University of British Columbia
CPSC 111, Intro to Computation
Alan J. Hu

Interfaces vs. Inheritance
Abstract Classes
Inner Classes

Lecture 22

Readings

This Week: No new readings. Consolidate!

(Reminder: Readings are absolutely vital for learning this stuff!)

Labs and Tutorials

This Week: Lab #10

Labs are due at end of lab time! (Lab #10 is fairly short, but make sure to finish by the end of lab.)

Final Exam

- Wednesday, April 15, 7pm in SRC A
  - This wasn't a good room last year, but we're stuck with the date, time, and room UBC assigns. ☹
  - Note: You are allowed to do scratch work on the exam paper! (But write your answer where indicated if the problem tells you to.)

Programming Assignment 3

- Assignment 3 is up on WebCT!
  - Click on the "Assignments" icon.
  - Assigned Sunday evening – sorry for delay!
- Due at NOON, April 6 (Monday), via electronic hand in.
- Tips:
  - There is some Eclipse setup. Set-up ASAP!
  - Work in pairs. Some conceptual stuff.
  - Think carefully before coding. If concepts right, the coding is much much easier.

Learning Goals

By the end of class today you will be able to...

- List similarities and differences between interfaces and inheritance.
- Create abstract classes and extend abstract classes.
- Write code that uses inner classes.
- (Work through complicated examples of parameter passing and arrays.)

Interfaces vs. Superclasses

We learned these as completely separate concepts:

- An **interface** is a contract, specifying some methods that must be implemented by any class that claims to implement the interface.
- A **superclass** is a class from which other classes can **inherit** methods and instance fields, so we can reuse the superclass's implementation.

Interfaces vs. Superclasses

But they have similarities…

- Both allow creating different, new classes that share some of the same methods, e.g.
  - **`Double`** and **`UBCStudent`** both implement **`Comparable`**, so they both have **`int compareTo()`**
  - **`Swimmer`** and **`Crawler`** both inherit from **`Animal`**, so they have e.g., **`changeImage()`**

Interfaces vs. Superclasses

But they have similarities…

- Both allow declaring references that can point to different kinds of objects, e.g.,
  - **`Comparable x;`**
    `x = new Double(3.14);`
    `x = new UBCStudent("Alan");`
  - **`Person y;`**
    `y = new UBCStudent("Alan",0.0);`
    `y = new Celebrity("Paris Hilton");`

Interfaces vs. Superclasses

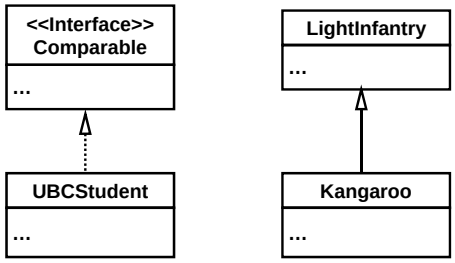
But they have similarities…

- Therefore, both allow polymorphism, e.g.,
  - **`Comparable x;`**
    `...`
    `if (x.compareTo(…)) …`
  - **`Animal y;`**
    `...`
    `y.advanceOneTimeStep(…);`

Interfaces vs. Superclasses

But they have similarities…

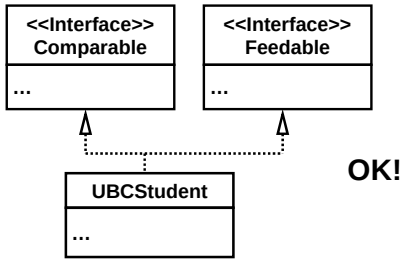
- They even have similar UML:



Interfaces vs. Superclasses

They have differences, too…

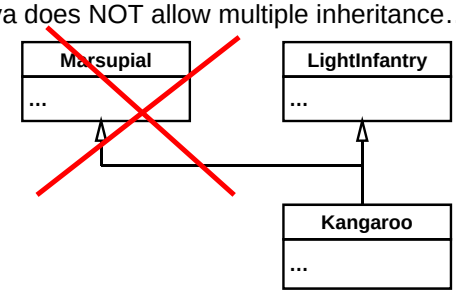
- Java allows implementing multiple interfaces



