合肥學院
HEFEI UNIVERSITY

OOP with Java

18. Visiblity, Encapsulation, `final`, and `static` Inner Classes

Thomas Weise · 汤卫思

tweise@hfuu.edu.cn · http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Faculty of Computer Science and Technology
Institute of Applied Optimization
230601 Shushan District, Hefei, Anhui, China
Econ. & Tech. Devel. Zone, Jinxiu Dadao 99

合肥学院 南艳湖校区/南2区
计算机科学与技术系
应用优化研究所
中国 安徽省 合肥市 蜀山区 230601
经济技术开发区 锦绣大道99号

Outline

1. Introduction
2. Visibility
3. Encapsulation
4. The `final` Keyword
5. Inner Classes
6. Summary

Introduction

- If you write a program, usually you are not just doing it alone

Introduction

- If you write a program, usually you are not just doing it alone
- Often, you work in a team

Introduction

- If you write a program, usually you are not just doing it alone
- Often, you work in a team
- Or you write code to be used by other people

Introduction

- If you write a program, usually you are not just doing it alone
- Often, you work in a team
- Or you write code to be used by other people
- Packages and Classes are ways to structure your code

Introduction

- If you write a program, usually you are not just doing it alone
- Often, you work in a team
- Or you write code to be used by other people
- Packages and Classes are ways to structure your code
- But how can we ensure that your code is used correctly?

Class Visibility

- Java allows you to create classes inside a package with two levels of visibility

Class Visibility

- Java allows you to create classes inside a package with two levels of visibility:
  - package-private classes

Class Visibility

- Java allows you to create classes inside a package with two levels of visibility:
  - package-private classes
    - are declared in the form `class XYZ`

Class Visibility

- Java allows you to create classes inside a package with two levels of visibility:
  - package-private classes
    - are declared in the form `class XYZ`
    - are only visible to the code inside the same package

Class Visibility

- Java allows you to create classes inside a package with two levels of visibility:
  - package-private classes
    - are declared in the form `class XYZ`
    - are only visible to the code inside the same package
    - cannot be refered to by their canonical name or `import`ed into from any other package or subpackage

Class Visibility

- Java allows you to create classes inside a package with two levels of visibility:
  - package-private classes
    - are declared in the form `class XYZ`
    - are only visible to the code inside the same package
    - cannot be refered to by their canonical name or `import`ed into from any other package or subpackage
  - public classes

Class Visibility

- Java allows you to create classes inside a package with two levels of visibility:
  - package-private classes
    - are declared in the form `class XYZ`
    - are only visible to the code inside the same package
    - cannot be refered to by their canonical name or `import`ed into from any other package or subpackage
  - public classes
    - are decleared in the form `public class XYZ`

Class Visibility

- Java allows you to create classes inside a package with two levels of visibility:
  - package-private classes
    - are declared in the form `class XYZ`
    - are only visible to the code inside the same package
    - cannot be refered to by their canonical name or `import`ed into from any other package or subpackage
  - public classes
    - are decleared in the form `public class XYZ`
    - can be `import`ed and used from anywhere

Class Visibility

- Java allows you to create classes inside a package with two levels of visibility:
  - package-private classes
    - are declared in the form `class XYZ`
    - are only visible to the code inside the same package
    - cannot be refered to by their canonical name or `import`ed into from any other package or subpackage
  - public classes
    - are decleared in the form `public class XYZ`
    - can be `import`ed and used from anywhere
- This way, we can separate our code into

Class Visibility

- Java allows you to create classes inside a package with two levels of visibility:
  - package-private classes
    - are declared in the form `class XYZ`
    - are only visible to the code inside the same package
    - cannot be refered to by their canonical name or `import`ed into from any other package or subpackage
  - public classes
    - are decleared in the form `public class XYZ`
    - can be `import`ed and used from anywhere
- This way, we can separate our code into
  - a public API and classes to be used by others (declared as `public`)

Class Visibility

- Java allows you to create classes inside a package with two levels of visibility:
  - package-private classes
    - are declared in the form `class XYZ`
    - are only visible to the code inside the same package
    - cannot be refered to by their canonical name or `import`ed into from any other package or subpackage
  - public classes
    - are decleared in the form `public class XYZ`
    - can be `import`ed and used from anywhere
- This way, we can separate our code into
  - a public API and classes to be used by others (declared as `public`) and
  - our internal helper classes which nobody should mess with

Why?

- Why separate code into private and public classes?

Why?

- Why separate code into private and public classes?
- All (exposed) APIs must be specified, maintained, and documented

Why?

- Why separate code into private and public classes?
- All (exposed) APIs must be specified, maintained, and documented because you are part of a team in a company

Why?

- Why separate code into private and public classes?
- All (exposed) APIs must be specified, maintained, and documented because you are part of a team in a company
- Because the more code/API you expose, the more you have to maintain!

Why?

- Why separate code into private and public classes?
- All (exposed) APIs must be specified, maintained, and documented because you are part of a team in a company
- Because the more code/API you expose, the more you have to maintain!
- Whatever you make accessible might be used by someone

Why?

