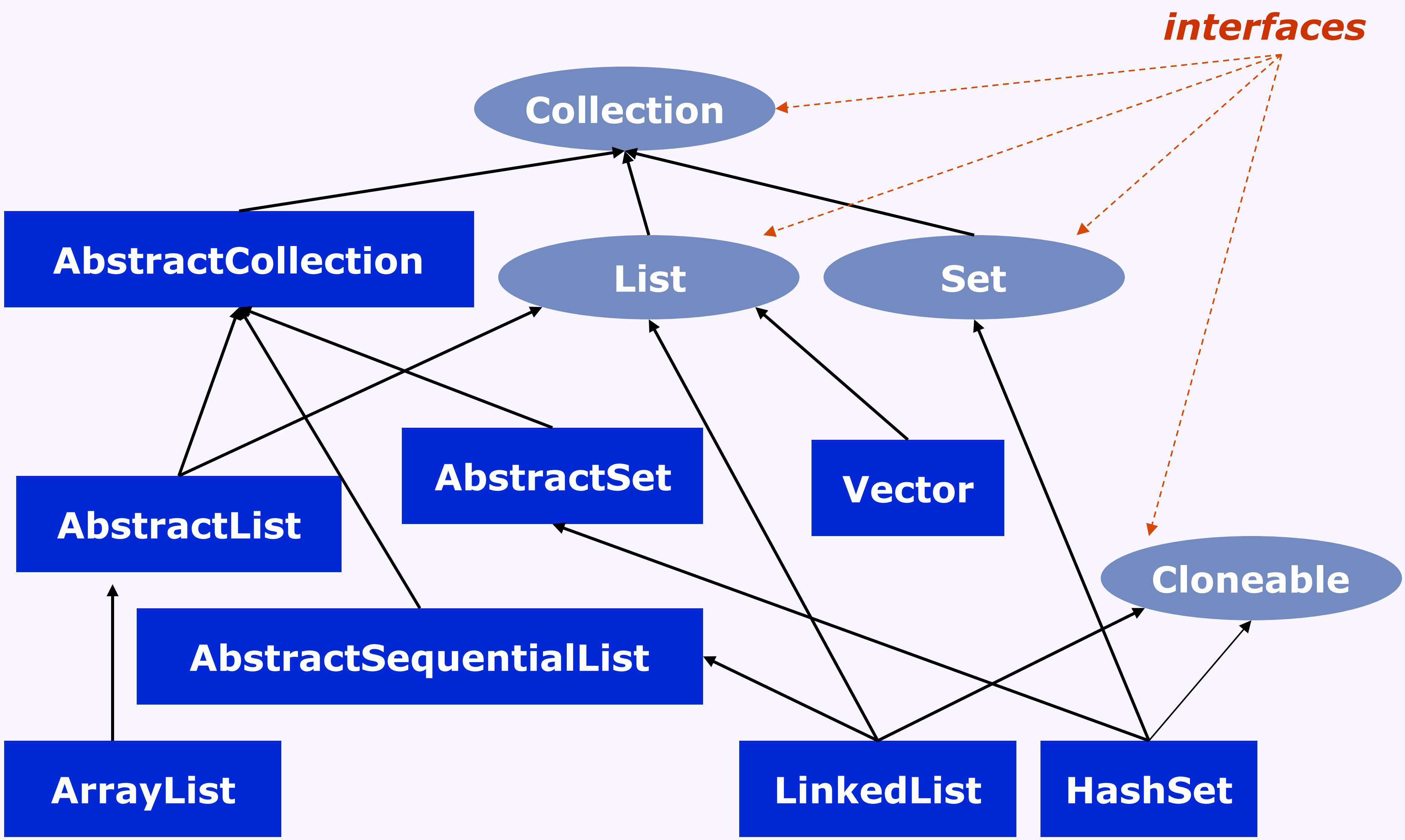


- Typical roles:
  - An interface define expectations / commitment for clients
  - An *abstract class* is a convenient hybrid between an interface and a full implementation
  - Subclass *overrides* a method definition to specialize its implementation