Interfaces vs. Superclasses

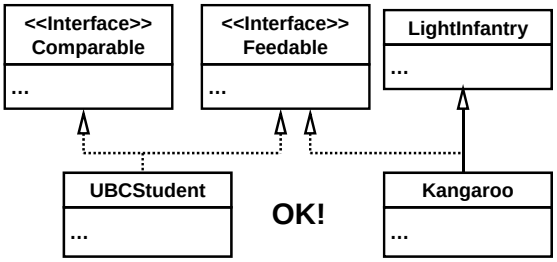
They have differences, too…

- Java does NOT allow multiple inheritance…



Interfaces vs. Superclasses

- (Java does allow one inheritance plus multiple interfaces.)



Interfaces vs. Superclasses

The fundamental difference:

- An interface provides **no** implementations.
- **Everything** in a (super)class is implemented.

Interfaces vs. Superclasses

The fundamental difference:

- To implement interface, a class **must implement** everything.
- Subclasses **automatically inherit** superclass **implementation**. Can **optionally override**.

Interfaces vs. Superclasses

Therefore:

- You **cannot create objects** of an interface type (only references).
- You **can create objects** of the (super)class type. (You can create references, too.)

Questions

Interfaces vs. Superclasses

The fundamental difference:

- An interface provides **no** implementations.
- **Everything** in a (super)class is implemented.

Wouldn't it be cool to have something in-between?

Abstract Classes

- The classes we have written so far are called ***concrete classes***.
- ***Abstract classes*** in Java provide a blend of the concepts of interfaces and inheritance:
  - Some (from none to all) methods are left unimplemented. These are called *abstract methods.*
  - Instances fields and implemented methods are also allowed.
  - Subclass must implement abstract methods (in order to become concrete).
  - Subclass inherits or overrides other stuff.

Abstract Class Syntax

- To create an abstract class, just add the **`abstract`** keyword:

```
public abstract class Foo {
  …
}
```

Abstract Method Syntax

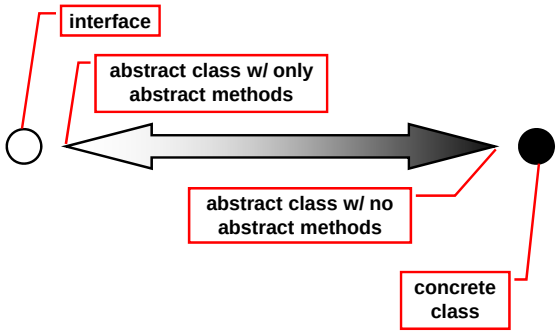
- Inside an abstract class, you are allowed to declare abstract methods:
  - Just add the **`abstract`** keyword.
  - And put a semicolon instead of a body.

```
public abstract class Foo {
  …
  public abstract void display();
  …
}
```

Abstract Classes vs. Interfaces/Inheritance

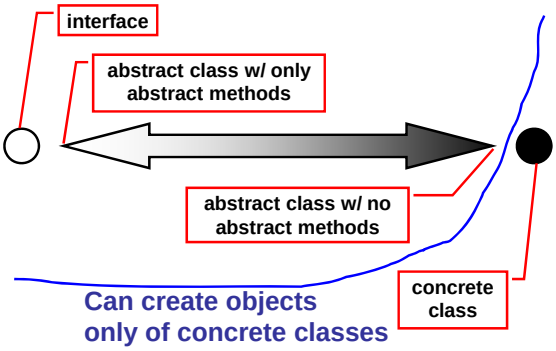
- An abstract class with no abstract methods is basically a normal, concrete class.
  - (But Java still thinks it's an abstract class!)
- An abstract class with all abstract methods is basically like an interface.
  - (But Java still thinks it's an abstract class!)
- An abstract class with some abstract methods is somewhere in-between.

