Principles of Software Construction: Objects, Design and Concurrency

Packages and Inheritance

15-214

toad

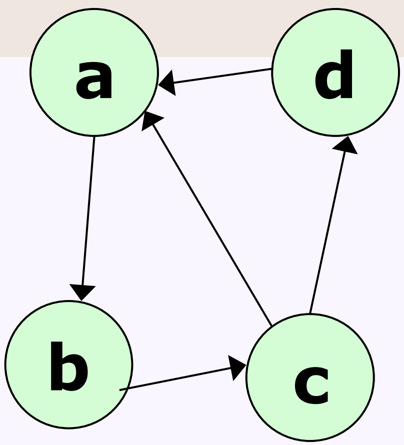
Spring 2013

Christian Kaestner **Charlie Garrod**

School of Computer Science

institute for SOFTWARE RESEARCH

Homework 1: Representing graphs



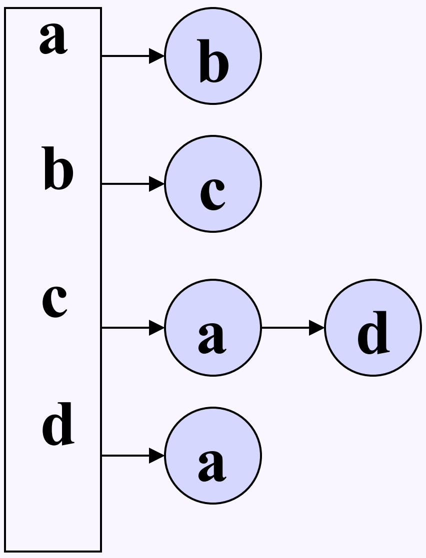
Two common representations

- *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

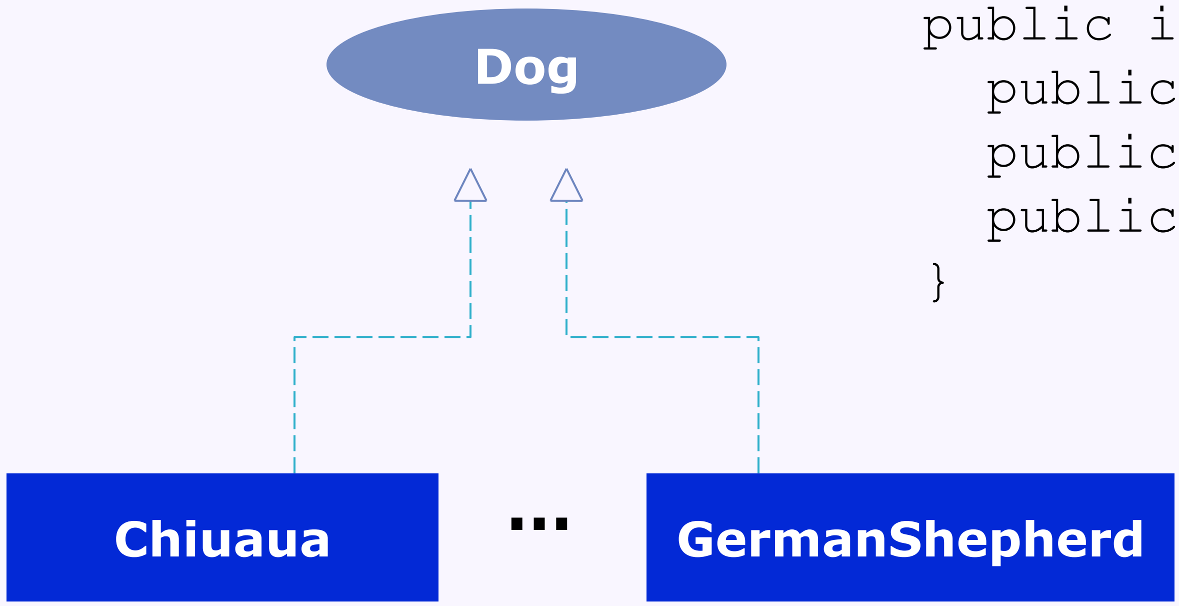


Key concepts from Thursday

Key concepts from Thursday

- Objects, Classes, and References
- Encapsulation and Visibility
- Polymorphism
  - Interfaces
  - Method Dispatch
- Object Equality

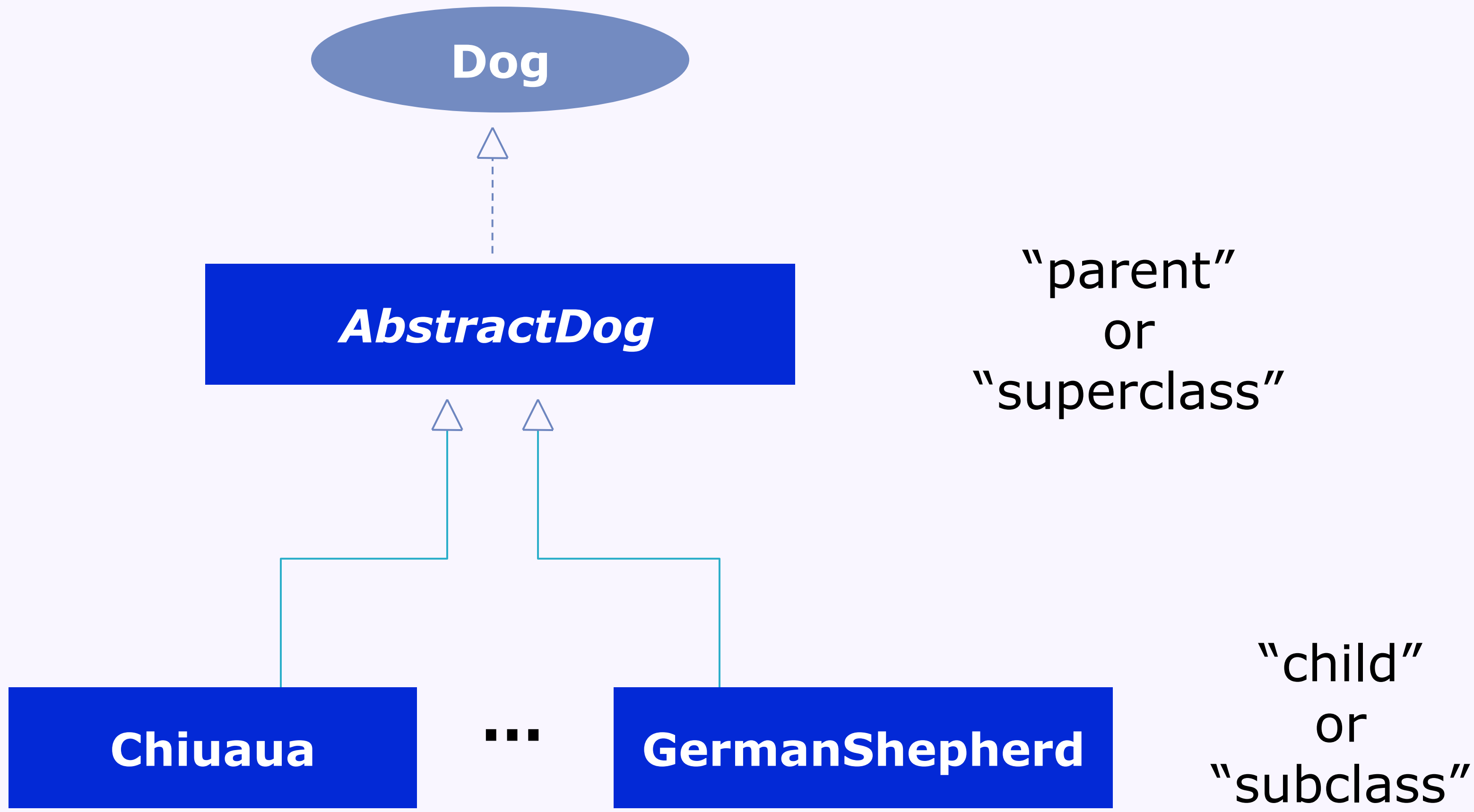
E.g., a Dog interface




```
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



institute for SOFTWARE RESEARCH

Key concepts for today

- Packages
  - Name and visibility management
  - Qualified names
- 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

Visibility of properties and methods

