Principles of Software Construction: Objects, Design, and Concurrency

An Introduction to Object-oriented Programming, Continued. Modules and Inheritance

Spring 2014

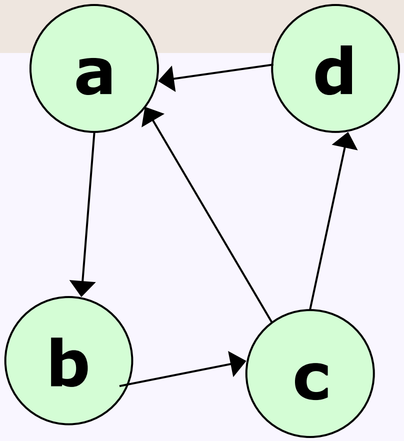
Charlie Garrod Christian Kästner

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

- Homework 0 due tonight, 11:59 p.m.
  - Access, turn in via your course Git repository
  - Note: your repository is shared
- Homework 1 due next Tuesday

Homework 1: Representing graphs



Two common representations

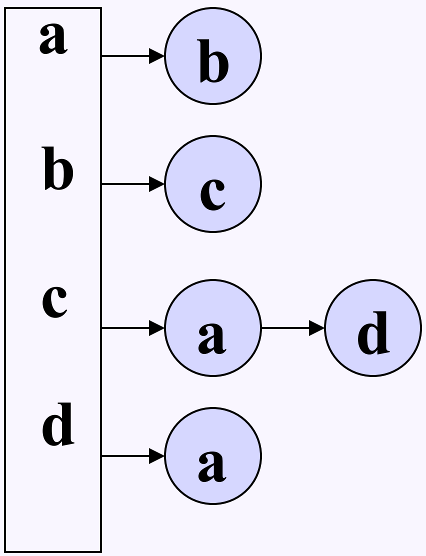
- *Adjacency matrix*
- *Adjacency list*

Adjacency matrix

target

source	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

Static type vs. dynamic type

- Static type: the declared, compile-time type of the object
- Dynamic type: the instantiated, run-time type of the object in memory

static type

```
Point p = null;
if (Math.random() < 0.5) {
    p = new CartesianPoint(0, 0);
} else {
    p = new PolarPoint(0, 0);
}
…
```

dynamic type

Object identity vs. object equality

- Every object is created with a unique identity

```
Point a = new PolarPoint(1,1); // new object
Point b = a;                   // same reference, same object
Point c = new PolarPoint(1,1); // new object
Point d = new PolarPoint(1,.9999999); // new object
```

  - Comparing object identity compares references

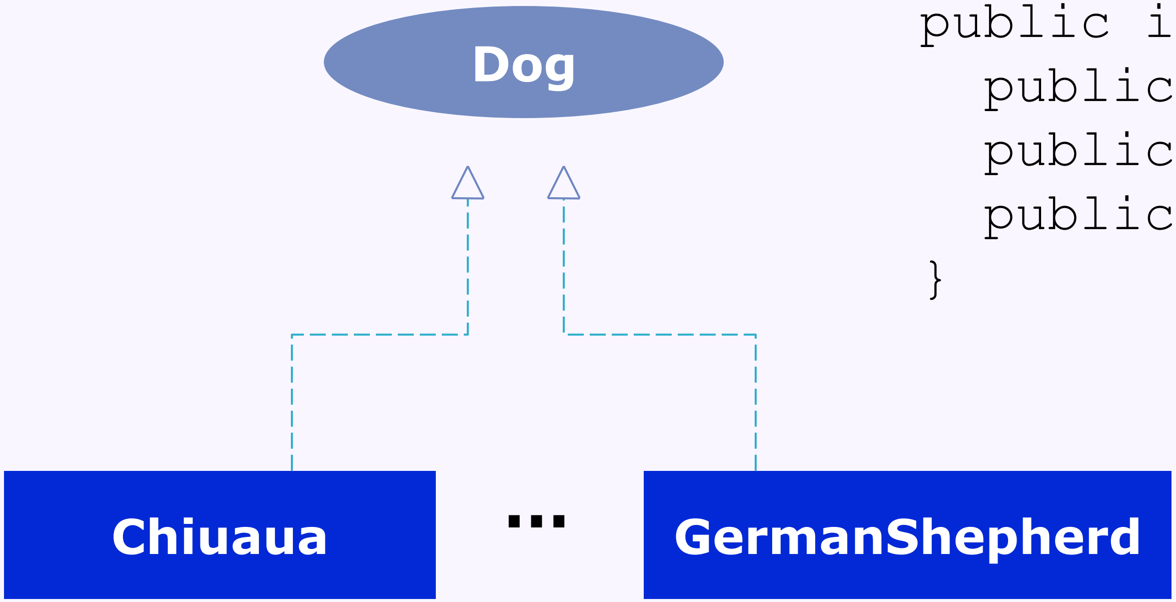
    `a == b` but `a != c`

- Object equality is domain specific
  - When are two points equal?