The Interface-Class Spectrum



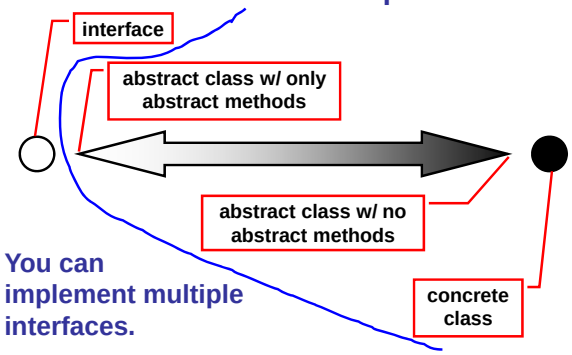
Presentation idea due to Steve Wolfman

The Interface-Class Spectrum



Presentation idea due to Steve Wolfman

The Interface-Class Spectrum



Presentation idea due to Steve Wolfman

Questions

Inner Classes

- So far, classes are always in a file by themselves (with the same name as the class).
- Sometimes, we just want a simple little class for something. It's a pain to have a separate file.
- We can declare classes ***inside*** another:
  - method
  - class

Example: Inner Class in a Method

```
public static void main(String[] args) {
   …
   class Foo {
        private int x;
        public int getX() { return x; }
   }
   …
}
```

Foo is available only inside main.

Foo can access surrounding vars with restrictions.

Inner Class Scope

- Remember the general principle that you can see out through curly braces, but not in.
- Therefore, inner classes should be able to see variables declared around them:
  - Instance variable in same class
  - Local variables if declared inside a method
- However, object of inner class can outlive the code where it was created!
  - Access is to instance variables of object that created inner class.
  - Access only to **final** local variables. (These can be references to mutable objects, though.)

Questions

Review: Passing Parameters

- In the object user (the caller):
  a.flatter(“fabulous”);

- In the class definition (the callee):

```
public void flatter(String adjective) {
    System.out.println("Wow, you look " +
                                adjective + "!");
}
```

Review: Passing Parameters

- In the object user (the caller):
  a.flatter(“fabulous”);

adjective = “fabulous”;

- In the class definition (the callee):

```
public void flatter(String adjective) {
    System.out.println("Wow, you look " +
                                adjective + "!");
}
```

Parameter Passing Challenge #1

Consider the following method:

```
public void swap( int a, int b )
{
  int temp = a;
  a = b;
  b = temp;
}
```

and the following code from main that calls swap:

```
  int x = 0;
  int y = 5;
  swap( x, y );
```

What values are stored in the x and y after the above code segment has executed?

Parameter Passing Challenge #2

Consider the following method:

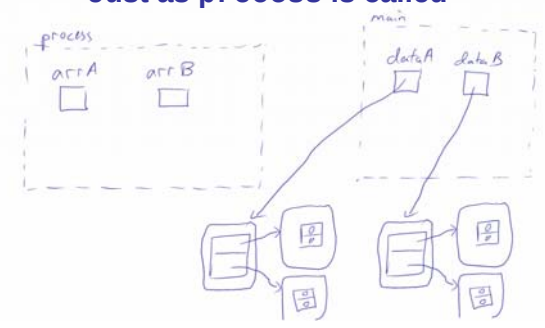
```
public void process( int[][] arrA, int[][] arrB )
{
  int row, col;
  int[][] arrC = new int[][] { { 1, 1, 1 }, { 1, 1, 1 } };
  arrA = arrC;
  for( row = 0; row < arrB.length; row++ )
    for( col = 0; col < arrB[ row ].length; col++ )
      arrB[ row ][ col ] = row + col;
}
```

and the following code from main that calls process:

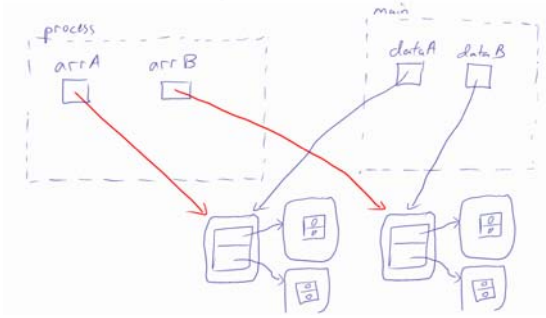
```
  int[][] dataA = new int[][] { { 0, 0 }, { 0, 0 } };
  int[][] dataB = new int[][] { { 0, 0 }, { 0, 0 } };
  process( dataA, dataB );
```

What values are stored in the arrays `dataA` and `dataB` after the above code segment has executed?