- Why separate code into private and public classes?
- All (exposed) APIs must be specified, maintained, and documented because you are part of a team in a company
- Because the more code/API you expose, the more you have to maintain!
- Whatever you make accessible might be used by someone
- Whenever you change it (API, behavior), that other guy's code will stop working

Why?

- Why separate code into private and public classes?
- All (exposed) APIs must be specified, maintained, and documented because you are part of a team in a company
- Because the more code/API you expose, the more you have to maintain!
- Whatever you make accessible might be used by someone
- Whenever you change it (API, behavior), that other guy's code will stop working
- But not all of your code will be intended for other people to use, some will really just be some internal stuff

Why?

- Why separate code into private and public classes?
- All (exposed) APIs must be specified, maintained, and documented because you are part of a team in a company
- Because the more code/API you expose, the more you have to maintain!
- Whatever you make accessible might be used by someone
- Whenever you change it (API, behavior), that other guy's code will stop working
- But not all of your code will be intended for other people to use, some will really just be some internal stuff
- You make these classes package private

Why?

- Why separate code into private and public classes?
- All (exposed) APIs must be specified, maintained, and documented because you are part of a team in a company
- Because the more code/API you expose, the more you have to maintain!
- Whatever you make accessible might be used by someone
- Whenever you change it (API, behavior), that other guy's code will stop working
- But not all of your code will be intended for other people to use, some will really just be some internal stuff
- You make these classes package private because only then you can change them whenever and however you like.

Why?

- Why separate code into private and public classes?
- All (exposed) APIs must be specified, maintained, and documented because you are part of a team in a company
- Because the more code/API you expose, the more you have to maintain!
- Whatever you make accessible might be used by someone
- Whenever you change it (API, behavior), that other guy's code will stop working
- But not all of your code will be intended for other people to use, some will really just be some internal stuff
- You make these classes package private because only then you can change them whenever and however you like.
- (of course, someone could change your code to make a package private class `public`, but then *they* break the specification and then it is *their* problem)

Member Visibility

- But we can do more than not just want to either expose or hide a class completely

Member Visibility

- But we can do more than not just want to either expose or hide a class completely
- Some part of a class may belong to a public API, some other may just internal implementation and helpers

Member Visibility

- But we can do more than not just want to either expose or hide a class completely
- Some part of a class may belong to a public API, some other may just internal implementation and helpers
- Java provides four levels of visibility that can be applied to all class members (instance variables, static variables, instance methods, static methods)

Member Visibility

- But we can do more than not just want to either expose or hide a class completely
- Some part of a class may belong to a public API, some other may just internal implementation and helpers
- Java provides four levels of visibility that can be applied to all class members (instance variables, static variables, instance methods, static methods):
  - `private`: only the methods of this very class can see the member

Member Visibility

- But we can do more than not just want to either expose or hide a class completely
- Some part of a class may belong to a public API, some other may just internal implementation and helpers
- Java provides four levels of visibility that can be applied to all class members (instance variables, static variables, instance methods, static methods):
  - `private`: only the methods of this very class can see the member
  - (nothing specified): package private, the member can be seen by all code in the same package

Member Visibility

- But we can do more than not just want to either expose or hide a class completely
- Some part of a class may belong to a public API, some other may just internal implementation and helpers
- Java provides four levels of visibility that can be applied to all class members (instance variables, static variables, instance methods, static methods):
  - `private`: only the methods of this very class can see the member
  - (nothing specified): package private, the member can be seen by all code in the same package
  - `protected`: the member can be accessed from code in this class and all of its subclasses and all classes in the same package

Member Visibility

- But we can do more than not just want to either expose or hide a class completely
- Some part of a class may belong to a public API, some other may just internal implementation and helpers
- Java provides four levels of visibility that can be applied to all class members (instance variables, static variables, instance methods, static methods):
  - `private`: only the methods of this very class can see the member
  - (nothing specified): package private, the member can be seen by all code in the same package
  - `protected`: the member can be accessed from code in this class and all of its subclasses and all classes in the same package
  - `public`: the member is visible to everybody

Member Visibility

- But we can do more than not just want to either expose or hide a class completely
- Some part of a class may belong to a public API, some other may just internal implementation and helpers
- Java provides four levels of visibility that can be applied to all class members (instance variables, static variables, instance methods, static methods):
  - `private`: only the methods of this very class can see the member
  - (nothing specified): package private, the member can be seen by all code in the same package
  - `protected`: the member can be accessed from code in this class and all of its subclasses and all classes in the same package
  - `public`: the member is visible to everybody
- We always use the strictest visibility

Member Visibility

- But we can do more than not just want to either expose or hide a class completely
- Some part of a class may belong to a public API, some other may just internal implementation and helpers
- Java provides four levels of visibility that can be applied to all class members (instance variables, static variables, instance methods, static methods):
  - `private`: only the methods of this very class can see the member
  - (nothing specified): package private, the member can be seen by all code in the same package
  - `protected`: the member can be accessed from code in this class and all of its subclasses and all classes in the same package
  - `public`: the member is visible to everybody
- We always use the strictest visibility, because the bigger the visibility, the more maintenance effort it will cost us later

Collection Example

- Arrays in Java have a fixed length

Collection Example

- Arrays in Java have a fixed length
- We want to implement some list classes which allow us to add elements, i.e., have a dynamic length