Inheritance: a glimpse at the hierarchy

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

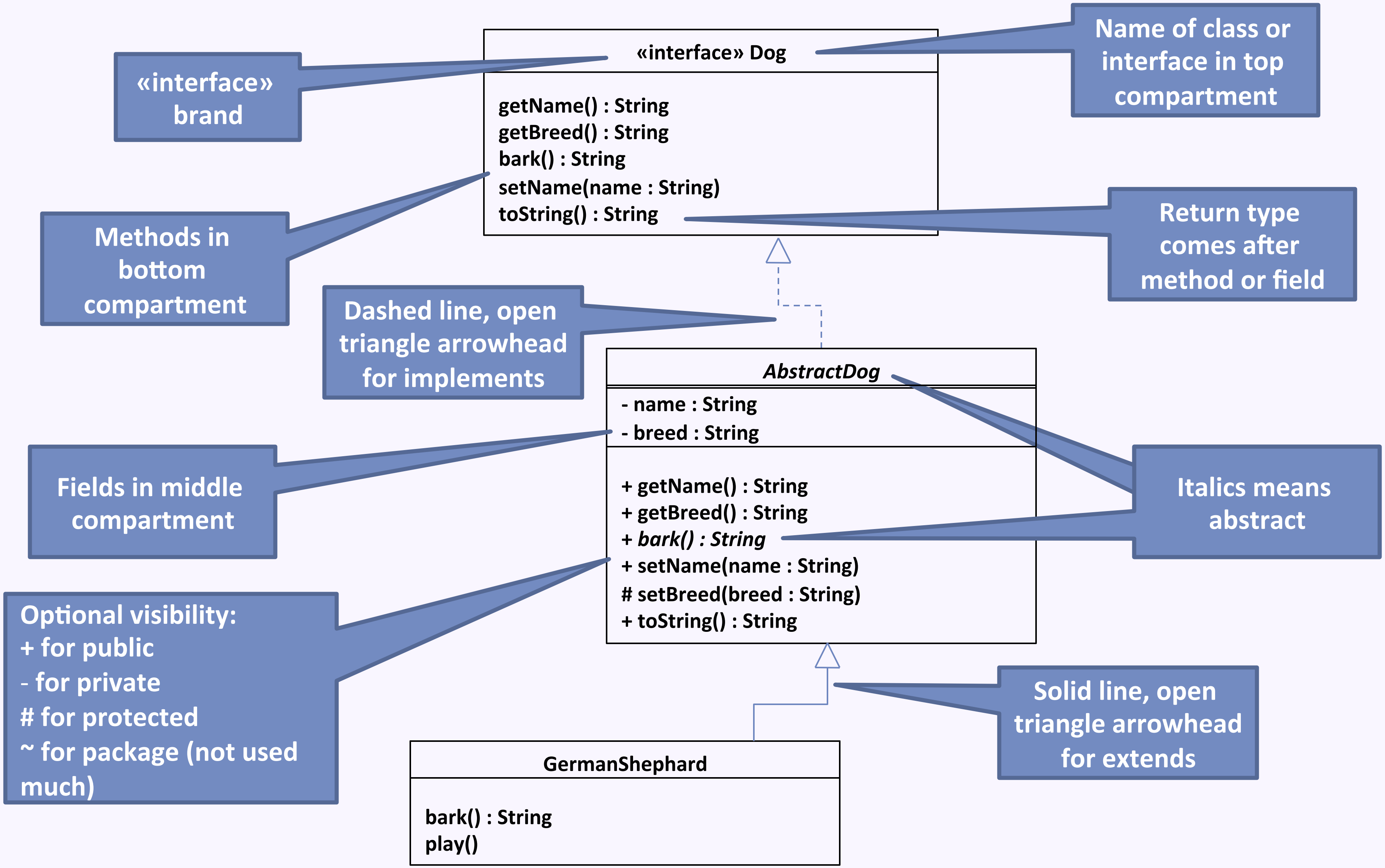
JavaCollection API (excerpt)