    `a.equals(b)?   c.equals(a)?   d.equals(a)?`

institute for SOFTWARE RESEARCH

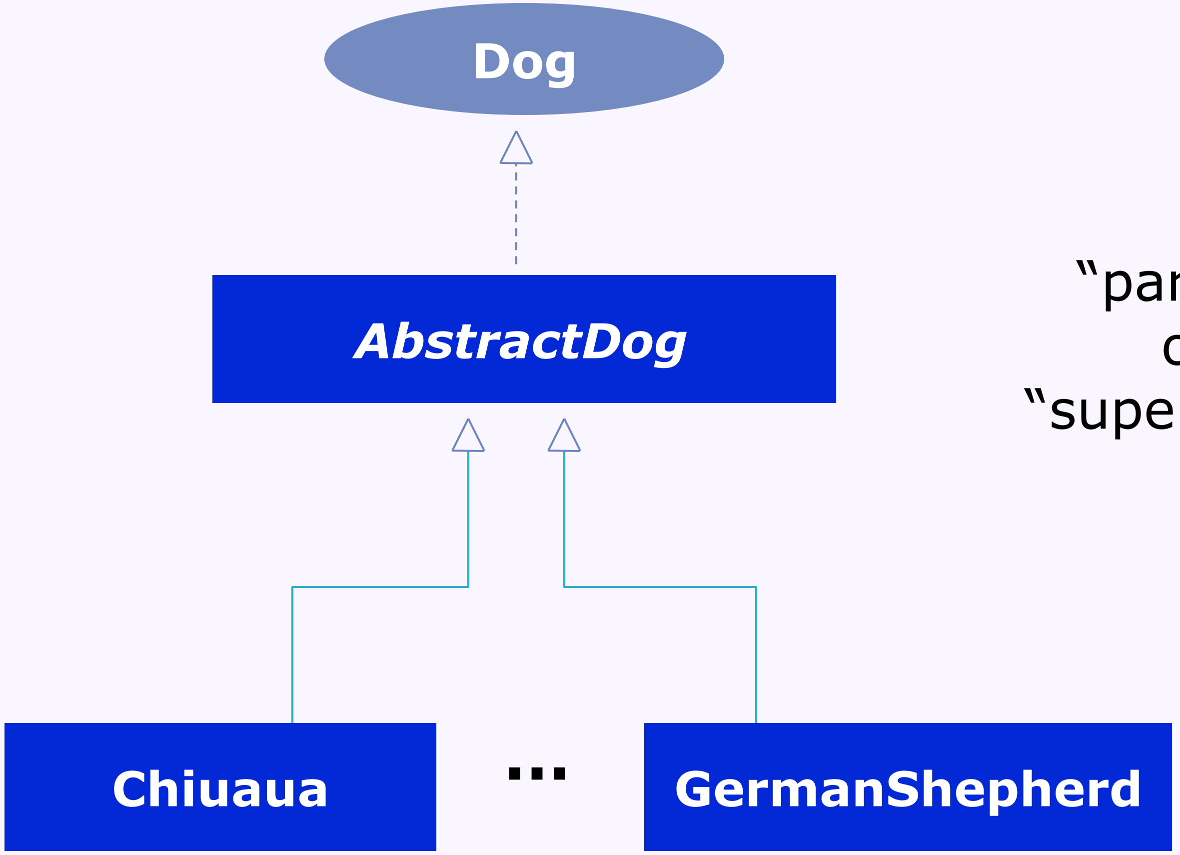
Polymorphism e.g., a Dog interface




```
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"

Today:

- Modular programming
  - Java packages
- Inheritance and polymorphism
  - For maximal code re-use
  - Diagrams to show the relationships between classes
  - Inheritance and its alternatives
  - Java details related to inheritance

Modular programming

- A software *module* is a separate, independent component that encapsulates some aspect of the underlying program
- Language support for software modules:
  - Separate groups of program source files
  - Independent, internal name spaces
  - Separate compilation, linking
  - Modular run-time features

Java packages

- Packages divide a global Java namespace to organize related classes

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

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 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

Visibility of Java 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

Encapsulation design principles

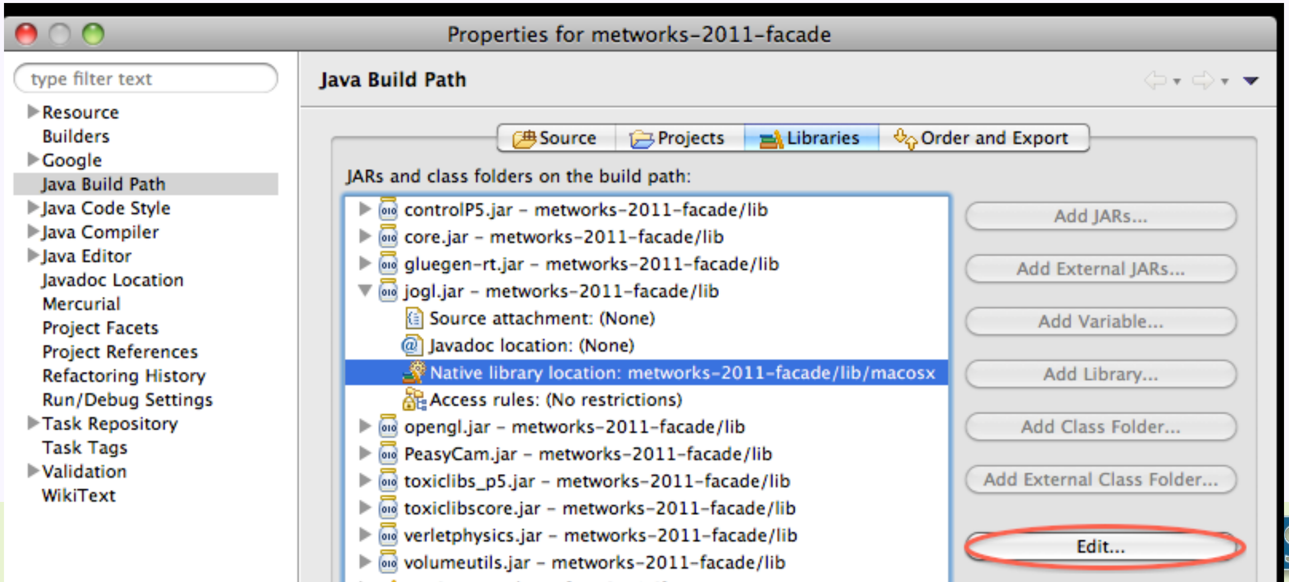
- Restrict accessibility as much as possible
  - Make data and methods private unless you have a reason to make it more visible
  - Use interfaces to abstract from implementations

"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

Java class loading

- Java class path controls run-time access to program components
  - .class files
  - .jar files
    - essentially just a .zip file to bundle classes
- Can add classes to class path when starting the Java Virtual Machine:

```
$ java -cp /home/xanadu:lib/parser.jar:. Main
```

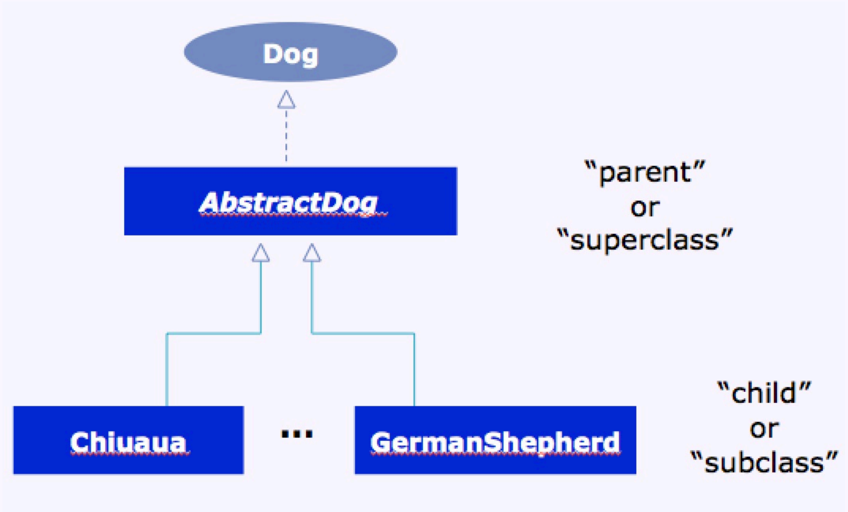


Today:

- Modular programming
  - Java packages
- Inheritance and polymorphism
  - For maximal code re-use
  - Diagrams to show the relationships between classes
  - Inheritance and its alternatives
  - Java details related to inheritance

An introduction to inheritance

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

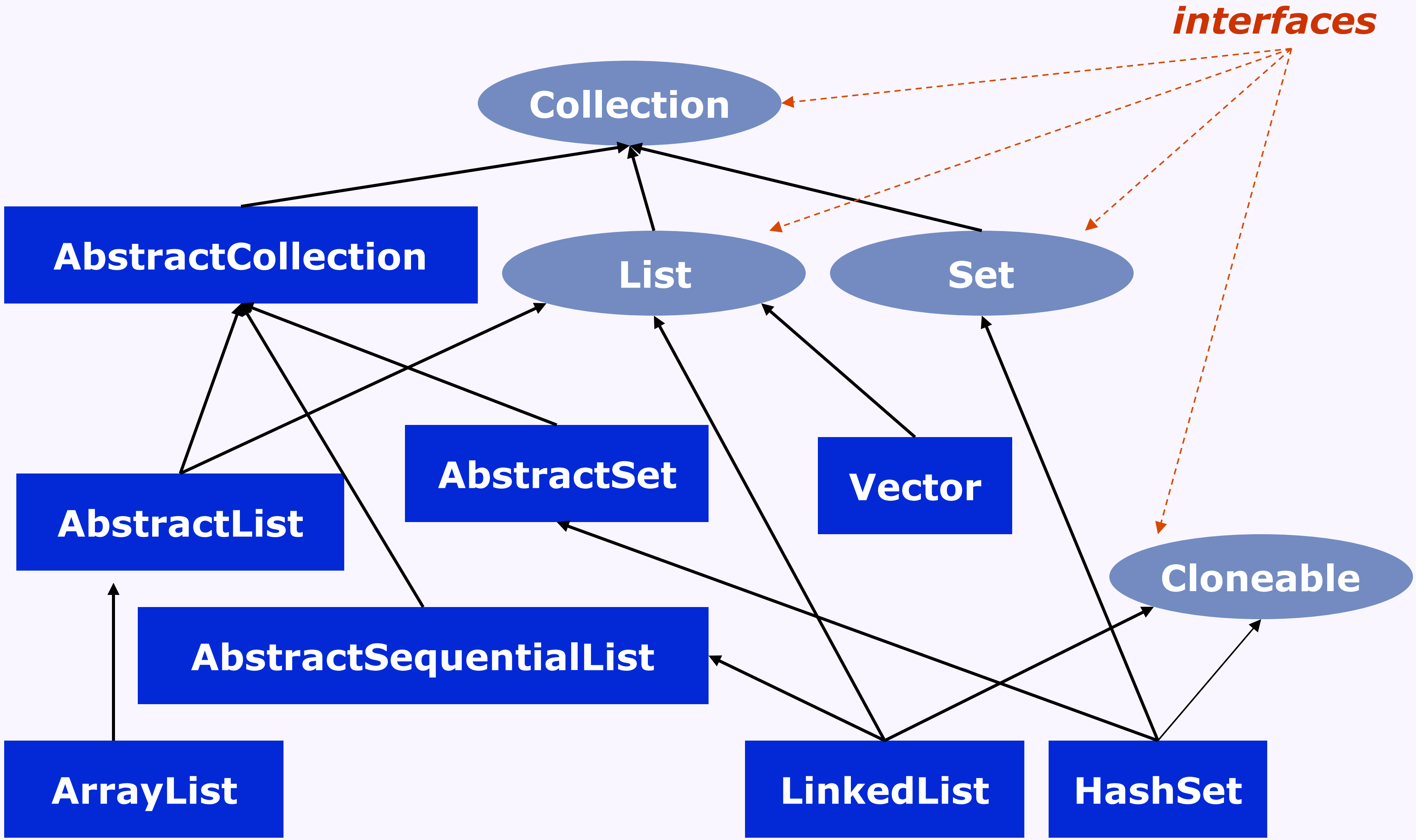


- Typical roles:
  - An interface define expectations / commitment for clients