Collection Example

- Arrays in Java have a fixed length
- We want to implement some list classes which allow us to add elements, i.e., have a dynamic length
- We internally use arrays to store their content and allocate new arrays when needed

Collection Example

- Arrays in Java have a fixed length
- We want to implement some list classes which allow us to add elements, i.e., have a dynamic length
- We internally use arrays to store their content and allocate new arrays when needed
- We define a base class with some public API

Collection Example

- Arrays in Java have a fixed length
- We want to implement some list classes which allow us to add elements, i.e., have a dynamic length
- We internally use arrays to store their content and allocate new arrays when needed
- We define a base class with some public API
- We extend this base class for `int`, `double`, and `float`

Collection Example

- Arrays in Java have a fixed length
- We want to implement some list classes which allow us to add elements, i.e., have a dynamic length
- We internally use arrays to store their content and allocate new arrays when needed
- We define a base class with some public API
- We extend this base class for `int`, `double`, and `float`
- The variables actually representing the data are `private`, the lists can only be manipulated via the `public` methods

Listing: Basic List Class: Package Private Constructor, Public API

```
package cn.edu.hfuu.iao.collections;

/** a base class for lists: public API but package private constructor */
public class List {

  /** package private variable holding list length */
  int size;

  /** package private constructor: only we can derive subclasses */
  List() {
    super();
  }

  /** get the size of this list */
  public int size() {
    return this.size;
  }

  /** reverse this list */
  public void reverse() { // do nothing yet
  }

  /** return array representation of this list */
  public Object toArray() {
    return null;
  }
}
```

IntList Class: Implementation for `int`

Listing: IntList Class: Implementation for `int`

```java
package cn.edu.hfuu.iao.collections;

/** a list of integer values, based on class List */
public class IntList extends List {
  /** the actual internal data */
  private int[] data;

  /** create an int list based on a data array */
  public IntList(int[] _data) {
    this.data = _data;
    this.size = _data.length;
  }

  /** copy the data from this list to a destination array */
  private void __copyTo(final int[] dest) {
   for(int index = this.size; (--index) >= 0; ) {
     dest[index] = this.data[index];
   }
  }

  /** add an integer value to the list */
  public void append(final int value) {
    if(this.size >= this.data.length) {      // if capacity limit is reached
      int[] newData = new int[this.size*2];  // allocate a much bigger array to avoid frequent re-allocation
      this.__copyTo(newData);                // copy data to new array
      this.data = newData;                   // remember new array, old value of data becomes subject to GC
    } // after this if-then body, we are sure that the data array is big enough to hold one more element
    this.data[this.size++] = value; // store value at the end of data and increment list length
  }


  /** reverse this list */
  @Override
  public void reverse() { // notice how we use two counters and two counter updates in the loop below
    for(int i = 0, j = this.size-1; i < j; ++i, --j) { // we are allowed to do that, is that cool or what?
      int t = this.data[i];         // copy value at index i
      this.data[i] = this.data[j]; // store value from index j there
      this.data[j] = t;            // now store old value from index i at data[j]
    }
  }

  /** Return the array representation of this list.
   *  Notice the usage of int[] instead of Object return value.
   *  We are allowed to do this, because int[] inherits (is subset of) Object. */
  @Override
  public int[] toArray() {          // transform this list to an int[] array
    int[] res = new int[this.size]; // allocate array of the right size
    this.__copyTo(res);             // use private __copyTo method to copy the contents of list to res
    return res;                     // return res
  }

  /** create string representation of list */
  @Override
  public String toString() {
    String s;
    s = ""; //$NON-NLS-1$
    for(int index = 0; index < this.size; ++index) {
      if(s != "") { s += ", "; } //$NON-NLS-1$ //$NON-NLS-2$
      s += this.data[index];
    }
    return s;
  }
}
```

A Program Using our IntList Class

Listing: A Program Using our IntList Class

```
package cn.edu.hfuu.iao;

import cn.edu.hfuu.iao.collections.IntList;

/** a class where we use int lists */
public class IntListTest {
  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    IntList list;

    list = new IntList(new int[] { 12, 3 }); // create list with contents 12, 3
    System.out.println(list); // prints 12, 3
    list.append(-4);          // appends -4 to the list
    System.out.println(list); // prints 12, 3, -4
    list.reverse();           // reverse list
    System.out.println(list); // print -4, 3, 12
    // list.__copyTo(new int[100]); // this is not possible, __copyTo is not visible
    System.out.println(list.size()); // this is allowed, prints 3
    // System.out.println(list.size); // not allowed, variable size is package private
  }
}
```