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

Aside: UML class diagram notation

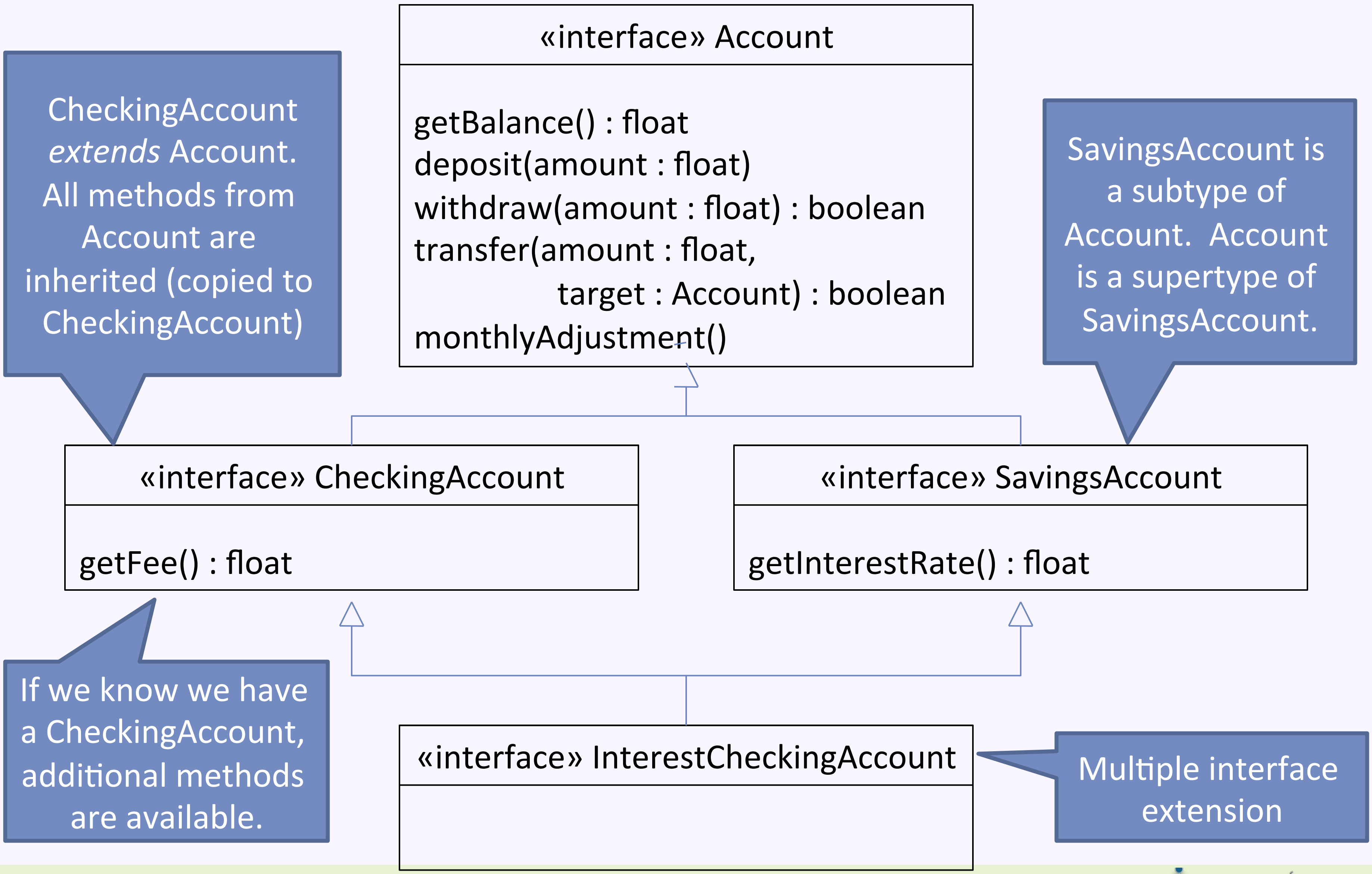


Another example: different kinds of bank accounts

«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



A better design: An account type hierarchy

«interface» Account

t) : boolean

transfer(amount : float,

```
public interface CheckingAccount
        extends Account {
    …
}
```

Account are
inherited (copied to
CheckingAccount)

SavingsAccount is
a subtype of
Account. Account

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

«interface» CheckingAccount

getFee() : float

«interface» SavingsAccount

getInterestRate() : float

If we know we have a CheckingAccount, additional methods are available.