  - An *abstract class* is a convenient hybrid between an interface and a full implementation
  - Subclass *overrides* a method definition to specialize its implementation

Inheritance: a glimpse at the hierarchy

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

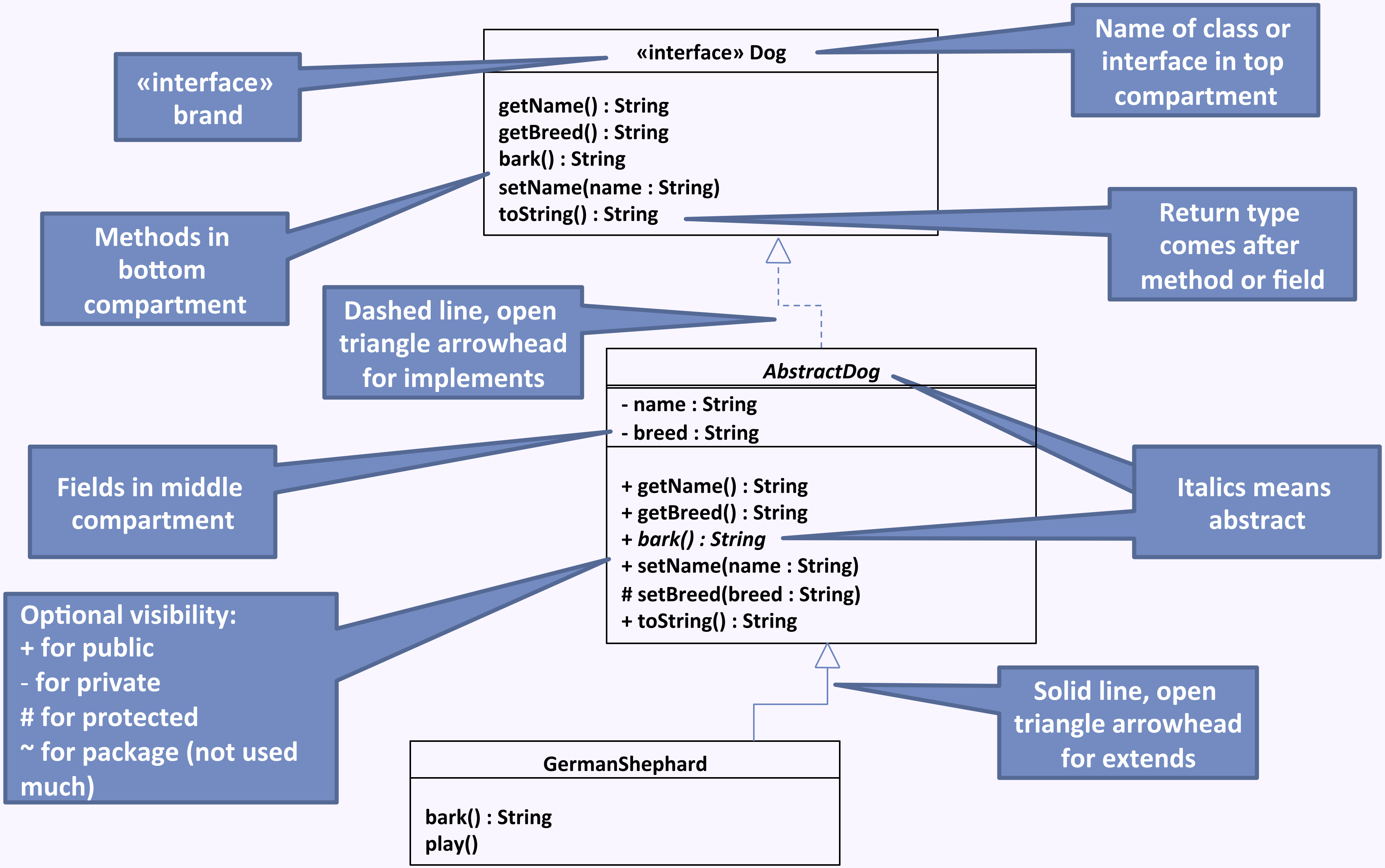
JavaCollection API (excerpt)



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

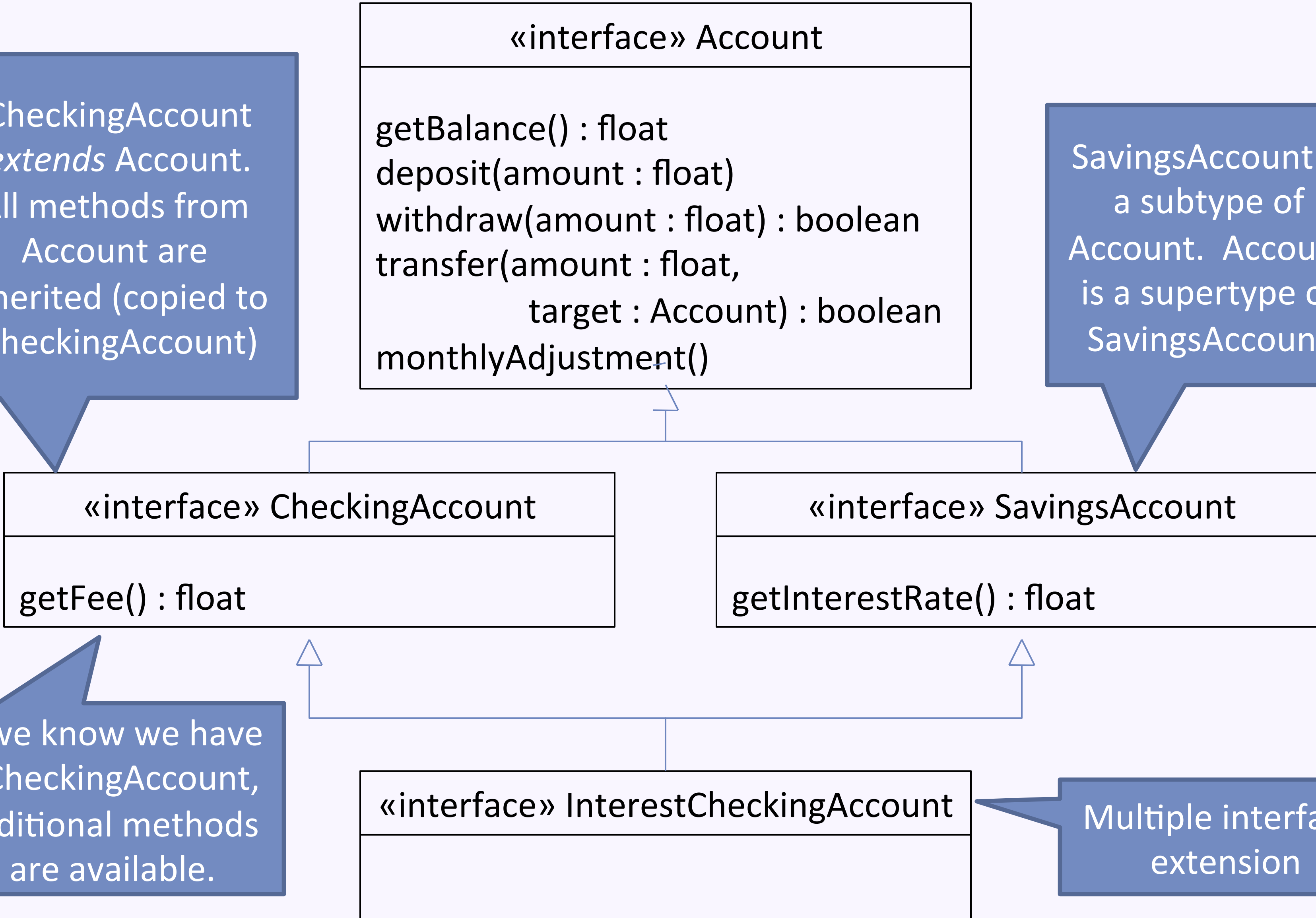


Another example: different kinds of bank accounts

«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