DoubleList Class: Implementation for `Double`

Listing: DoubleList Class: Implementation for `Double`

```
package cn.edu.hfuu.iao.collections;

/** A list of double values, based on class List.
 *  Notice how all methods are marked with final so they cannot be overriden. A subclass can only add new methods. */
public class DoubleList extends List {
  /** the actual internal data */
  private double[] data;

  /** create an double list based on a data array */
  public DoubleList(double[] _data) {
    this.data = _data;
    this.size = _data.length;
  }

  /** copy the data from this list to a destination array */
  private final void __copyTo(final double[] dest) {
   for(int index = this.size; (--index) >= 0; ) {
     dest[index] = this.data[index];
   }
  }

  /** add an double value to the list */
  public final void append(final double value) {
    if(this.size >= this.data.length) {      // if capacity limit is reached
      double[] newData = new double[this.size*2];  // allocate a much bigger array to avoid frequent re-allocation
      this.__copyTo(newData);                 // copy data to new array
      this.data = newData;                    // remember new array, old value of data becomes subject to GC
    } // after this if-then body, we are sure that the data array is big enough to hold one more element
    this.data[this.size++] = value; // store value at the end of data and increment list length
  }


  /** reverse this list */
  @Override
  public final void reverse() { // notice how we use two counters and two counter updates in the loop below
    for(int i = 0, j = this.size-1; i < j; ++i, --j) { // we are allowed to do that, is that cool or what?
      double t = this.data[i];     // copy value at index i
      this.data[i] = this.data[j]; // store value from index j there
      this.data[j] = t;            // now store old value from index i at data[j]
    }
  }

  /** Return the array representation of this list.
   *  Notice the usage of double[] instead of Object return value.
   *  We are allowed to do this, because double[] inherits (is subset of) Object. */
  @Override
  public final double[] toArray() {       // transform this list to an double[] array
    double[] res = new double[this.size]; // allocate array of the right size
    this.__copyTo(res);                   // use private __copyTo method to copy the contents of list to res
    return res;                           // return res
  }

  /** create string representation of list */
  @Override
  public final String toString() {
    String s;
    s = ""; //$NON-NLS-1$
    for(int index = 0; index < this.size; ++index) {
      if(s != "") { s += ",␣"; } //$NON-NLS-1$ //$NON-NLS-2$
      s += this.data[index];
    }
    return s;
  }
}
```

A Program Using our DoubleList Class

Listing: A Program Using our DoubleList Class

```
package cn.edu.hfuu.iao;

import cn.edu.hfuu.iao.collections.DoubleList;

/** a class where we use float lists */
public class DoubleListTest {
  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    DoubleList list;

    list = new DoubleList(new double[] { 12d, 3d }); // create list with contents 12.0,
        3.0
    System.out.println(list); // prints 12.0, 3.0
    list.append(-4d);          // appends -4.0 to the list
    System.out.println(list); // prints 12.0, 3.0, -4.0
    list.reverse();           // reverse list
    System.out.println(list); // print -4.0, 3.0, 12.0
    // list.__copyTo(new double[100]); // this is not possible, __copyTo is not visible
    System.out.println(list.size()); // this is allowed, prints 3
    // System.out.println(list.size); // not allowed, variable size is package private
  }
}
```

FloatList Class: Implementation for `Float`

Listing: FloatList Class: Implementation for `Float`

```
package cn.edu.hfuu.iao.collections;

/** A list of float values, based on class List.
 *  Notice how the class is marked as final, so it cannot be subclassed. */
public final class FloatList extends List {
  /** the actual internal data */
  private float[] data;

  /** create an float list based on a data array */
  public FloatList(float[] _data) {
    this.data = _data;
    this.size = _data.length;
  }

  /** copy the data from this list to a destination array */
  private final void __copyTo(final float[] dest) {
   for(int index = this.size; (--index) >= 0; ) {
     dest[index] = this.data[index];
   }
  }

  /** add an float value to the list */
  public final void append(final float value) {
    if(this.size >= this.data.length) {      // if capacity limit is reached
      float[] newData = new float[this.size*2];  // allocate a much bigger array to avoid frequent re-allocation
      this.__copyTo(newData);                 // copy data to new array
      this.data = newData;                    // remember new array, old value of data becomes subject to GC
    } // after this if-then body, we are sure that the data array is big enough to hold one more element
    this.data[this.size++] = value; // store value at the end of data and increment list length
  }


  /** reverse this list */
  @Override
  public final void reverse() { // notice how we use two counters and two counter updates in the loop below
    for(int i = 0, j = this.size-1; i < j; ++i, --j) { // we are allowed to do that, is that cool or what?
      float t = this.data[i];     // copy value at index i
      this.data[i] = this.data[j]; // store value from index j there
      this.data[j] = t;            // now store old value from index i at data[j]
    }
  }

  /** Return the array representation of this list.
   *  Notice the usage of float[] instead of Object return value.
   *  We are allowed to do this, because float[] inherits (is subset of) Object. */
  @Override
  public final float[] toArray() {       // transform this list to an float[] array
    float[] res = new float[this.size]; // allocate array of the right size
    this.__copyTo(res);              // use private __copyTo method to copy the contents of list to res
    return res;                      // return res
  }

  /** create string representation of list */
  @Override
  public final String toString() {
    String s;
    s = ""; //$NON-NLS-1$
    for(int index = 0; index < this.size; ++index) {
      if(s != "") { s += ", "; } //$NON-NLS-1$ //$NON-NLS-2$
      s += this.data[index];
    }
    return s;
  }
}
```