```
package edu.cmu.cs.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 and visibility

- Packages divide the Java namespace to organize related classes
- Visibility of names:
  - `public`: visible everywhere
  - `private`: visible only within class
  - default (no modifier): visible only within package
  - `protected`: visible within package and also to subclasses

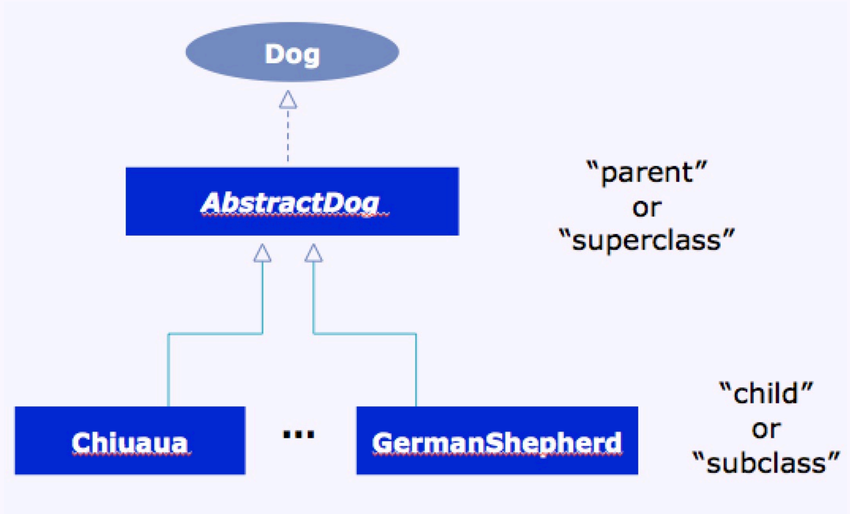
Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N

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

An introduction to inheritance

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



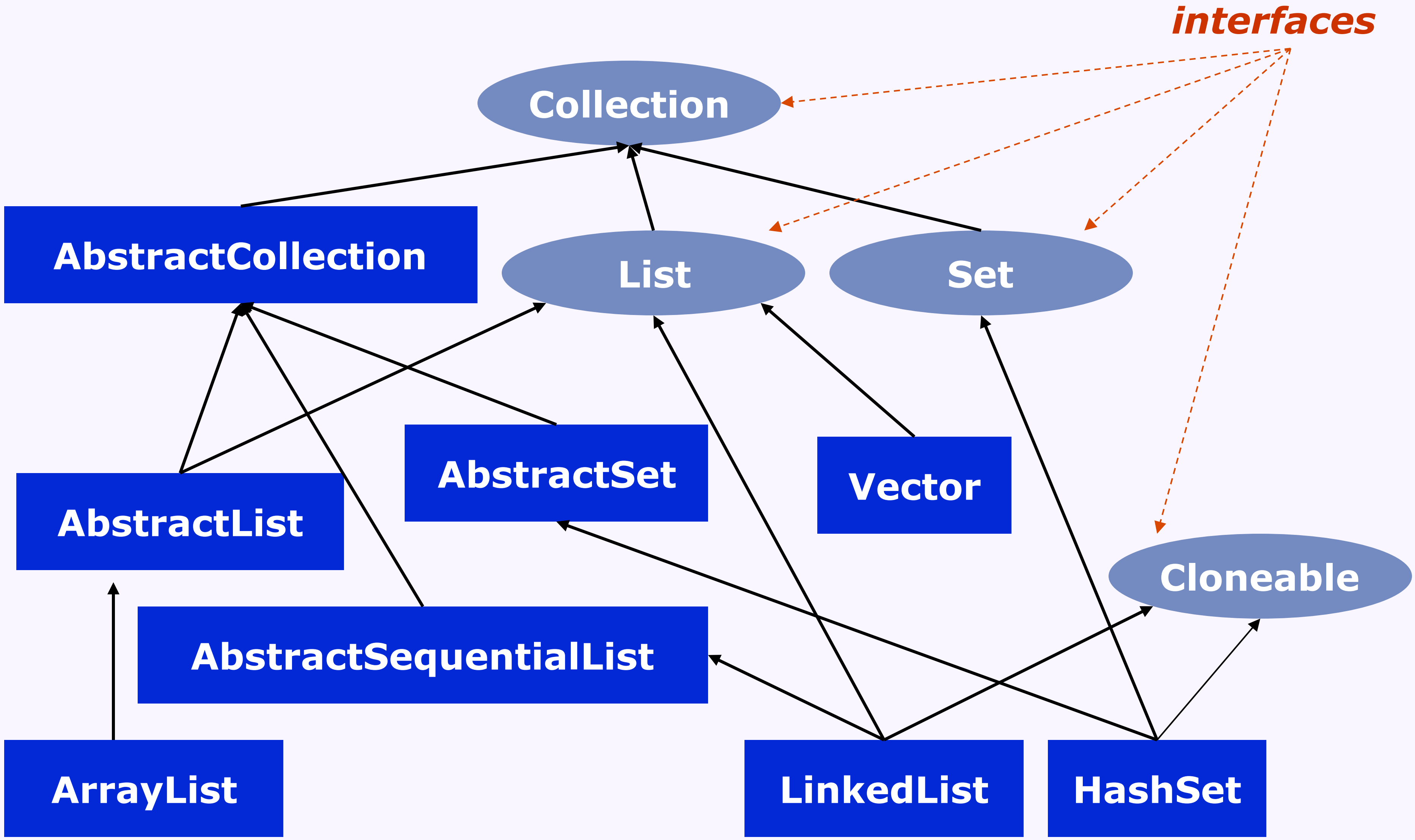
- Typical roles:
  - An interface define expectations / commitment for clients
  - An *abstract class* is a convenient hybrid between an interface and a full implementation
  - Can *override* a method definition to specialize its implementation