CheckingAccount *extends* Account. All methods from Account are inherited (copied to CheckingAccount)

SavingsAccount is a subtype of Account. Account is a supertype of SavingsAccount.

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

Multiple interface extension

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

«interface» InterestCheckingAccount

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

Multiple interface extension

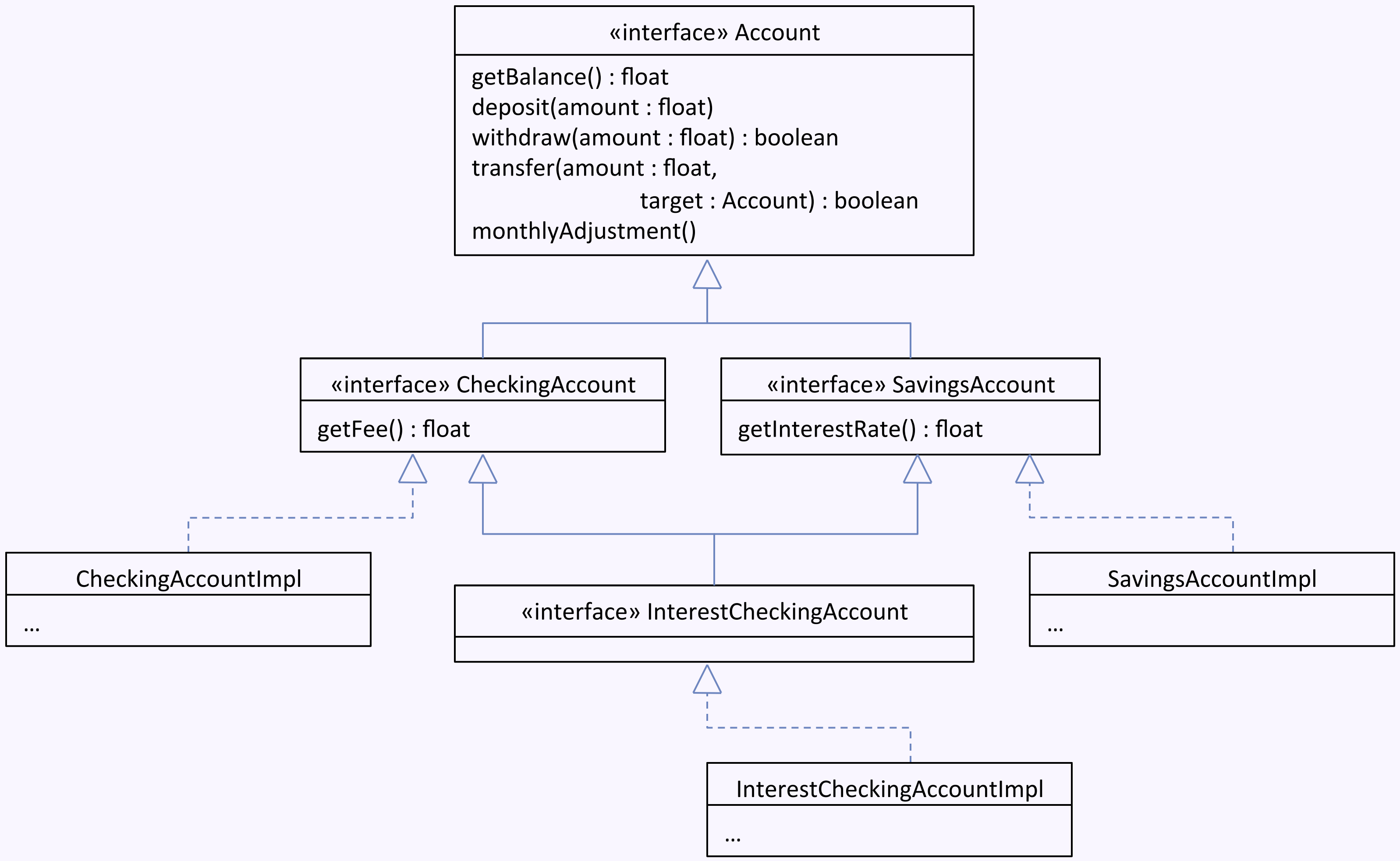
The power of object-oriented interfaces

- 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();
```