«interface» InterestCheckingAccount

Multiple interface extension

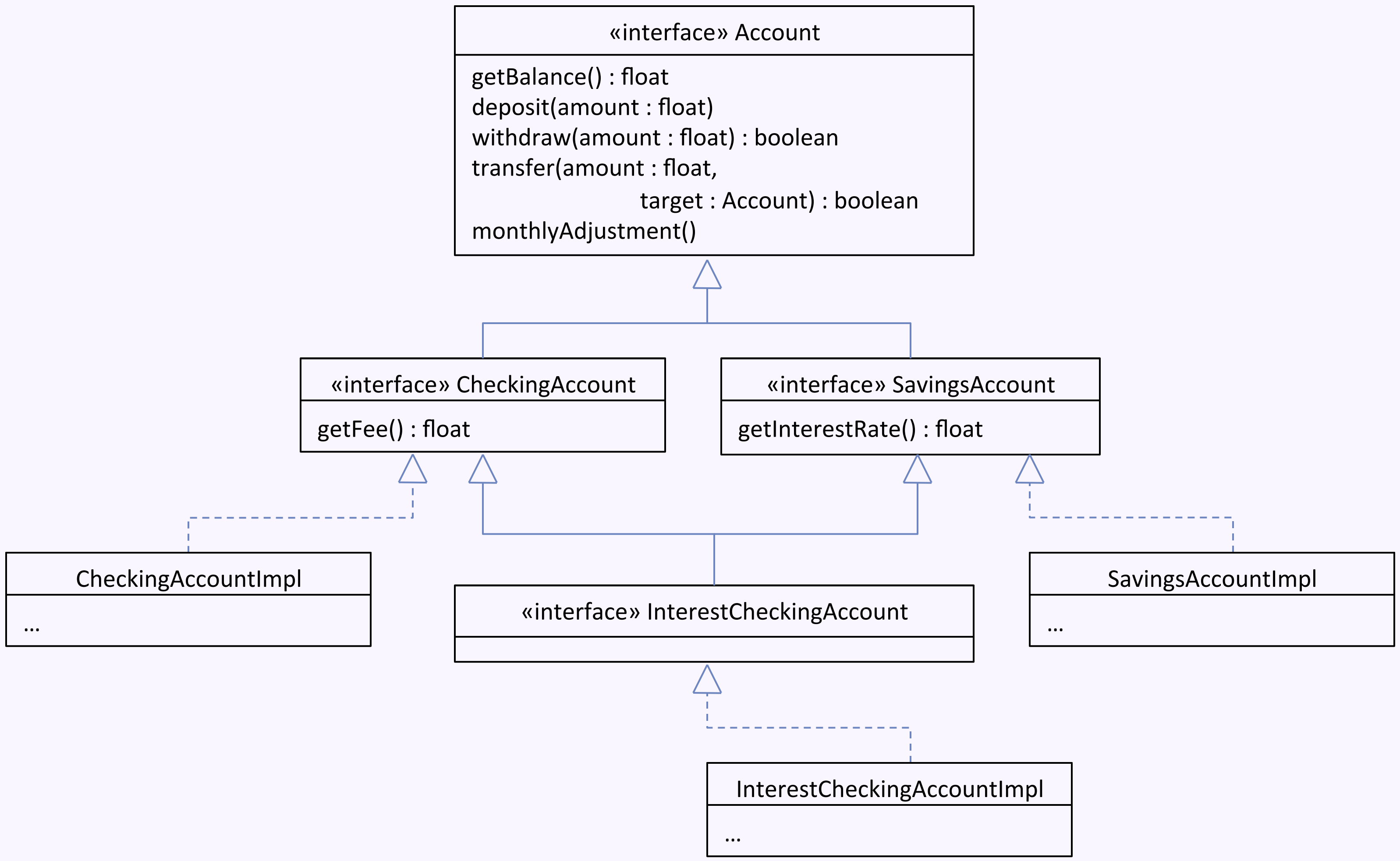
The power of object-oriented interfaces

- 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:

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

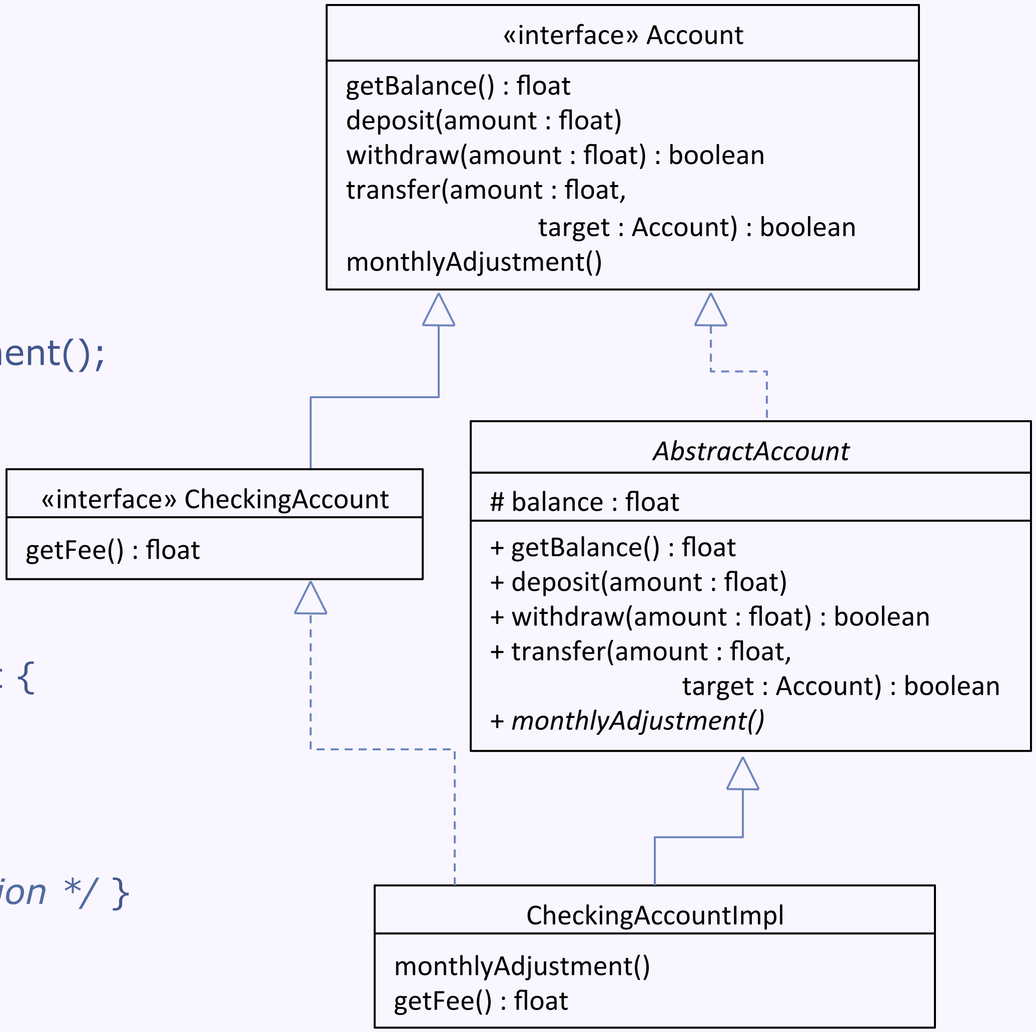
  - See the DogWalker example

One implementation: Just use interface inheritance



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

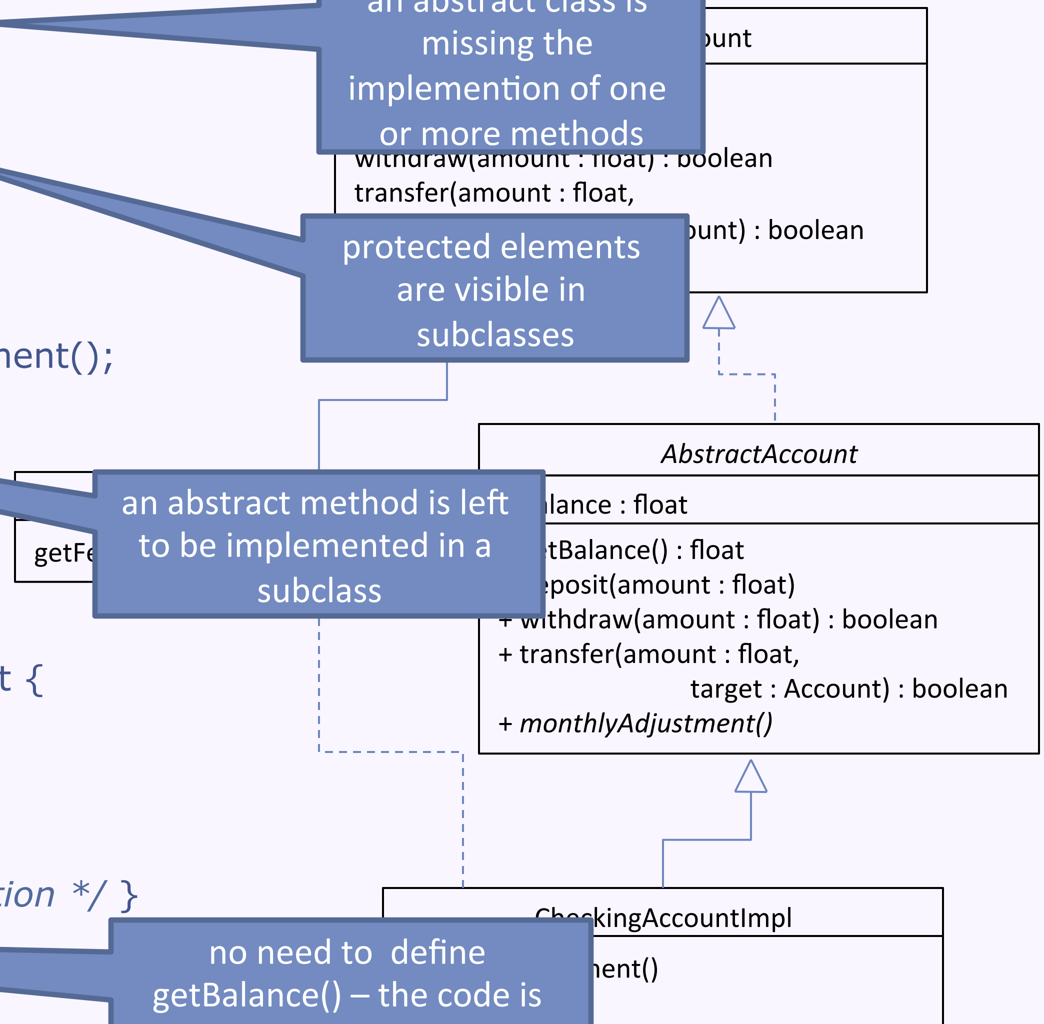
```
public abstract class AbstractAccount
            implements Account {
   protected float balance = 0.0;
   public float getBalance() {
            return balance;
   }
   abstract public void monthlyAdjustment();
   // other methods…
}

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