A Program Using our FloatList Class

Listing: A Program Using our FloatList Class

```
package cn.edu.hfuu.iao;

import cn.edu.hfuu.iao.collections.FloatList;

/** a class where we use double lists */
public class FloatListTest {
  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    FloatList list;

    list = new FloatList(new float[] { 12f, 3f }); // create list with contents 12.0, 3.0
    System.out.println(list); // prints 12.0, 3.0
    list.append(-4f);           // appends -4.0 to the list
    System.out.println(list); // prints 12.0, 3.0, -4.0
    list.reverse();           // reverse list
    System.out.println(list); // print -4.0, 3.0, 12.0
    // list.__copyTo(new float[100]); // this is not possible, __copyTo is not visible
    System.out.println(list.size()); // this is allowed, prints 3
    // System.out.println(list.size); // not allowed, variable size is package private
  }
}
```

Encapsulation

- Object = code + data

Encapsulation

- Object = code + data
- It is always bad to let somebody else manipulate the data by directly accessing the variables

Encapsulation

- Object = code + data
- It is always bad to let somebody else manipulate the data by directly accessing the variables
- Because then you have no control about what they might do

Encapsulation

- Object = code + data
- It is always bad to let somebody else manipulate the data by directly accessing the variables
- Because then you have no control about what they might do
- Imagine what kind of bugs could result from someone directly working on the `data` and `size` fields of our `IntLis`?

Encapsulation

- Object = code + data
- It is always bad to let somebody else manipulate the data by directly accessing the variables
- Because then you have no control about what they might do
- Imagine what kind of bugs could result from someone directly working on the `data` and `size` fields of our `IntLis`?
- We should always aim for achieving encapsulation of the data

Encapsulation

- Object = code + data
- It is always bad to let somebody else manipulate the data by directly accessing the variables
- Because then you have no control about what they might do
- Imagine what kind of bugs could result from someone directly working on the `data` and `size` fields of our `IntLis`?
- We should always aim for achieving encapsulation of the data
- Encapsulation means that the variables of an object can only be accessed and manipulated via methods

Encapsulation

- Object = code + data
- It is always bad to let somebody else manipulate the data by directly accessing the variables
- Because then you have no control about what they might do
- Imagine what kind of bugs could result from someone directly working on the `data` and `size` fields of our `IntLis`?
- We should always aim for achieving encapsulation of the data
- Encapsulation means that the variables of an object can only be accessed and manipulated via methods, i.e., are `private`

An example for a class with complete encapsulation

Listing: Class representing bank accounts: we need sanity checks!

```
package cn.edu.hfuu.iao;

/** A class for a bank account with complete encapsulation */
public class BankAccount {

  /** the account number: clearly private */
  private String accountNumber;
  /** the amount of money in the account in cents: also private */
  private long balance; // we use long, not double, because an account cannot have "fractional" cents

  /** create a new bank account with balance 0 */
  public BankAccount(String number){
    this.accountNumber = number;
  }

  /** get the account's balance */
  public double getBalance() {
    return this.balance;
  }

  /** add some money to the bank account */
  public void deposit(long amount) {
    if((amount > 0L) && (amount < 1_000_000_00L)) { // sanity check: you can only deposit a positive amount
      this.balance += amount;                       // of money, and anything above 1 million is probably an error
    } else { // an invalid amount cannot be put into the account
      System.out.println("Invalid deposit amount " + amount + //$NON-NLS-1$
          " for account " + this); //$NON-NLS-1$
    }
  }

  /** withdraw some money from the bank account */
  public void withdraw(long amount) {
    if((amount > 0L) && (amount < 1_000_00L)) { // sanity check: you can only withdraw a positive amount of
      this.balance -= amount;                   // money and at most 1000 RMB at once
    } else {
      System.out.println("Invalid withdrawal amount " + amount + //$NON-NLS-1$
          " for account " + this); //$NON-NLS-1$
    }
  }

  /** transfer some money from this account to another one */
  public void transferTo(long amount, BankAccount other) {
    if((other  != null) && (other != this) &&          // the other bank account must not be null and different
       (amount  >   0L) && (amount < 1_000_000_00L) && // you can only transfer a positive amount in 0..1 million RMB
       (amount  < this.balance)) {                     // and you must have enough money for the transfer
      this.balance  -= amount;
      other.balance += amount;
    } else {
      System.out.println("Cannot transfer " + amount +      //$NON-NLS-1$
                         " from " + this + " to " + other); //$NON-NLS-1$ //$NON-NLS-2$
    }
  }

  public String toString() {
    return '(' + this.accountNumber + ": " + this.balance + ')'; //$NON-NLS-1$
  }
}
```

The `final` Keyword

- Besides "hiding" fields, we can also make them impossible to modify.

The `final` Keyword

- Besides "hiding" fields, we can also make them impossible to modify.
- For this, there exists `final` keyword.