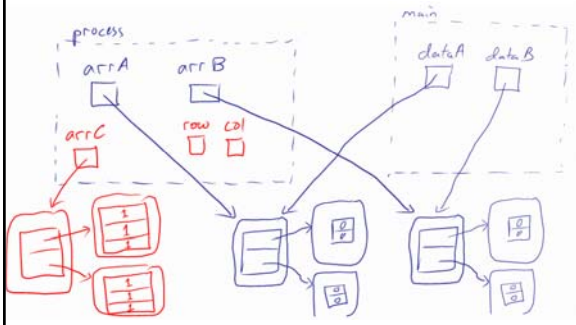
Just as `process` is called



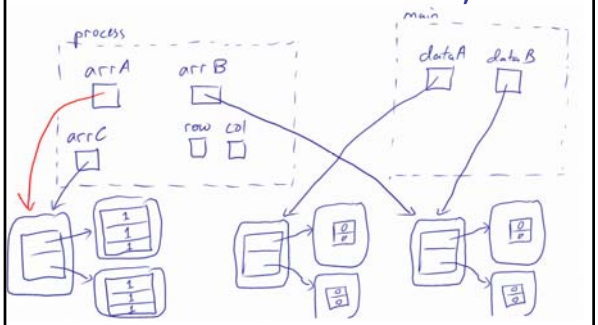
Just before `process`'s first line



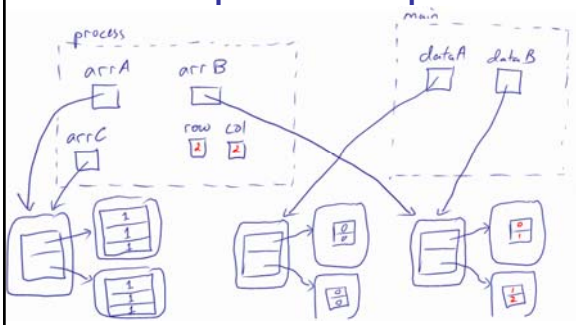
Just after `arrC` is initialized



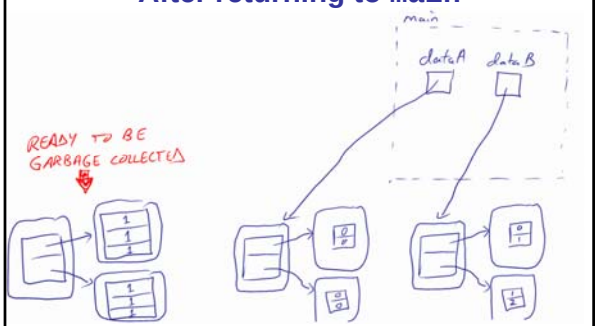
Just after "`arrA = arrC;`"



Just as `process` completes



After returning to `main`



Parameter Passing Intuition

- Calling a method is like asking a friend to do something for you.
- Passing parameters is giving your friend the information needed to do the task.
- In Java, parameters are **always** "call by value" (sometimes "pass by value"):
  - You make a copy of the info for your friend.
  - Java makes a copy of the parameters (the value of the parameters) for the method call.
- Passing primitive type values vs. objects?

Real Life Analogy

- You show me a $20 bill. I make a photocopy. Can I spend your money? If I burn my photocopy, do I destroy your $20 bill?
- You show me your credit card. I make a photocopy. Can I spend your money? If I burn my photocopy, do I destroy your credit card?
- $20 bill is like a primitive type. It **is** the value.
- Credit card is like an object reference. It says how to find the value (your credit line).