Inheritance: a glimpse at the hierarchy

- Examples from Java
  - Collections library [*next slide*]
  - Graphics objects
  - java.lang.Object

- Benefits and risks of inheritance
  - Reuse of code
  - Modeling flexibility
    - Specialization ←→ Subtyping
  - Multiple inheritance
    - In Java:
      - Can extend only one parent class
      - Can implement multiple interfaces

JavaCollection API (excerpt)

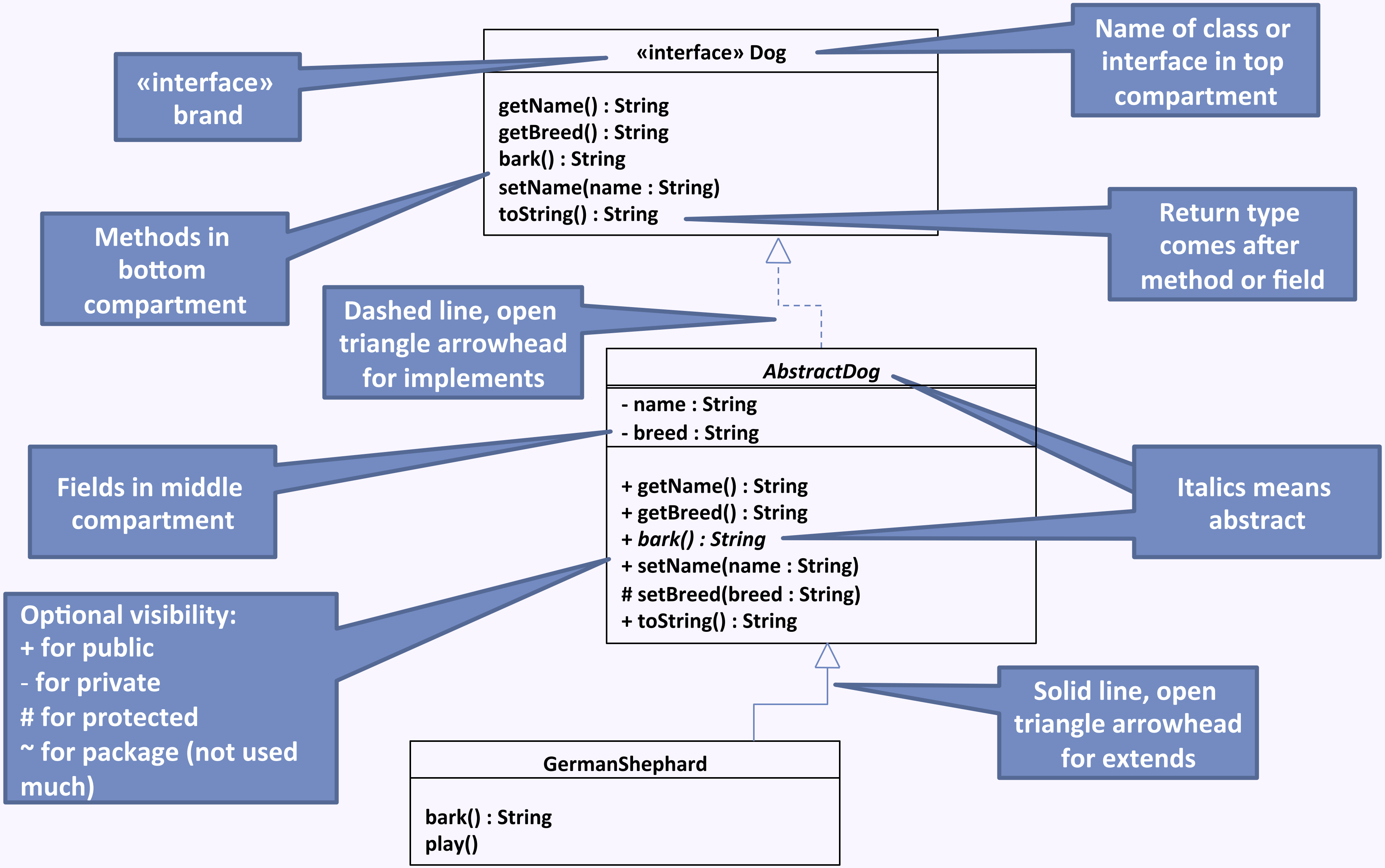


Inheritance: a glimpse at the hierarchy

- Examples from Java
  - Collections library
  - Graphics objects
  - java.lang.Object

- Benefits and risks of inheritance
  - Reuse of code
  - Modeling flexibility
    - Specialization ←→ Subtyping
  - Multiple inheritance
    - In Java:
      - Can extend only one parent class
      - Can implement multiple interfaces