The `final` Keyword

- Besides "hiding" fields, we can also make them impossible to modify.
- For this, there exists `final` keyword.
- Declaring an XXX as `final` means

The `final` Keyword

- Besides "hiding" fields, we can also make them impossible to modify.
- For this, there exists `final` keyword.
- Declaring an XXX as `final` means...
  - instance variable: you have to set its value in the constructor and afterwards can never change it again

The `final` Keyword

- Besides "hiding" fields, we can also make them impossible to modify.
- For this, there exists `final` keyword.
- Declaring an XXX as `final` means...
  - instance variable: you have to set its value in the constructor and afterwards can never change it again
  - `static` variable: you have to set its value right in the declarion, it's the same as a constant

The `final` Keyword

- Besides "hiding" fields, we can also make them impossible to modify.
- For this, there exists `final` keyword.
- Declaring an XXX as `final` means...
  - instance variable: you have to set its value in the constructor and afterwards can never change it again
  - `static` variable: you have to set its value right in the declarion, it's the same as a constant
  - instance method: subclasses cannot override it

The `final` Keyword

- Besides "hiding" fields, we can also make them impossible to modify.
- For this, there exists `final` keyword.
- Declaring an XXX as `final` means...
  - instance variable: you have to set its value in the constructor and afterwards can never change it again
  - `static` variable: you have to set its value right in the declarion, it's the same as a constant
  - instance method: subclasses cannot override it
  - `static` method: subclasses cannot declare a `static` method with same signature hiding it (always do this)

The `final` Keyword

- Besides "hiding" fields, we can also make them impossible to modify.
- For this, there exists `final` keyword.
- Declaring an XXX as `final` means...
  - instance variable: you have to set its value in the constructor and afterwards can never change it again
  - `static` variable: you have to set its value right in the declarion, it's the same as a constant
  - instance method: subclasses cannot override it
  - `static` method: subclasses cannot declare a `static` method with same signature hiding it (always do this)
  - local variable inside a method: the local variable can only be assigned once

The `final` Keyword

- Besides "hiding" fields, we can also make them impossible to modify.
- For this, there exists `final` keyword.
- Declaring an XXX as `final` means...
  - instance variable: you have to set its value in the constructor and afterwards can never change it again
  - `static` variable: you have to set its value right in the declarion, it's the same as a constant
  - instance method: subclasses cannot override it
  - `static` method: subclasses cannot declare a `static` method with same signature hiding it (always do this)
  - local variable inside a method: the local variable can only be assigned once
  - method parameter: cannot be changed inside method – always do this, changing parameter values in a method is confusing

The `final` Keyword

- Besides "hiding" fields, we can also make them impossible to modify.
- For this, there exists `final` keyword.
- Declaring an XXX as `final` means...
  - instance variable: you have to set its value in the constructor and afterwards can never change it again
  - `static` variable: you have to set its value right in the declarion, it's the same as a constant
  - instance method: subclasses cannot override it
  - `static` method: subclasses cannot declare a `static` method with same signature hiding it (always do this)
  - local variable inside a method: the local variable can only be assigned once
  - method parameter: cannot be changed inside method – always do this, changing parameter values in a method is confusing
- We should declare as much stuff as `final` as possible

The `final` Keyword

- Besides "hiding" fields, we can also make them impossible to modify.
- For this, there exists `final` keyword.
- Declaring an XXX as `final` means...
  - instance variable: you have to set its value in the constructor and afterwards can never change it again
  - `static` variable: you have to set its value right in the declarion, it's the same as a constant
  - instance method: subclasses cannot override it
  - `static` method: subclasses cannot declare a `static` method with same signature hiding it (always do this)
  - local variable inside a method: the local variable can only be assigned once
  - method parameter: cannot be changed inside method – always do this, changing parameter values in a method is confusing
- We should declare as much stuff as `final` as possible, because if something can be changed, someone will change it, and this makes debugging and maintenance harder

Example for a class with encapsulation and `final`

Listing: Class representing bank accounts using `final`

```
package cn.edu.hfuu.iao;

/** A class for a bank account with complete encapsulation and the final keyword */
public final class BankAccountFinal { // we declare the class final, we don't allow subclassing

  /** the account number: clearly private, clearly never changes, so it should be final */
  private final String accountNumber;
  /** the amount of money in the account in cents: also private */
  private long balance; // we use long, not double, because an account cannot have "fractional" cents

  /** create a new bank account with balance 0 */
  public BankAccountFinal(final String number){ // number parameter is final, it cannot be changed inside constructor
    this.accountNumber = number;                // why would we want to change it anyway..
  }

  /** get the account's balance */
  public final double getBalance() {  // the method is marked as final. If the class was not already marked as final,
    return this.balance;              // then it would still not be possible to override the method
  }

  /** add some money to the bank account */
  public final void deposit(final long amount) {
    if((amount > 0L) && (amount < 1_000_000_00L)) { // sanity check: you can only deposit a positive amount
      this.balance += amount;                       // of money, and anything above 1 million is probably an error
    } else { // an invalid amount cannot be put into the account
      System.out.println("Invalid deposit amount " + amount + //$NON-NLS-1$
          " for account " + this); //$NON-NLS-1$
    }
  }

  /** withdraw some money from the bank account */
  public final void withdraw(final long amount) {
    if((amount > 0L) && (amount < 1_000_00L)) { // sanity check: you can only withdraw a positive amount of
      this.balance -= amount;                   // money and at most 1000 RMB at once
    } else {
      System.out.println("Invalid withdrawal amount " + amount + //$NON-NLS-1$
          " for account " + this); //$NON-NLS-1$
    }
  }

  /** transfer some money from this account to another one */
  public final void transferTo(final long amount, final BankAccountFinal other) {
    if((other  != null) && (other != this) &&          // the other bank account must not be null and different
       (amount  >   0L) && (amount < 1_000_000_00L) && // you can only transfer a positive amount in 0..1 million RMB
       (amount  < this.balance)) {                     // and you must have enough money for the transfer
      this.balance  -= amount;
      other.balance += amount;
    } else {
      System.out.println("Cannot transfer " + amount +      //$NON-NLS-1$
                         " from " + this + " to " + other); //$NON-NLS-1$ //$NON-NLS-2$
    }
  }

  public final String toString() {
    return '(' + this.accountNumber + ": " + this.balance + ')'; //$NON-NLS-1$
  }
}
```

