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

```
float bal = account.getBalance();
float interest = bal * interestRate;
account.deposit(interest);
```

### Account Interface
- `getBalance() : float`
- `deposit(amount : float)`
- `withdraw(amount : float) : boolean`
- `transfer(amount : float, target : Account) : boolean`
- `monthlyAdjustment()`

### BasicAccountImpl
- `balance : float`
- `getBalance() : float`
- `deposit(amount : float)`
- `withdraw(amount : float) : boolean`
- `transfer(amount : float, target : Account) : boolean`
- `monthlyAdjustment()`

### Adjustment Interface
- `doAdjust()`

### SavingsAccountAdjustment
- `doAdjust()`

### Requires two-way dependence
- The adjustment requires two-way dependence.
public void adjustAll(Adjustment[] adjs) {
    for (Adjustment a : adjs) {
        a.doAdjust();
    }
}

float bal = account.getBalance();
float interest = bal * interestRate;
account.deposit(interest);
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

- 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
    - \( m_1 \) is more specific than \( m_2 \) if each argument of \( m_1 \) is a subtype of the corresponding argument of \( m_2 \)

Example: \( x.\text{foo}(\text{apple}, 42) \)
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

```java
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

```java
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!"; }
}

public class GenericAnimal A = new GenericAnimal();
System.out.print(A.getNoise());
```
Method dispatch practice, part B-1

```java
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!"; }
}

public class Bird B = new Bird();
System.out.println(B.getNoise());
```

What is printed by:
Method dispatch practice, part B-2 (on paper!)

```java
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!"; }
}

public class GenericAnimal B = new Bird();
System.out.println(B.getNoise());
```
Method dispatch practice, part C

```java
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!"; }
}

GenericAnimal C = new Cat();
System.out.print(C.getNoise());
```
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.println(D.getNoise());
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!"; }
}

public class GenericAnimal E = new Ewokian();
System.out.println(E.getNoise());
Method dispatch practice, part E-2

```java
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!"; }
}

Ewokian E = new Ewokian();
GenericAnimal F = E;
System.out.println(F.getNoise());
```
Extended re-use with `super`

```java
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 AbstractAccount 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

```java
public class CheckingAccountImpl extends AbstractAccount 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 another constructor in this same class

Invokes a constructor of the superclass. Must be the first statement of the constructor.
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.,
    ```java
    float pi = 3.14;
    int indianaPi = (int) pi;
    ```

• Useful if you know you have a more specific subtype:
  - e.g.,
    ```java
    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., \( \text{Foo } f = \text{expression}; \)
  - Determine the type \( T_{\text{source}} \) of the source expression
  - Determine the type \( T_{\text{dest}} \) of the destination variable \( f \)
  - Check that \( T_{\text{source}} \) is a subtype of \( T_{\text{dest}} \)

• **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 $T_1$ of the receiver expression $e_1$
  2. Determine the type $T_2$ of the argument expression $e_2$
  3. Find the method declaration $m$ in type $T_1$ (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`:**
  ```java
class Object {
    public boolean equals(Object obj) {
        return this == obj;
    }
}
```

- **An aside: Do you like:**
  ```java
class CheckingAccountImpl {
    public boolean equals(Object obj) {
        return false;
    }
}
```
The \texttt{.equals(Object obj)} contract

- An equivalence relation
  - Reflexive: \( \forall x \ x\texttt{.equals}(x) \)
  - Symmetric: \( \forall x, y \ x\texttt{.equals}(y) \) if and only if \( y\texttt{.equals}(x) \)
  - Transitive: \( \forall x, y, z \ x\texttt{.equals}(y) \) and \( y\texttt{.equals}(z) \) implies \( x\texttt{.equals}(z) \)

- Consistent
  - Invoking \( x\texttt{.equals}(y) \) repeatedly returns the same value unless \( x \) or \( y \) is modified

- \( x\texttt{.equals}(\texttt{null}) \) is always false