Aside: UML class diagram notation

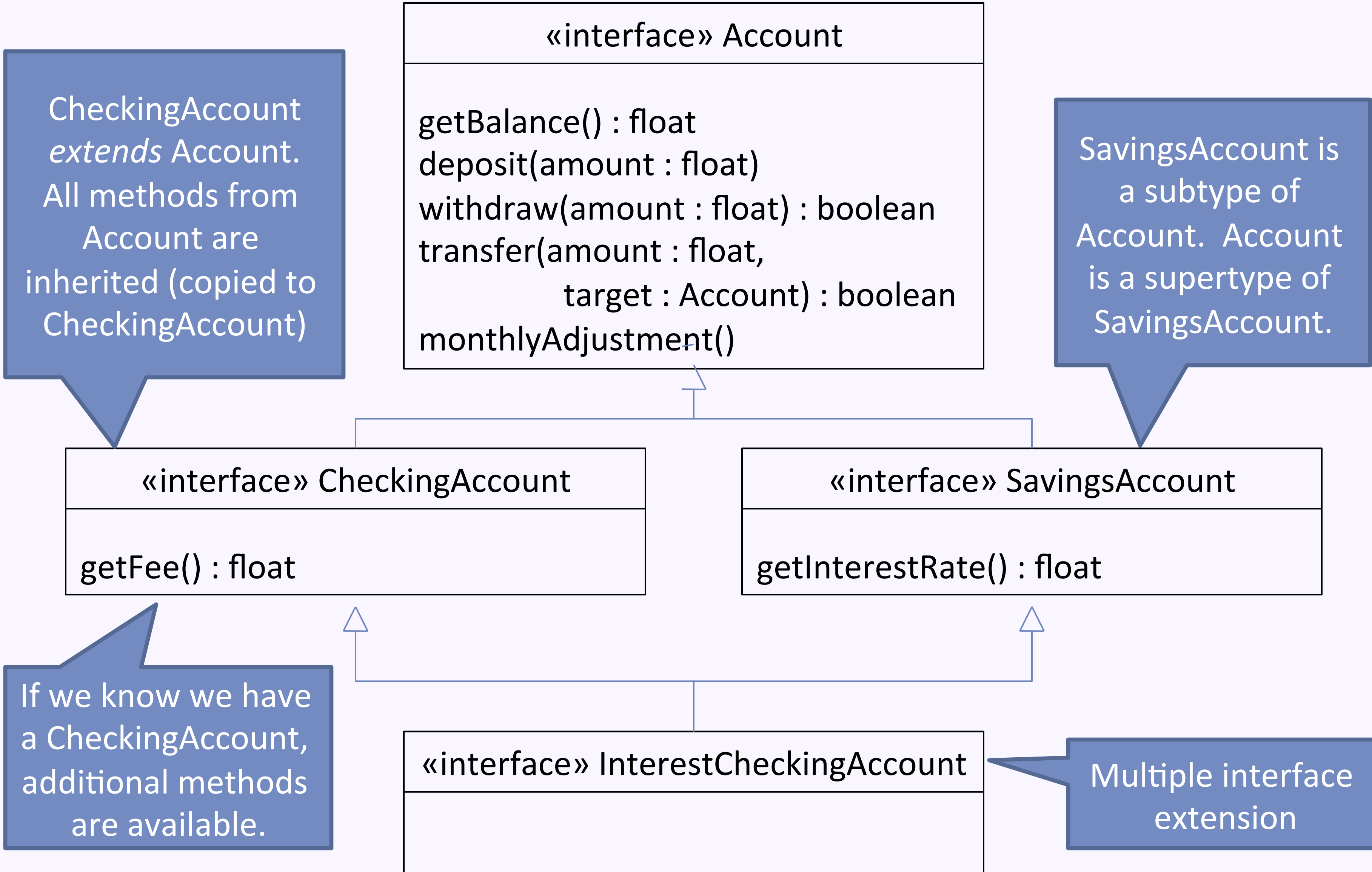


Another example: different kinds of bank accounts

«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

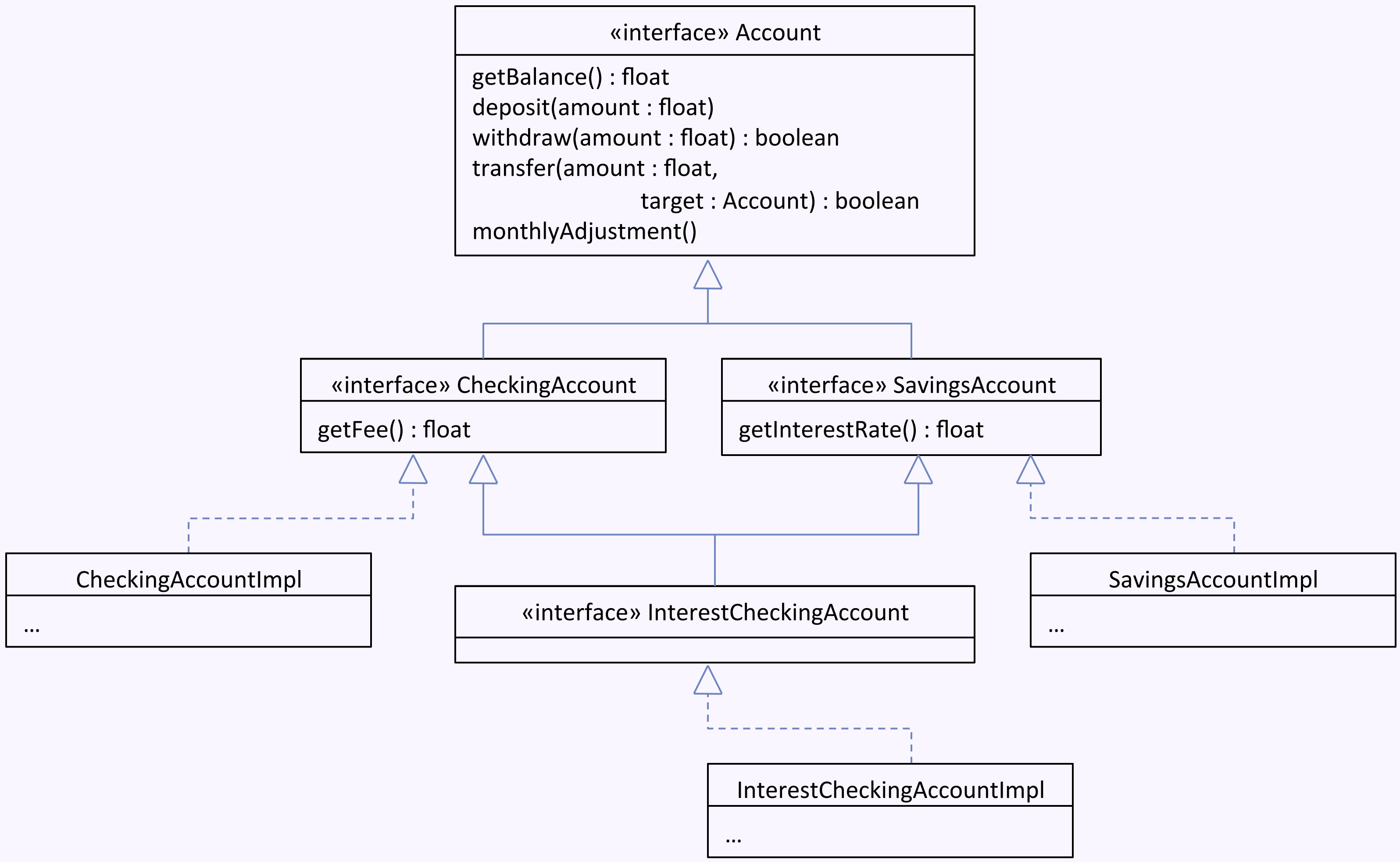
The power of object-oriented interfaces

- 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();
```

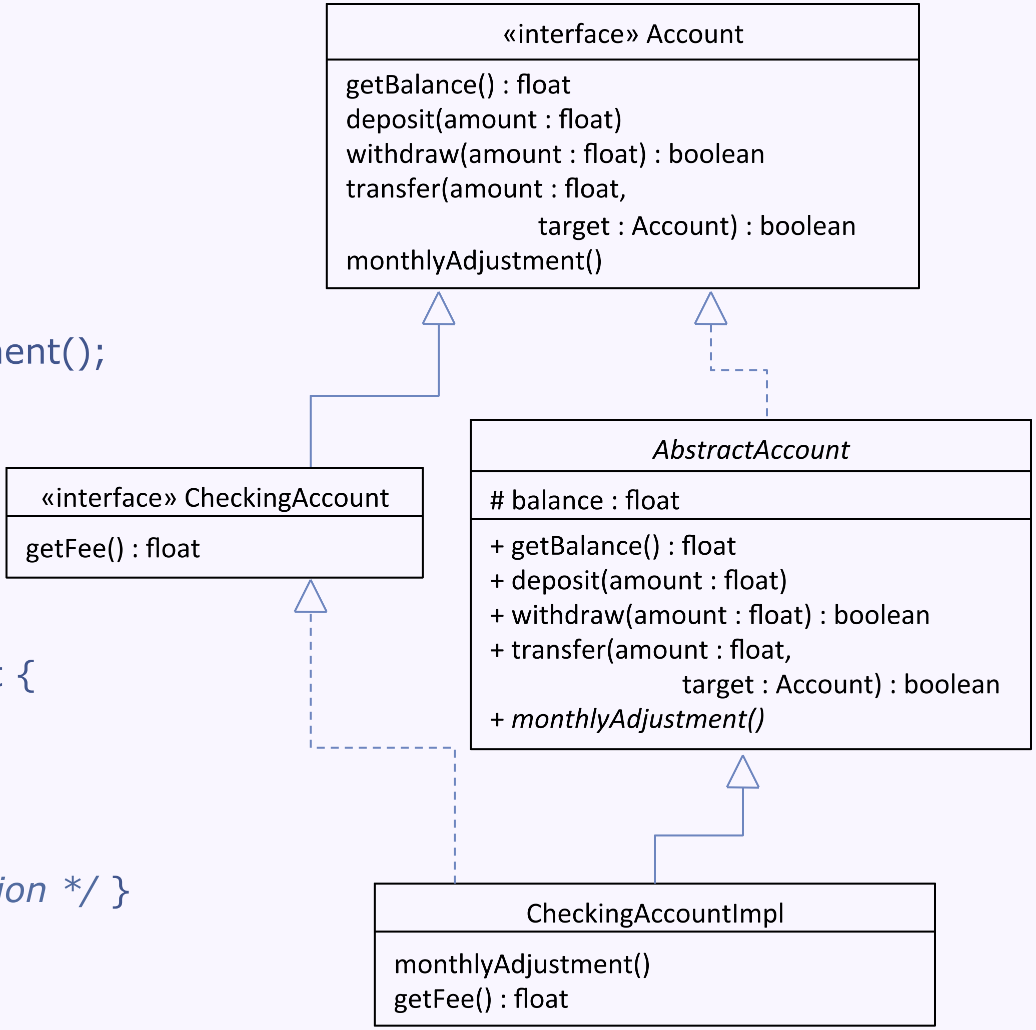
