Lecture 07: Object Encapsulation & References

AITI Nigeria Summer 2012
University of Lagos.
Data Field Encapsulation

- Sometimes we want variables to be accessible only within the class itself
  - Hide from other classes

- Prevents undesired/incorrect tampering with variables by methods outside of the class
  - Maintain consistency of state
Without Encapsulation..

class BankAccount {
    //Fields
    double balance;
    String name;

    //constructor
    BankAccount(String name, double openBalance) {
        this.name = name;
        this.balance = openBalance;
    }
}
class AnotherClass {
    static void main(String[] args) {
        //create bank account
        BankAccount mikesAccount =
            new BankAccount("Mike", 10000000);

        //some tampering...
        mikesAccount.name = "Zach";
    }
}

This is not good for poor Mike!
Visibility Modifiers

- **public** – makes methods and data fields accessible by any other class
- **private** – makes methods and data fields accessible only from within its own class
- (neither) – similar to public but a bit more restricted
class BankAccount {

    // data fields
    private double balance;
    private String name;

    // constructor
    BankAccount(String name, double openBalance) {
        this.name = name;
        this.balance = openBalance;
    }
}
Common Object Oriented Practices

- **Accessors** – *get* the value of a data field
  - Sometimes called *getters*

- **Mutators** – *set* the value of a data field
  - Sometimes called *setters*
public class BankAccount {

    //accessors
    public double getBalance(){
        return balance;
    }

    public String getName(){
        return name;
    }
}
BankAccount, add mutators

//mutators
public void deposit(double amount){
  ...
}

public void withdraw (double amount){
  ...
}

Notice there is no access to the name data field! Now Zach can’t steal Mike’s account.
class AnotherClass {
    static void main(String[] args) {
        //create bank account
        BankAccount mikesAccount =
            new BankAccount ("Mike", 5);

        //Illegal
        mikesAccount.name = "Zach";
        //Illegal
        mikesAccount.balance = 100000000;
    }
}

Now we are safe!
private Methods

• Methods of a class that are declared private can only be called within the class.

```java
private void setName(String newName) {
    ...
}
```
class AnotherClass {
    static void main(String[] args) {
        //create bank account
        BankAccount mikesAccount =
            new BankAccount ("Mike", 5);

        //Illegal, private method of Bank Account
        mikesAccount.setName("Zach");
    }
}

Now we are safe!
Accessibility Intuition

• Accessibility modifiers are not used for safety
  – There are ways around them in Java!

• They are used for encapsulation!
  – Hide unnecessary state/methods from user of class
  – Prevent access to state to maintain object consistency
class Family {
    Person[] males;
    Person[] females;

    //want totalMembers = males + females
    int totalMembers = 0;

    …

    public void addFemale(Person person)…
    public void addMale(Person person)…
}
Inconsistent

class AnotherClass {
    void method() {
        Family myFam = new Family();
        myFam.addMale(new Person("Mike"));
        myFam.addFemale(new Person("Mary"));
        myFam.totalMembers = 10;
        //now myFam is inconsistent!
    }
}
A Better Way!

class Family {
    private Person[] males;
    private Person[] females;
    //want totalMembers = males + females
    private int totalMembers = 0;

    ... 

    public void addFemale(Person person) {
        females[...] = person;
        totalMembers++;
    }
}

Object References

• An object variable is really a reference to the object.
  – A pointer is a good way of thinking about it

• You must “dereference” the variable to access method and fields
  – Ex: `person.getName()` , `course.number`
You can have 2 variables reference the same object

```java
Integer a = new Integer(5);
Integer b = a;
// a and b reference the same object
```
Primitive Argument Passing

• Remember that primitive arguments are passed by value.

• If you change a primitive argument inside of a method, the variable in the calling method will remain unchanged.
public static int meth(int a, int b) {
    a = a * 2;
    b = b * 3;
    return a + b;
}

public static void main(String[] args) {
    int x = 5;
    int y = 10;
    int z = 0;

    z = meth(x, y);
    //what is the value of x and y?
}
Object Argument Passing

- Object Arguments are pass by reference
  - A copy is not made

- Any changes to the object in the method are visible in the calling method
void changeName(Person person) {
    person.setName("Mike");
}

public static void main(String[] args) {
    Person cory = new Person("Cory");
    changeName(person);
    // what is the value cory.getName()?