Inner Classes

- Besides putting a class into a package, we can also put it inside another class

Inner Classes

- Besides putting a class into a package, we can also put it inside another class
- This treats the outer class basically as a package

Inner Classes

- Besides putting a class into a package, we can also put it inside another class
- This treats the outer class basically as a package
- It allows us to specify an internal helper class as `private`, i.e., to be more strict than "package private" with it

Inner Classes

- Besides putting a class into a package, we can also put it inside another class
- This treats the outer class basically as a package
- It allows us to specify an internal helper class as `private`, i.e., to be more strict than "package private" with it
- Inner classes can be `static`, i.e., unrelated to any instance of the outer class (we look only at this case)

Inner Classes

- Besides putting a class into a package, we can also put it inside another class
- This treats the outer class basically as a package
- It allows us to specify an internal helper class as `private`, i.e., to be more strict than "package private" with it
- Inner classes can be `static`, i.e., unrelated to any instance of the outer class (we look only at this case)
- They can also be non-`static` instance classes and need a surrounding instance of the outer class ... let's ignore this for now

Example for a class with an inner class: A Map

Listing: Class representing a Map using Inner Class for Map Entries

```java
package cn.edu.hfuu.iao.collections;

/** a map which stores key-value relationships */
public final class Map {

  /** the entry list: see the private class Entry below */
  private Entry[] entries;

  /** create a map */
  public Map() {                  // create the map
    this.entries = new Entry[32]; // with space for some entries
  }

  /** store that key should now be related to value */
  public final void put(final Object key, final Object value) {
    for (int index = 0; index < this.entries.length; index++) {     // first check all stored keys
      if (this.entries[index] == null) {                            // if we get here, we have reached the end of the map
        this.entries[index] = new Entry(key, value);                // since we did not find key, just put a new entry
        return; // and we can exit
      }
      if (this.entries[index].key == key) {                         // check if there already is an entry for the specified key
        this.entries[index].value = value;                          // if so, we need to change its associated value
        return;                                                     // and can exit
      }
    } // if we get to after the loop, this means that the entry list is full, but does not contain key
    Entry[] newEntries = new Entry[this.entries.length * 2];        // so we need to allocate a larger entry list
    for(int i = this.entries.length; (--i) >= 0; ) { newEntries[i] = this.entries[i]; } // copy all existing entries
    newEntries[this.entries.length] = new Entry(key, value);        // and at the end of this list, we put the new entry
    this.entries = newEntries;                                      // and store the new entry list
  }

  /** transform to string */
  public final String toString() {
    String string = ""; //$NON-NLS-1$
    for (Entry entry : this.entries) {                              // fast iteration over all entries
      if (entry == null) { return string; }                         // end of list reached
      if (string != "") { string += ",␣"; } //$NON-NLS-1$ //$NON-NLS-2$
      string += entry.key + "=" + entry.value;                      //add key-value relationship //$NON-NLS-1$
    }
    return string; // return string
  }

  /** an inner class storing a map entry: This class is ONLY visible inside the Map class */
  private static final class Entry {
    /** the key, can never be changed and only accessed from enclosing class */
    final Object key;
    /** the value stored in the entry, can be changed (only from enclosing class) */
    Object value;
    /** create an entry */
    Entry(final Object _key, final Object _value) {
      this.key = _key;
      this.value = _value;
    }
  }
}
```

Example for Using our Map

Listing: Class using our Map class

```
package cn.edu.hfuu.iao;

import cn.edu.hfuu.iao.collections.Map;

/** a class where we use our map */
public class MapTest {
  /**
   * The main routine
   *
   * @param args
   *          we ignore this parameter
   */
  public static void main(String[] args) {
    Map map = new Map();

    map.put("Hello", "World!"); //$NON-NLS-1$ //$NON-NLS-2$
    System.out.println(map); // Hello=World!

    map.put("Country", "China"); //$NON-NLS-1$ //$NON-NLS-2$
    System.out.println(map); // Hello=World!, Country=China

    map.put("Computer␣Science", "Fun"); //$NON-NLS-1$ //$NON-NLS-2$
    System.out.println(map); // Hello=World!, Country=China, Computer Science=Fun

    map.put("Hello", "Class!"); //$NON-NLS-1$ //$NON-NLS-2$
    System.out.println(map); // Hello=Class!, Country=China, Computer Science=Fun

    map.put("This␣Course", "Nice"); //$NON-NLS-1$ //$NON-NLS-2$
    System.out.println(map); // Hello=Class!, Country=China, Computer Science=Fun, This Course=Nice

    map.put("This␣Course", "so-so"); //$NON-NLS-1$ //$NON-NLS-2$
    System.out.println(map); // Hello=Class!, Country=China, Computer Science=Fun, This Course=so-so
  }
}
```