Thoughts

In **every** case, the variables in main still stored the same values they did before.

Java is "pass-by-value"; we only pass the **values** of arguments to the parameters of a method.

So, we can **never** change those arguments.

But… if the variable is a reference variable, the object it **points to** can be changed!

That's what happened to dataB.

Why didn't dataA change?

Parameter Passing Challenge #3

Consider the following method:

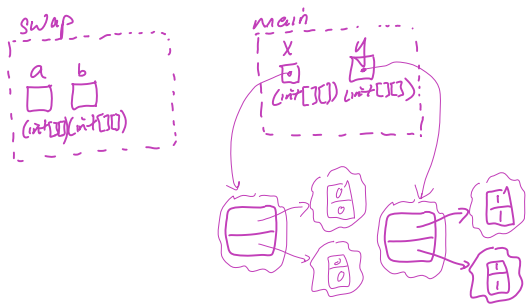
```
public void swap( int[][] a, int[][] b )
{
  int[][] temp = a;
  a = b;
  b = temp;
}
```

and the following code that calls swap:

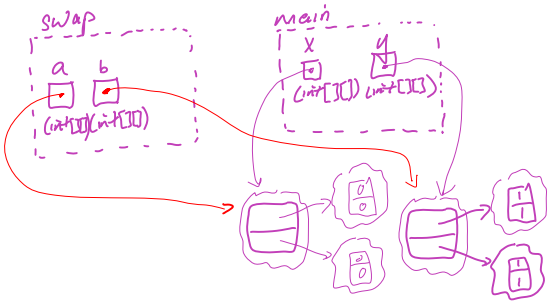
```
  int[][] x = new int[][] {{0, 0}, {0, 0}};
  int[][] y = new int[][] {{1, 1}, {1, 1}};
  swap( x, y );
```

What values are stored in the x and y after the above code segment has executed?

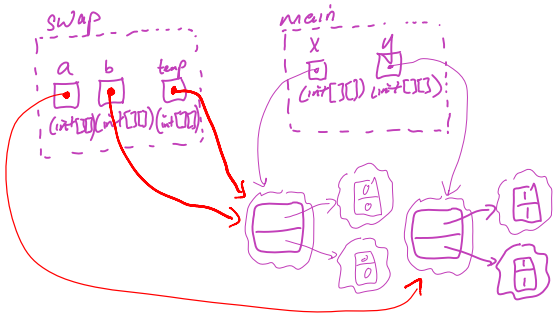
Just as swap is called



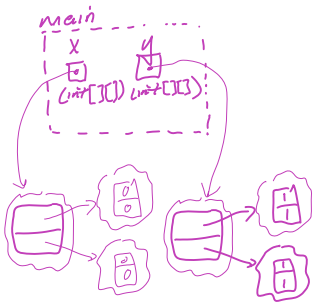
Just before swap's first line



Just as swap completes



After returning to `main`



How do I get swap to work?

- Java doesn't really have a good way to do this. How do I swap the contents of two pockets if you give me just copies of the contents?
- With object references, I can kind of do it:
  - I can't swap your references, but I can follow the object references to the objects and change the contents of the objects.
- Other languages (like C++) let you create references to any variable (even primitive types).
  - If two of you give me (a copy of) the **locations** of your dorm rooms, I can swap the contents.
  - This is called "call by reference" or "pass by reference".

Something like swap

```
private static void swap(int[][] a, int[][] b) {
   assert a.length == b.length;
   int[][] temp = new int[a.length][];
   for (int i=0; i<a.length; i++) {
       temp[i] = a[i];
   }
   for (int i=0; i<b.length; i++)
       a[i] = b[i];
   for (int i=0; i<b.length; i++)
       b[i] = temp[i];
}
```

and the following code that calls swap:

```
int[][] x = new int[][] {{0, 0}, {0, 0}};
int[][] y = new int[][] {{1, 1}, {1, 1}};
swap( x, y );
```

What values are stored in the x and y after the above code segment has executed?