an abstract class is missing the implemention 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

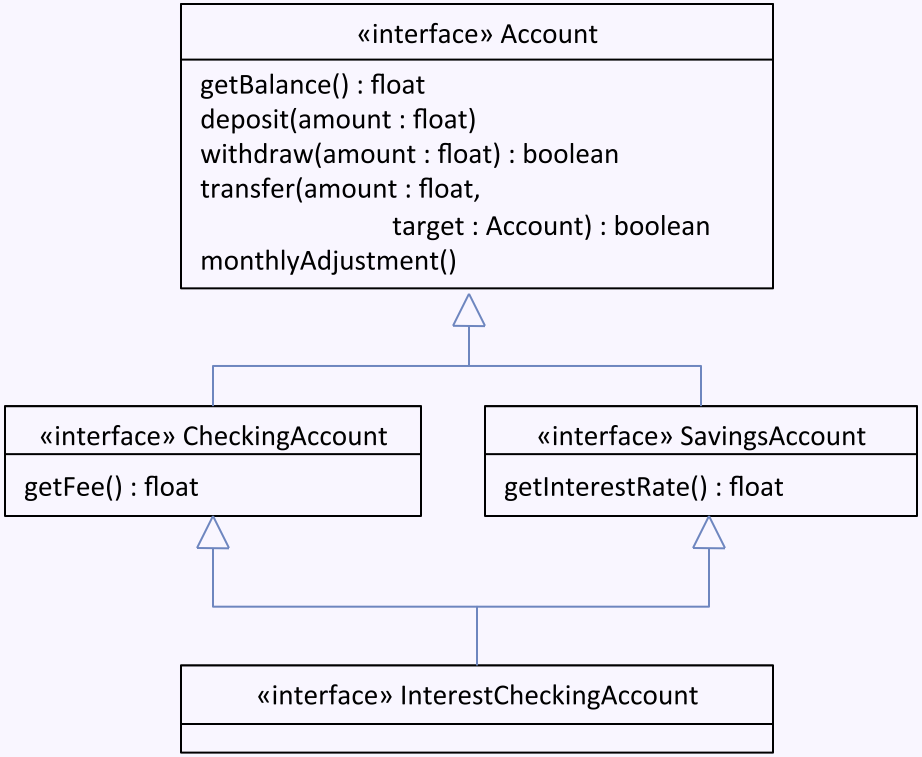
```
class A extends B
```

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

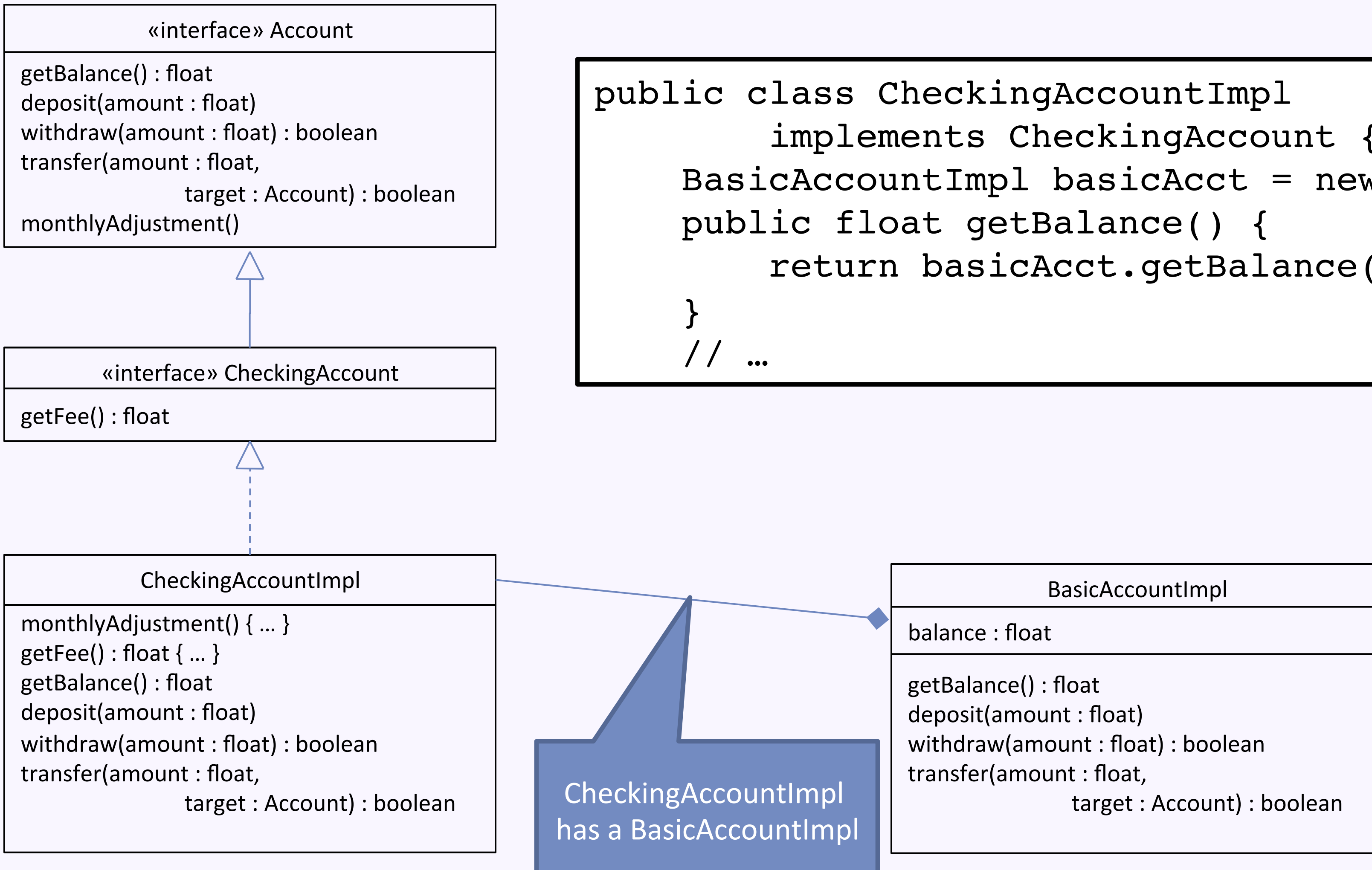
```
class A implements I
class A extends B
```

Challenge: Is inheritance necessary?

- Can we get the same amount of code reuse without inheritance?



Reuse via *composition* and *forwarding*




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

Inheritance vs. composition

- Composition can be cleaner than inheritance
  - Reused code in a separate object
- Inheritance has less boilerplate code
  - No forwarding functions
  - Easier to avoid recursive dependencies
- Inheritance violates principles of encapsulation
  - Subclass dependent on superclass implementation
- Advice: Use inheritance sparingly
  - Before you define a class `Foo` to extend `Bar`, ask: "Is every Foo really a Bar?"

Extended re-use 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 AbstractAcount 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

Constructor calls with `this` and `super`