  - See the DogWalker example

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

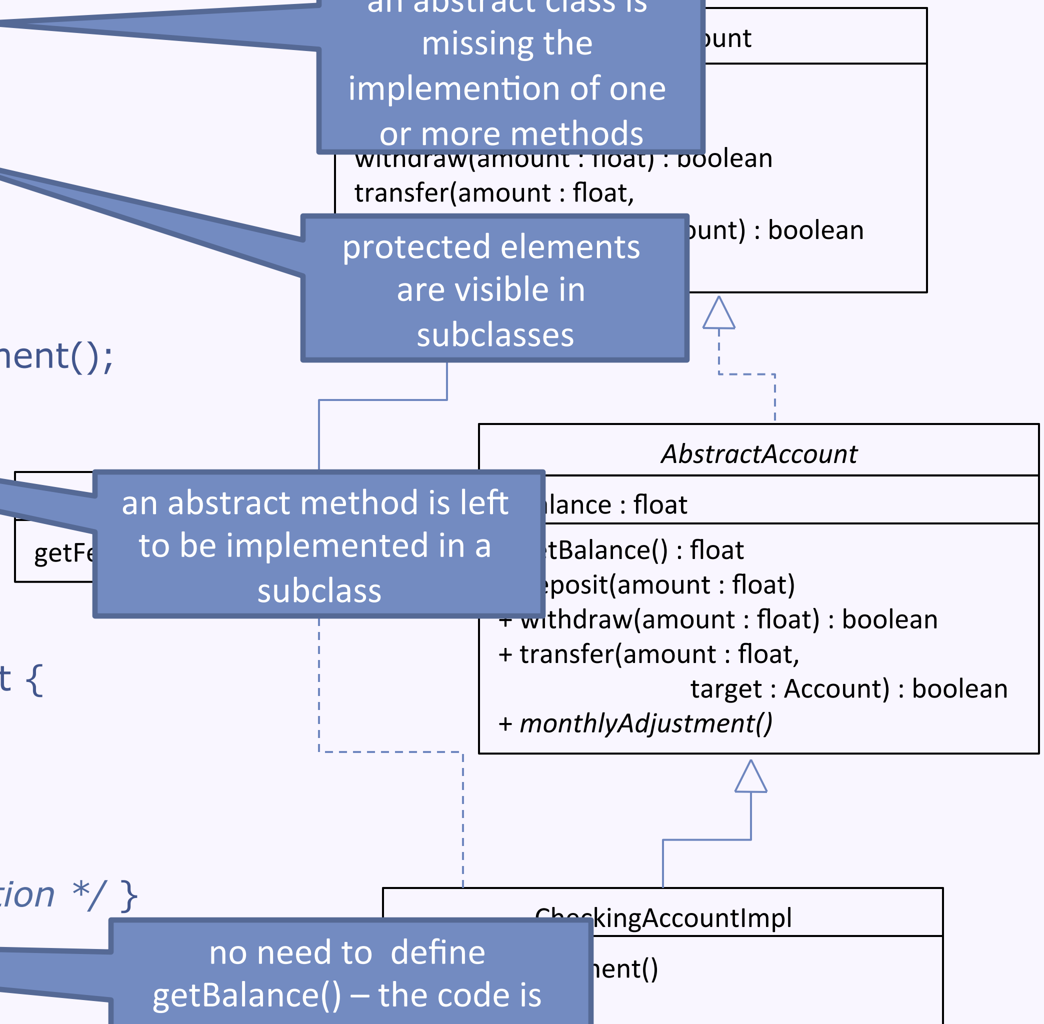
```
public abstract class AbstractAccount
            implements Account {
   protected float balance = 0.0;
   public float getBalance() {
            return balance;
   }
   abstract public void monthlyAdjustment();
   // other methods…
}

