Principles of Software Construction: Objects, Design and Concurrency

Packages and Inheritance

Christian Kaestner    Charlie Garrod

© 2012-13 C Garrod, J Aldrich, and W Scherlis
Homework 1: Representing graphs

Two common representations

- *Adjacency matrix*
- *Adjacency list*

**Adjacency matrix**

```
   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**

```
a -> b
b -> c
c -> a
a -> d
d -> a
```
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

**AbstractDog**

- **Dog**
  - **Chihuaua**
  - ... (other breeds)
  - **German Shepherd**
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

<table>
<thead>
<tr>
<th></th>
<th>Small-scale</th>
<th>Larger-scale</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data</strong></td>
<td>Primitives, Arrays, Structures</td>
<td>Objects, Heaps</td>
</tr>
<tr>
<td><strong>Control</strong></td>
<td>Basic (if, while, ;), Function/method calls</td>
<td>Method dispatch, Concurrency</td>
</tr>
<tr>
<td><strong>Naming and Reference</strong></td>
<td>Local variables, Parameters</td>
<td>Package, imports, Visibility, Qualification</td>
</tr>
</tbody>
</table>
Visibility of properties and methods

```java
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

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Class</th>
<th>Package</th>
<th>Subclass</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>protected</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>default</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>private</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
Packages and qualified names

- E.g., three ways to refer to a `java.util.Queue`:
  - Use the full name:
    ```java
    java.util.Queue q = ...;
    q.add(...);
    ```
  - Import `java.util.Queue`, then use the unqualified name:
    ```java
    import java.util.Queue;
    Queue q = ...;
    ```
  - Import the entire `java.util` package:
    ```java
    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

- «interface» brand
- Name of class or interface in top compartment
- Dashed line, open triangle arrowhead for implements
- Return type comes after method or field
- Fields in middle compartment
- Italics means abstract
- Optional visibility:
  + for public
  - for private
  # for protected
  ~ for package (not used much)
- Solid line, open triangle arrowhead for extends

```
«interface» Dog

getName() : String
getBreed() : String
bark() : String
setName(name : String)
toString() : String

AbstractDog

- name : String
  - breed : String

+ getName() : String
+ getBreed() : String
+ bark() : String
+ setName(name : String)
  # setBreed(breed : String)
  + toString() : String

GermanShephard

bark() : String
play()
```

15-214 Garrod
Another example: different kinds of bank accounts

<table>
<thead>
<tr>
<th>«interface» CheckingAccount</th>
<th>«interface» SavingsAccount</th>
</tr>
</thead>
<tbody>
<tr>
<td>getBalance() : float</td>
<td>getBalance() : float</td>
</tr>
<tr>
<td>deposit(amount : float)</td>
<td>deposit(amount : float)</td>
</tr>
<tr>
<td>withdraw(amount : float) : boolean</td>
<td>withdraw(amount : float) : boolean</td>
</tr>
<tr>
<td>transfer(amount : float, target : Account) : boolean</td>
<td>transfer(amount : float, target : Account) : boolean</td>
</tr>
<tr>
<td>getFee() : float</td>
<td>getInterestRate() : float</td>
</tr>
</tbody>
</table>
A better design: An account type hierarchy

<table>
<thead>
<tr>
<th>«interface» Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>getBalance() : float</td>
</tr>
<tr>
<td>deposit(amount : float)</td>
</tr>
<tr>
<td>withdraw(amount : float) : boolean</td>
</tr>
<tr>
<td>transfer(amount : float, target : Account) : boolean</td>
</tr>
<tr>
<td>monthlyAdjustment()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>«interface» CheckingAccount</th>
</tr>
</thead>
<tbody>
<tr>
<td>getFee() : float</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>«interface» InterestCheckingAccount</th>
</tr>
</thead>
</table>

Multiple interface extension

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

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

If we know we have a CheckingAccount, additional methods are available.
A better design: An account type hierarchy

```java
public interface CheckingAccount extends Account {
    // Methods
}

public interface InterestCheckingAccount extends CheckingAccount, SavingsAccount {
    // Methods
}
```

- Account is a supertype of SavingsAccount.
- SavingsAccount is a subtype of Account. Account methods are inherited (copied to CheckingAccount).
- 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

```
«interface» Account
getBalance() : float
deposit(amount : float)
withdraw(amount : float) : boolean
transfer(amount : float, target : Account) : boolean
monthlyAdjustment()

«interface» CheckingAccount
getFee() : float

CheckingAccountImpl...

«interface» SavingsAccount
getInterestRate() : float

SavingsAccountImpl...

«interface» InterestCheckingAccount

InterestCheckingAccountImpl...
```
Better: Reuse abstract account code

```java
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 AbstractAccount
    implements CheckingAccount {
    public void monthlyAdjustment() {
        balance -= getFee();
    }

    public float getFee() {
        /* fee calculation */
    }
}
```

«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()

CheckingAccountImpl

monthlyAdjustment()
getFee() : float
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 AbstractAccount implements CheckingAccount {
    public void monthlyAdjustment() {
        balance -= getFee();
    }
    public float getFee() { /* fee calculation */ }
}

// diagram explanation:
- 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 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();
    }
    // ...

CheckingAccountImpl depends on BasicAccountImpl
Reuse via code wrappers, version 2: Delegation

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

Account

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

adjustment

- doAdjust()

adjustment

- doAdjust()

BasicAccountImpl

- balance: float

«interface» Account

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

«interface» Adjustment

- doAdjust()
Reuse via code wrappers, version 2: Delegation

```java
public void adjustAll(Adjustment[] adjs) {
    for (Adjustment a : adjs) {
        a.doAdjust();
    }
}
```

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

```java
public abstract class AbstractAccount implements Account {
    protected float balance = 0.0;
    public boolean withdraw(float amount) {
        // withdraws money from account (code not shown)
        return true; // placeholder
    }
}
```

```java
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
  
  ```java
  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.,
      ```
      void baz(Car f rob) {
       /* 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 $T_1$ 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!
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 to be executed
  ▪ 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 \( m_2 \)

\[ \text{e.g.: } x.\text{foo}(\text{apple, 42}) \]
Method dispatch, revisited

- **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

Example: \( x\text{.foo(apple, 42)} \)
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.println(A.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!"; }
}

Bird B = new Bird();
System.out.print(B.getNoise());
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!"; }
}

GenericAnimal B = new Bird();
System.out.print(B.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!"; }
}

GenericAnimal C = new Cat();
System.out.print(C.getNoise());
Method dispatch practice, part D

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

What is printed by:
### Method dispatch practice, part E-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!"; }
}

GenericAnimal E = new Ewokian();
System.out.print(E.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!"; }
}

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:
  ```java
  String    toString()
  boolean   equals(Object obj)
  int       hashCode()
  Object    clone()
  ```
Method dispatch practice, part F

```java
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();
    }
}
```