```
public class CheckingAccountImpl
      extends AbstractAcount 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… */ }
```

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

Invokes another constructor in this same class

Inheritance Details: `final`

- A final class: cannot extend the class
  - e.g., `public final class CheckingAccountImpl { …`
- A final method: cannot override the method
- A final field: cannot assign to the field
  - (except to initialize it)

- Why might you want to use final in each of the above cases?

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

Inheritance Details: `instanceof`

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

```
Account acct = …;
float adj = 0.0;
if (acct instanceof CheckingAccount) {
  checkingAcct = (CheckingAccount) acct;
  adj = checkingAcct.getFee();
}
```

- Advice: avoid `instanceof` if possible

Typechecking

- The key idea: Analyze a program to determine whether each operation is applicable to the types it is invoked on

- Benefits:
  - Finds errors early
    - e.g., `int h = “hi” / 2;`
  - Helps document program code
    - e.g.,

```
baz(frob) { /* what am I supposed to do
                with a frob? */ }
void baz(Car frob) { /* oh, look,
                        I can drive it! */ }
```

Value Flow and Subtyping

- Value flow: assignments, passing parameters
  - e.g., `Foo f = expression;`
  - Determine the type T_{source} of the source expression
  - Determine the type T_{dest} of the destination variable `f`
  - Check that T_{source} is a subtype of T_{dest}
- Subtype relation *A* <: *B*
  - *A* <: B if *A* extends *B* or *A* implements *B*
  - Means you can substitute a thing of type *A* for a thing of type *B*
- Subtypes are:
  - Reflexive: *A* <: *A*
  - Transitive: if *A* <: *B* and *B* <: *C* then *A* <: *C*

Typechecking expressions in Java

- Base cases:
  - variables and fields
    - the type is explicitly declared
  - Expressions using `new ...()`
    - the type is the class being created
  - Type-casting
    - the type is the type forced by the cast
- For method calls, e.g., e1.m(e2)
  1. Determine the type *T1* of the receiver expression e1
  2. Determine the type *T2* of the argument expression e2
  3. Find the method declaration m in type *T1* (or supertypes), using dispatch rules
  4. The type is the return type of the method declaration identified in step 3

Subtyping Rules

- If a concrete class B extends type A
  - B must define or inherit all concrete methods declared in A
- If B overrides a method declared in supertype A
  - The argument types must be **the same** as those in A’s method
  - The result type must be a subtype of the result type from A’s method
  - Advice: Always use `@Override`
- Behavioral subtyping
  - If B overrides a method declared in A, it should conform to the specification from A
  - If `Cowboy.draw()` overrides `Circle.draw()`, somebody gets hurt!