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



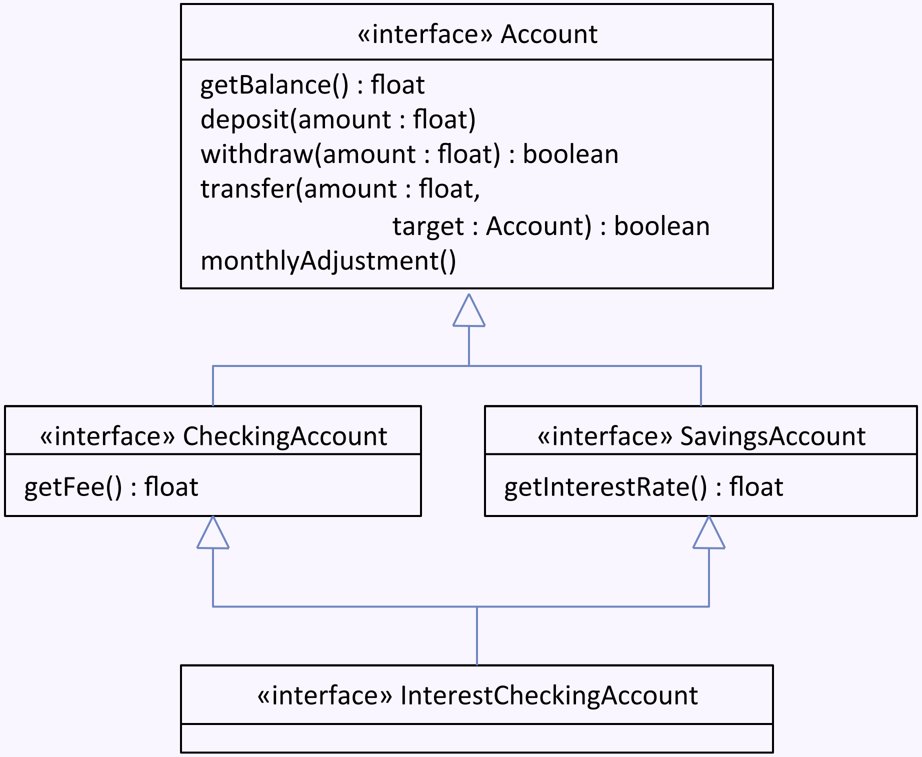
Inheritance vs. subtyping

- Inheritance
  - A class reuses code from a superclass
    - **class** A **extends** B
  - Inheritance is for **code reuse**
    - Write code once and only once
    - Code from superclass implicitly available in subclass

- Subtyping
  - A class implements a (Java) interface
    - **class** A **implements** I
  - A class implements the (implicit) interface of another class
    - **class** A **extends** B : both subtyping and inheritance
  - Subtyping is for **polymorphism**
    - Accessing objects the *same way*, but getting *different behavior*
    - Subtype is *substitutable* for supertype

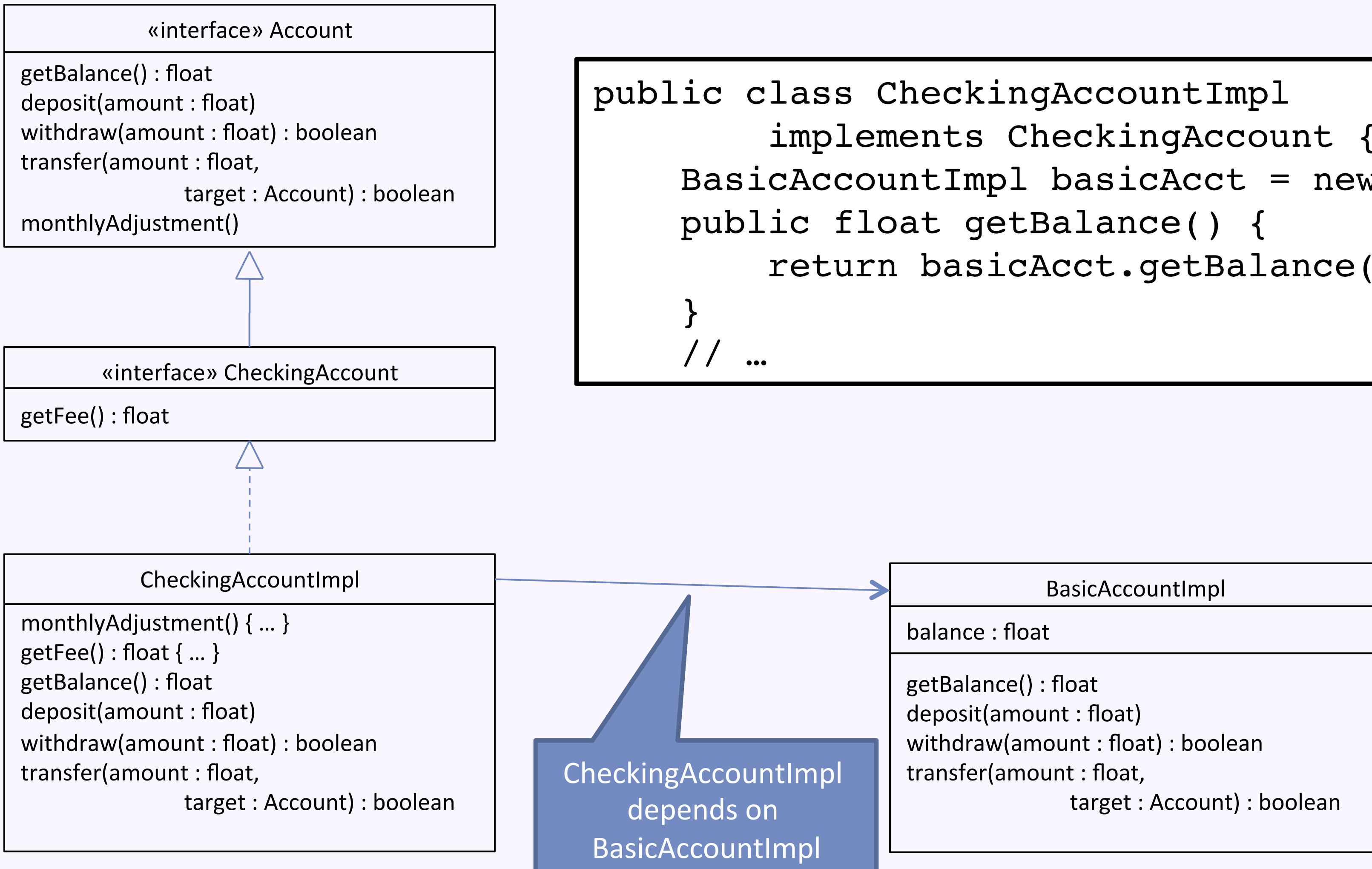
Challenge: Is inheritance necessary?

- Can we get the same amount of code reuse using only interfaces?