Summary

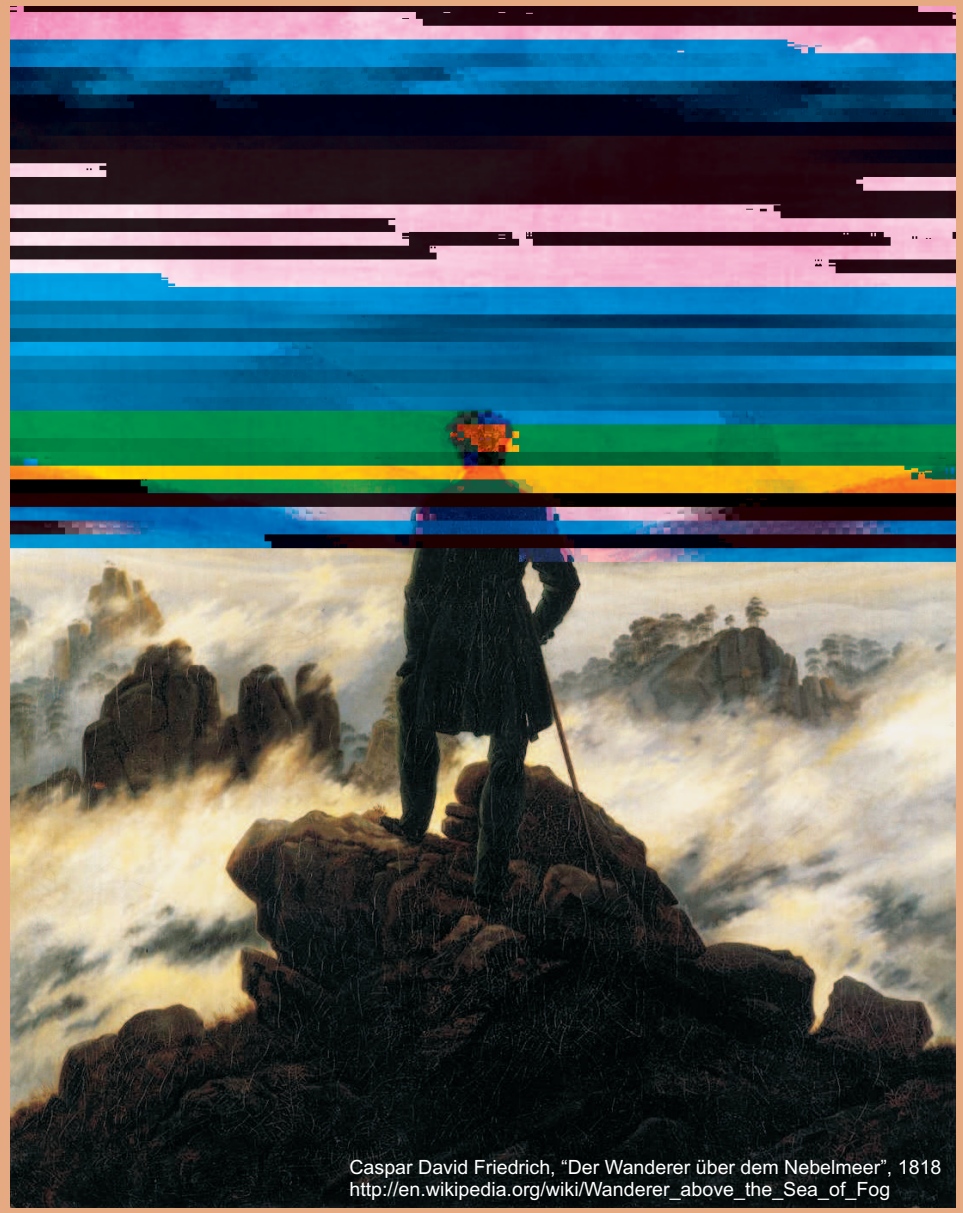
- If we contribute code to a project or team, be aware that:
  - everything which is visible will be used by someone (and any change you apply to it later may break other code)
  - every variable which is visible and can be changed will be changed by someone (and you will not have any control over how it will be changed)
  - any classes that can be subclassed and methods that can be overridden will eventually be subclassed/overridden
- Always apply the tightest possible visibility to any variable or method, ideally `private`, which makes them visible only to the current class
- If a variable does not need to be changed, mark it as `final`
- If a method does not need to be overridden or a class does not need to be subclassed, mark them as `final`
- Inner classes allow us to make classes `private` to an enclosing class

谢谢

Thank you

Thomas Weise [汤卫思]
tweise@hfuu.edu.cn
http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Institute of Applied Optimization
Shushan District, Hefei, Anhui,
China



Caspar David Friedrich, “Der Wanderer über dem Nebelmeer”, 1818
http://en.wikipedia.org/wiki/Wanderer_above_the_Sea_of_Fog