Principles of Software Construction: Objects, Design and Concurrency

Polymorphism, part 2

15-214

toad

Fall 2012

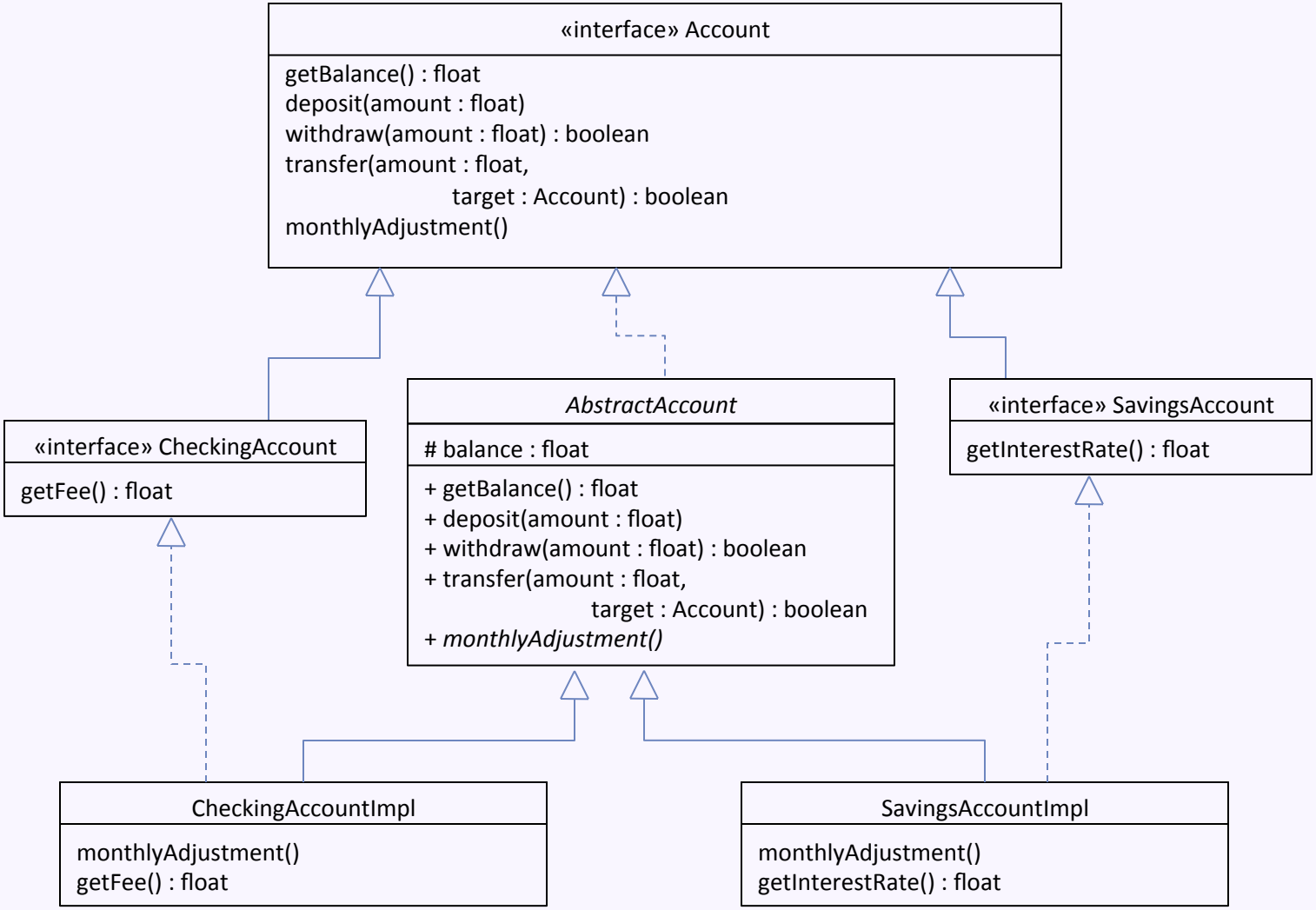
Jonathan Aldrich **Charlie Garrod**

School of Computer Science
institute for SOFTWARE RESEARCH

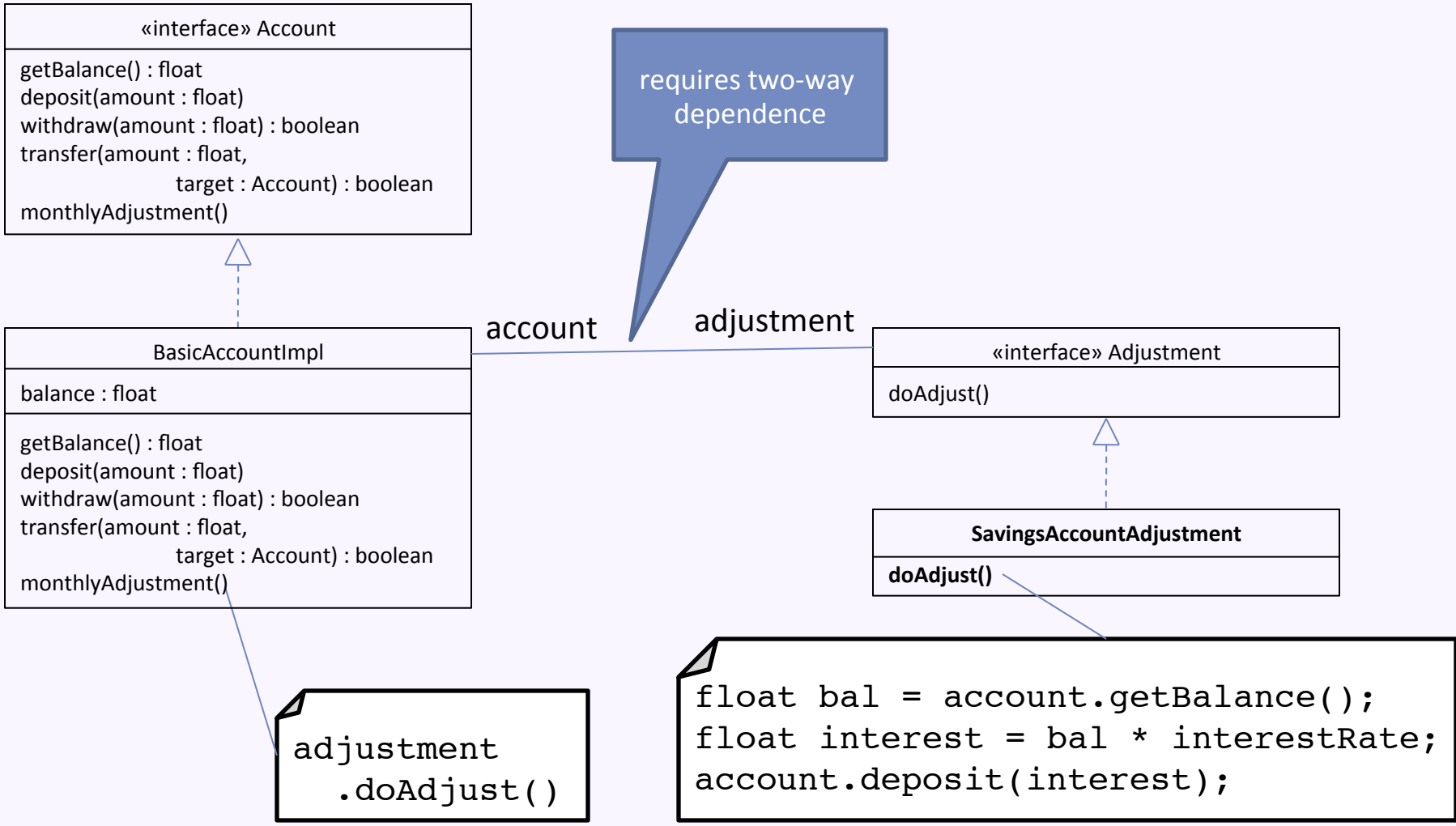
Key concepts from Thursday

- Packages
- Inheritance and polymorphism
  - Subtyping

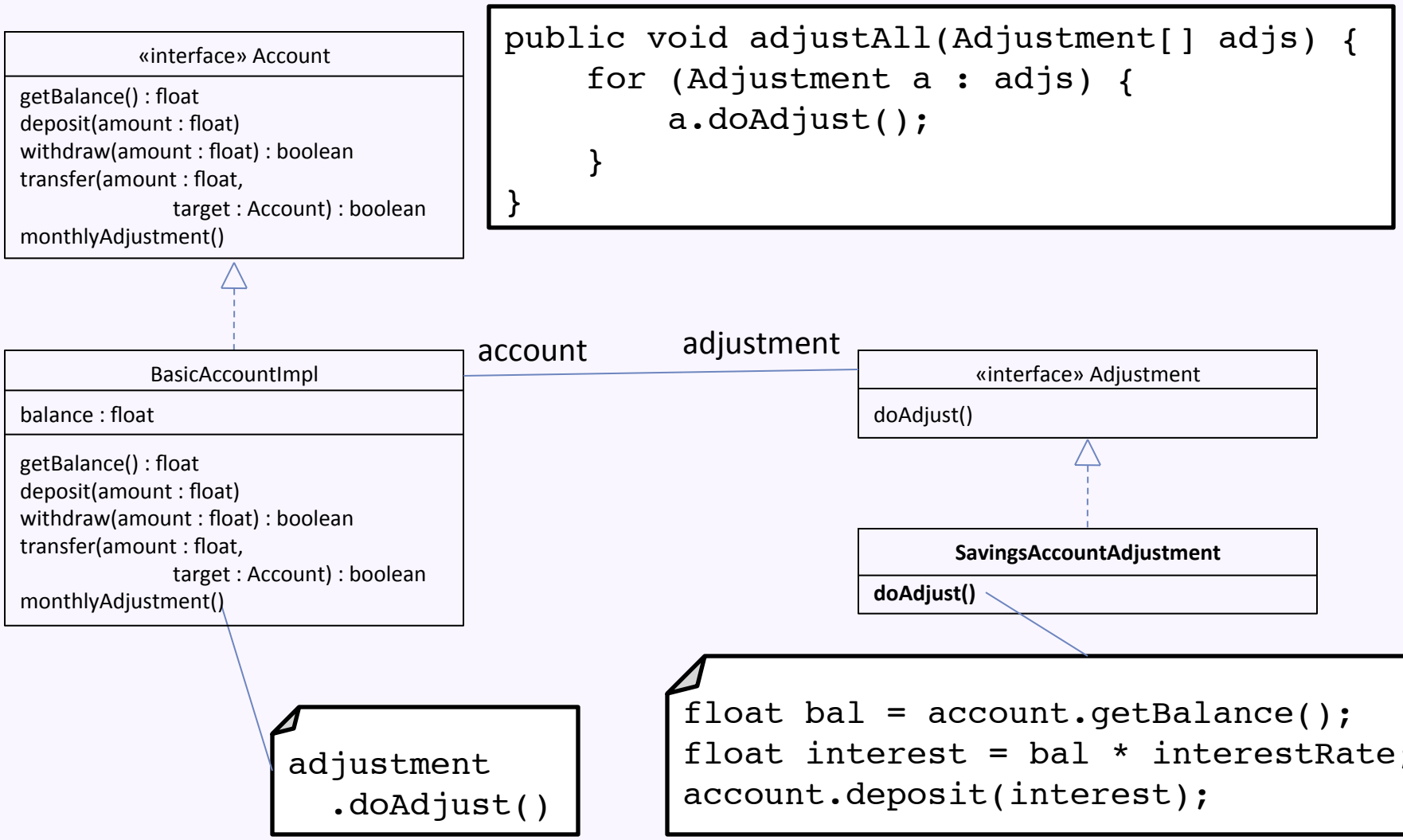
Subtyping


«interface» Account
getBalance() : float
deposit(amount : float)
withdraw(amount : float) : boolean
transfer(amount : float,
target : Account) : boolean
monthlyAdjustment()
«interface» CheckingAccount
getFee() : float
AbstractAccount
balance : float
+ getBalance() : float
+ deposit(amount : float)
+ withdraw(amount : float) : boolean
+ transfer(amount : float,
target : Account) : boolean
+ monthlyAdjustment()
«interface» SavingsAccount
getInterestRate() : float
CheckingAccountImpl
monthlyAdjustment()
getFee() : float
SavingsAccountImpl
monthlyAdjustment()
getInterestRate() : float

Reuse via code wrappers, version 2: Delegation