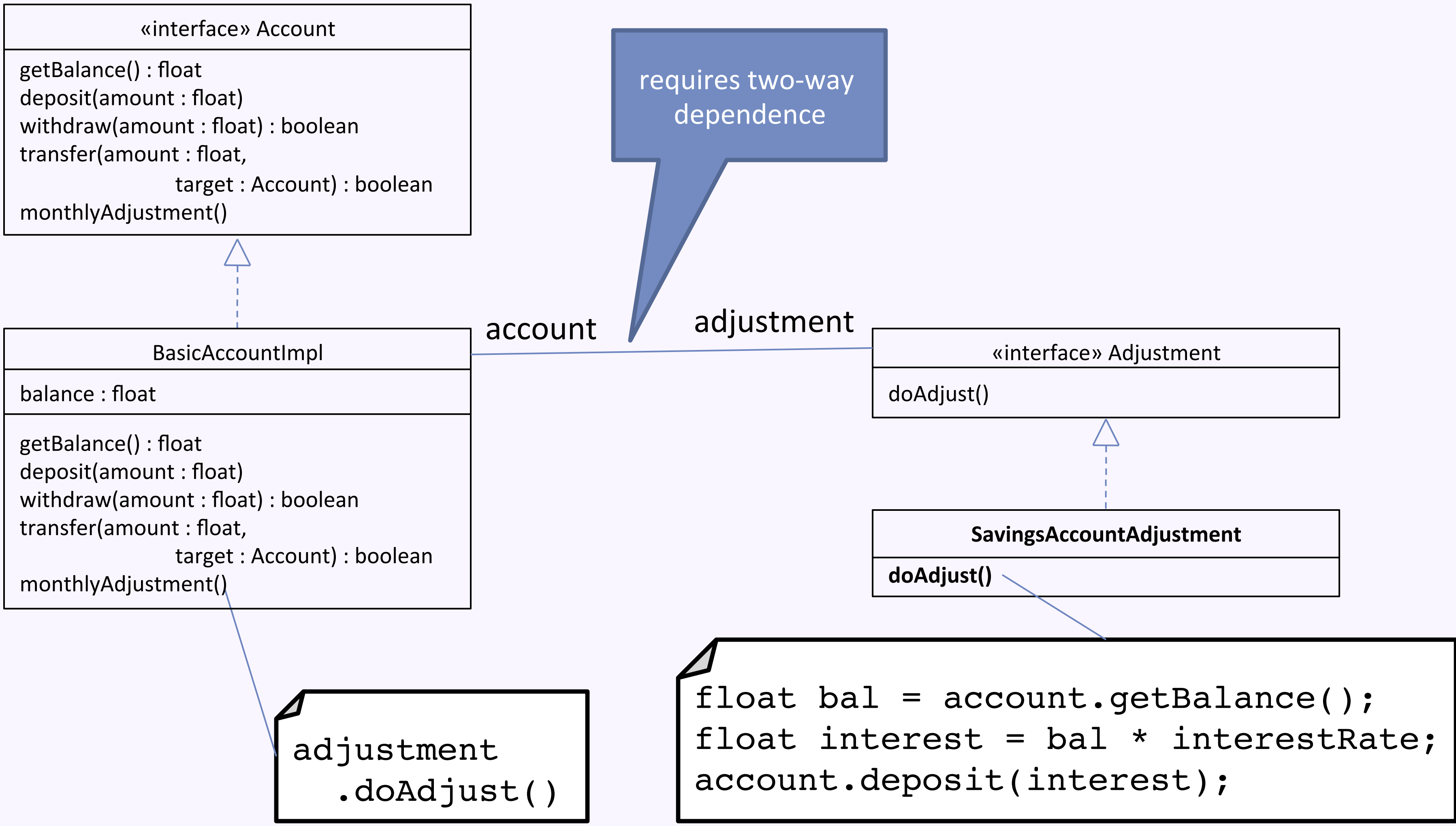


Reuse via code wrappers

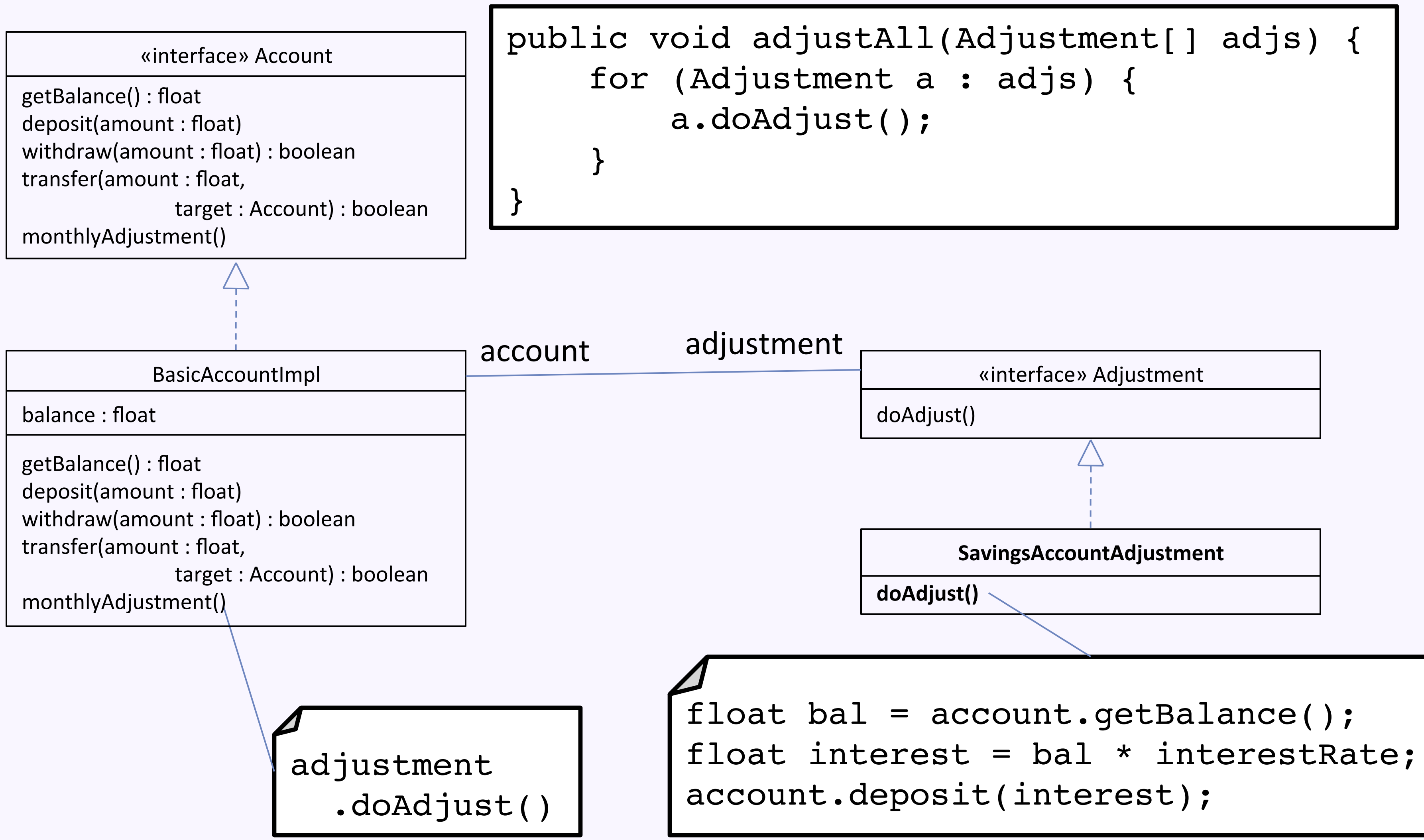
```
public class CheckingAccountImpl
        implements CheckingAccount {
    BasicAccountImpl basicAcct = new(…);
    public float getBalance() {
        return basicAcct.getBalance();
    }
    // …
```



Reuse via code wrappers, version 2: Delegation



Reuse via code wrappers, version 2: Delegation



Inheritance vs. delegation

- Delegation can be cleaner than inheritance
  - Reused code in a separate object
  - Interfaces between objects
- Inheritance has less boilerplate code
  - No forwarding functions, recursive dependencies

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

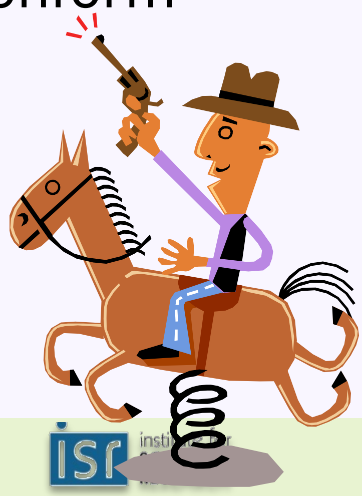
- Value flow: assignments, passing parameters
  - e.g., `Foo f = expression;`
  - Determine the type T_{source} of the source expression
  - Determine the type T_{dest} of the destination variable `f`
  - Check that T_{source} is a subtype of T_{dest}
- Subtype relation *A* <: *B*
  - *A* <: B if *A* 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
- 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!