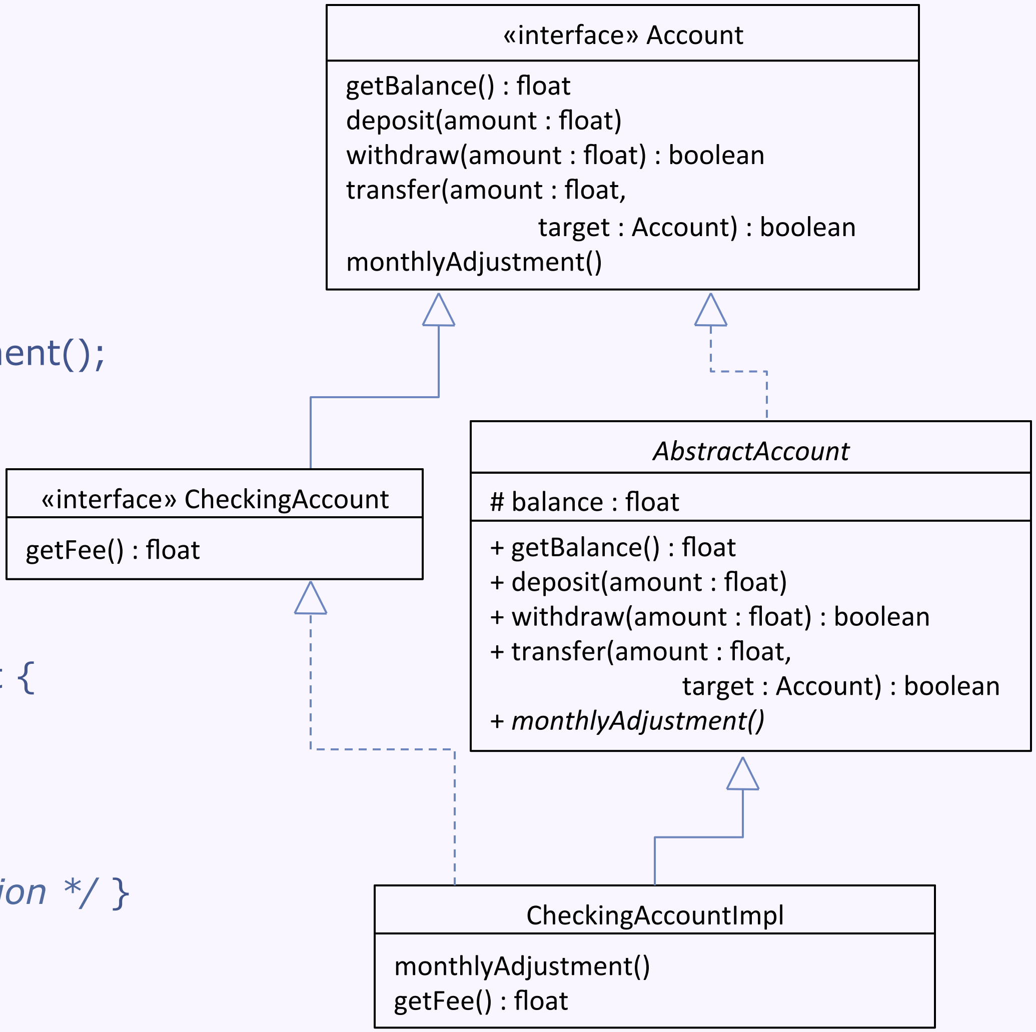
  - See the DogWalker example

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

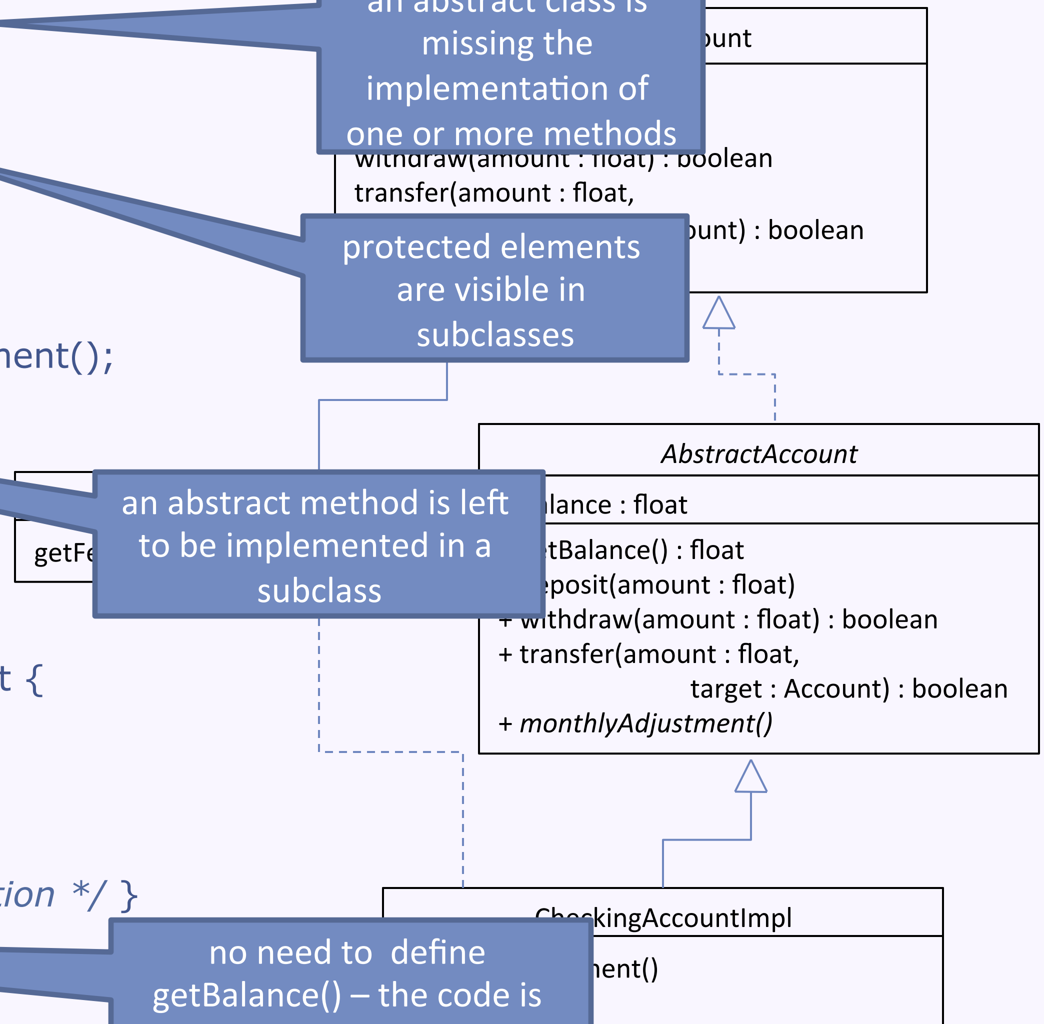
```
public abstract class AbstractAccount
            implements Account {
   protected float balance = 0.0;
   public float getBalance() {
            return balance;
   }
   abstract public void monthlyAdjustment();
   // other methods…
}

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

an abstract class is missing the 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



Inheritance and subtyping

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

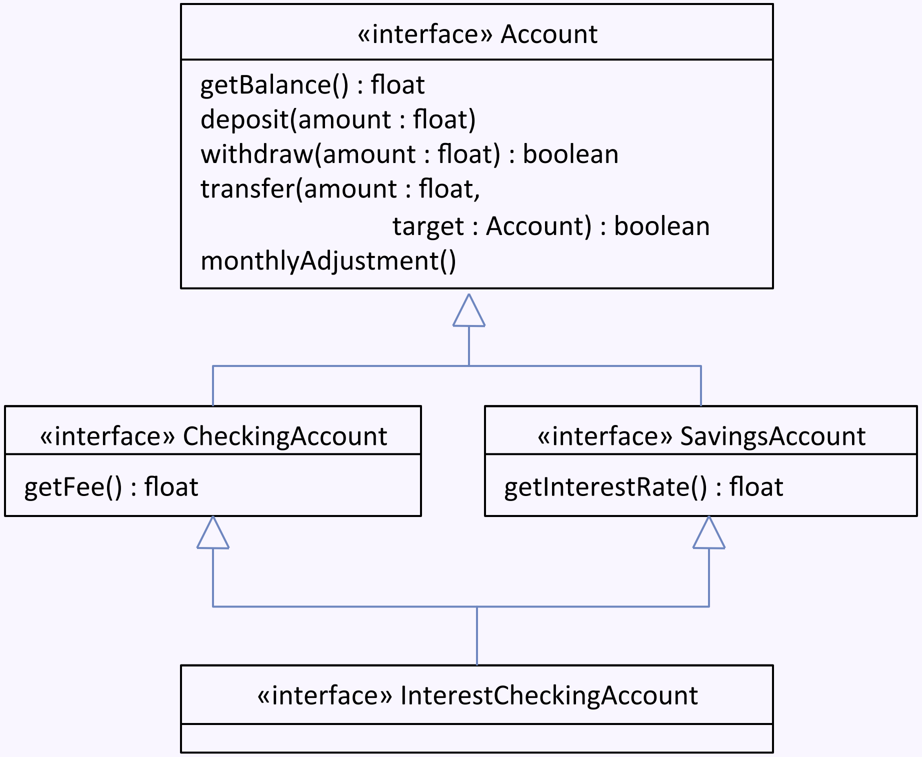