Reuse via code wrappers, version 2: Delegation



Key concepts for today

- More inheritance and polymorphism
  - Method dispatch, revisited
  - `super`, `this`, `final`, `instanceof`, type casting
- Types and typechecking
- The `java.lang.Object`
- Type polymorphism

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
  - 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 right name and argument types that are identical to those in the method found statically (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
- 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());
```


toad

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());
```


toad

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());
```


toad

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

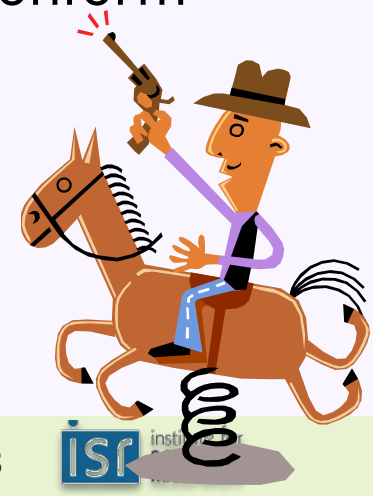
- Value flow: assignments, passing parameters
  - e.g., `Foo f = expression;`
  - Determine the type T_{source} of the source expression
  - Determine the type T_{dest} of the destination variable `f`
  - Check that T_{source} is a subtype of T_{dest}
- Subtype relation *A* <: *B*
  - *A* <: B if *A*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



Key concepts for today

- More inheritance and polymorphism
  - Method dispatch, revisited
  - `super`, `this`, `final`, `instanceof`, type casting
- Types and typechecking
- The `java.lang.Object`
- Type polymorphism

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()`

Overriding `java.lang.Object`'s `.equals`

- The default `.equals`:

```
public class Object {
    public boolean equals(Object obj) {
        return this == obj;
    }
}
```

- An aside: Do you like:

```
public class CheckingAccountImpl
        implements CheckingAccount {
    @Override
    public boolean equals(Object obj) {
        return false;
    }
}
```

The `.equals(Object obj)` contract

- An equivalence relation
  - Reflexive: ∀`x` `x.equals(x)`
  - Symmetric: ∀`x,y` `x.equals(y)` if and only if `y.equals(x)`
  - Transitive: ∀`x,y,z` `x.equals(y)` and `y.equals(z)` implies `x.equals(z)`
- Consistent
  - Invoking `x.equals(y)` repeatedly returns the same value unless `x` or `y` is modified
- `x.equals(null)` is always false