Method dispatch, revisited

e.g.: `x.foo(apple, 42)`

- Step 1 (compile time): determine which class to look in
  - Here, the type of `x`
- Step 2 (compile time): determine the method signature to be executed
  - Find all accessible, applicable methods
  - Select the most specific method
    - `m1` is more specific than `m2` if each argument of `m1` is a subtype of the corresponding argument of `m2`

Method dispatch, revisited

e.g.: `x.foo(apple, 42)`

- Step 3 (run time): Determine the run-time class of the receiver
- Step 4 (run time): Locate the method to invoke
  - Starting at the run-time class, look for a method with the same signature found in step 2
    - If it is found in the run-time class, invoke it.
    - Otherwise, continue the search in the superclass of the run-time class and etc.
- I claim: this procedure will always find a method to invoke

Method dispatch practice

```
public class GenericAnimal {
   public String getNoise() { return "Noise"; }
}
```

```
public class Bird extends GenericAnimal {
   public String getNoise() { return "Chirp"; }
}
```

```
public class Cat extends GenericAnimal {
   public String getNoise() { return "Meow"; }
}
```

```
public class GenericDog extends GenericAnimal {
   // nothing special to hear here
}
```

```
public class Ewokian extends GenericDog {
   public String getNoise() { return "Oonga!"; }
}
```

Method dispatch practice, part A

```
public class GenericAnimal {
   public String getNoise() { return "Noise"; }
}
```

```
public class Bird extends GenericAnimal {
   public String getNoise() { return "Chirp"; }
}
```

```
public class Cat extends GenericAnimal {
   public String getNoise() { return "Meow"; }
}
```

```
public class GenericDog extends GenericAnimal {
   // nothing special to hear here
}
```

```
public class Ewokian extends GenericDog {
   public String getNoise() { return "Oonga!"; }
}
```

What is printed by:

```
GenericAnimal A = new GenericAnimal();
System.out.print(A.getNoise());
```

Method dispatch practice, part B-1

```
public class GenericAnimal {
   public String getNoise() { return "Noise"; }
}
```

```
public class Bird extends GenericAnimal {
   public String getNoise() { return "Chirp"; }
}
```

```
public class Cat extends GenericAnimal {
   public String getNoise() { return "Meow"; }
}
```

```
public class GenericDog extends GenericAnimal {
   // nothing special to hear here
}
```

```
public class Ewokian extends GenericDog {
   public String getNoise() { return "Oonga!"; }
}
```

What is printed by:

```
Bird B = new Bird();
System.out.print(B.getNoise());
```

Method dispatch practice, part B-2 (on paper!)

```
public class GenericAnimal {
    public String getNoise() { return "Noise"; }
}
```

```
public class Bird extends GenericAnimal {
    public String getNoise() { return "Chirp"; }
}
```

```
public class Cat extends GenericAnimal {
    public String getNoise() { return "Meow"; }
}
```

```
public class GenericDog extends GenericAnimal {
    // nothing special to hear here
}
```

```
public class Ewokian extends GenericDog {
    public String getNoise() { return "Oonga!"; }
}
```

What is printed by:

```
GenericAnimal B = new Bird();
System.out.print(B.getNoise());
```

Method dispatch practice, part C

```
public class GenericAnimal {
    public String getNoise() { return "Noise"; }
}
```

```
public class Bird extends GenericAnimal {
    public String getNoise() { return "Chirp"; }
}
```

```
public class Cat extends GenericAnimal {
    public String getNoise() { return "Meow"; }
}
```

```
public class GenericDog extends GenericAnimal {
    // nothing special to hear here
}
```

```
public class Ewokian extends GenericDog {
    public String getNoise() { return "Oonga!"; }
}
```

What is printed by:

```
GenericAnimal C = new Cat();
System.out.print(C.getNoise());
```

Method dispatch practice, part D

```
public class GenericAnimal {
   public String getNoise() { return "Noise"; }
}
```

```
public class Bird extends GenericAnimal {
   public String getNoise() { return "Chirp"; }
}
```

```
public class Cat extends GenericAnimal {
   public String getNoise() { return "Meow"; }
}
```

```
public class GenericDog extends GenericAnimal {
   // nothing special to hear here
}
```

```
public class Ewokian extends GenericDog {
   public String getNoise() { return "Oonga!"; }
}
```

What is printed by:

```
GenericAnimal D = new GenericDog();
System.out.print(D.getNoise());
```

Method dispatch practice, part E-1

```
public class GenericAnimal {
   public String getNoise() { return “Noise”; }
}
```

```
public class Bird extends GenericAnimal {
   public String getNoise() { return “Chirp”; }
}
```

```
public class Cat extends GenericAnimal {
   public String getNoise() { return “Meow”; }
}
```

```
public class GenericDog extends GenericAnimal {
   // nothing special to hear here
}
```

```
public class Ewokian extends GenericDog {
   public String getNoise() { return “Oonga!”; }
}
```

What is printed by:

```
GenericAnimal E = new Ewokian();
System.out.print(E.getNoise());
```

Method dispatch practice, part E-2

```
public class GenericAnimal {
   public String getNoise() { return "Noise"; }
}
```

```
public class Bird extends GenericAnimal {
   public String getNoise() { return "Chirp"; }
}
```

```
public class Cat extends GenericAnimal {
   public String getNoise() { return "Meow"; }
}
```

```
public class GenericDog extends GenericAnimal {
   // nothing special to hear here
}
```

```
public class Ewokian extends GenericDog {
   public String getNoise() { return "Oonga!"; }
}
```

What is printed by:

```
Ewokian E = new Ewokian();
GenericAnimal F = E;
System.out.print(F.getNoise());
```

The `java.lang.Object`

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

```
String    toString()
boolean   equals(Object obj)
int       hashCode()
Object    clone()
```

Method dispatch parctice, part F

```
public class Object {
    String toString()             { … }
    boolean      equals(Object obj) { … }
    int          hashCode()         { … }
    Object clone()                { … }
}

public class Point {
    private final int x, y;
    // …
    String  toString {
        return String.valueOf(x) + " " +
               String.valueOf(y);
    }
    boolean equals(Point p) {
      return x == p.x && y == p.y;
    }
    int     hashCode() {
      return toString().hashCode();
    }
```