```
class A extends B
```

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

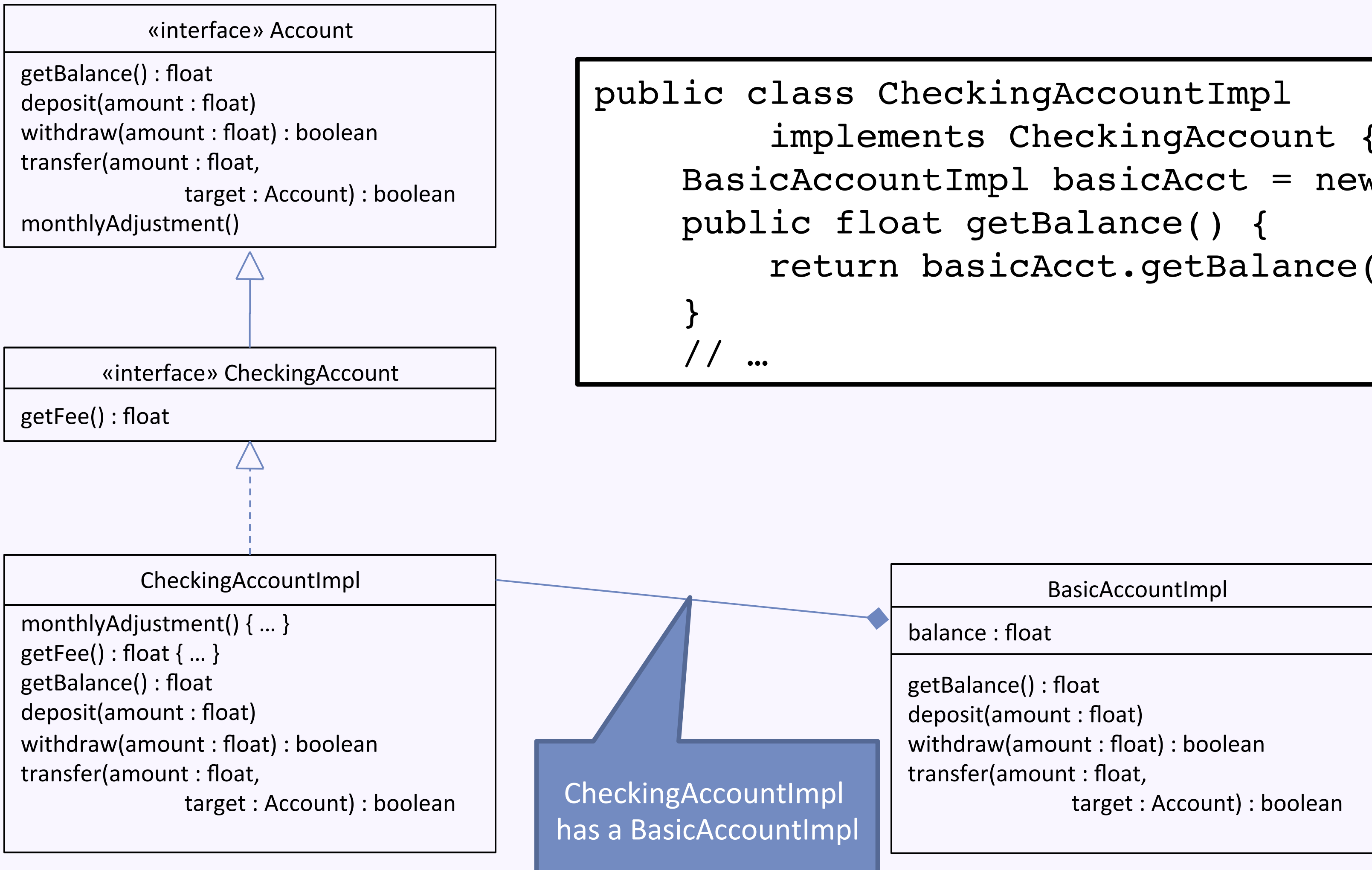
```
class A implements I
class A extends B
```

Challenge: Is inheritance necessary?

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



Reuse via *composition* and *delegation*




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

Java details: 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

Java details: constructors 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

Java 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?

Recall: 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
- Advice: avoid down-casting types if possible

Recall: `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();
} else if (acct instanceof SavingsAccount) {
    savingsAcct = (SavingsAccount) acct;
    adj = savingsAcct.getInterest();
}
```

- Advice: avoid `instanceof` if possible

Avoiding `instanceof` with the Template Method pattern

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

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

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

Avoiding `instanceof` with the Template Method pattern

```
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();
}
```

```
Account acct = …;
float adj = acct.getMonthlyAdjustment();
```