Overloading, simple constructors, static methods
Function/method overloading

- *Overloading* allows us to reuse method names, i.e., implement the same operator for different types of data.
- C++ allows overloading of *infix, unary and other operators*, Java only allows overloading of methods.
- While overloaded methods have the same name, their *signatures must differ*.
  - Not enough for return values to differ, arguments themselves must differ.

- Overloading **IS NOT** overriding.
  - *Overriding* is defining a method in a derived class with the same name and signature as a method in a base class (*"same" means after conversions*).
  - *Overloading* is using the same method name but different signatures within a class or in a base class, in Java.
import java.io.*;

class User {
    private String name;
    private int age;

    public User(String str, int yy) { name = str; age = yy; }

    public void print() {
        System.out.println("name: " + name + " age: " + age);
    }

    public String foo(String s, int i) {
        System.out.println("User::foo was called with a string and an int");
        return s+i;
    }

    public String foo(String s1, String s2) {
        System.out.println("User::foo was called with a string and a string");
        return s1.concat(s2);
    }

    public String foo(int i, int j) {
        System.out.println("User::foo was called with an int and an int");
        return ""+i+""+i;
    }
}

Overloading in Java -- examples
Overloading in Java -- examples

class User {
    public String foo(String s, int i) {
        System.out.println("User::foo was called with a string and an int");
        return s+i;
    }

    public String foo(String s1, String s2) {
        System.out.println("User::foo was called with a string and a string");
        return s1.concat(s2);
    }

    public String foo(int i, int j) {
        System.out.println("User::foo was called with an int and an int");
        return "+i"+"+i;"
    }
}
Example of calling overloaded methods

import java.io.*;

class Test2 {

    public static void main(String args[]) {
        User u = new User("Car", 54);
        String s1 = u.foo("hello ", 0);
        String s2 = u.foo(-1, 0);
        String s3 = u.foo("super", " man");
        System.out.println("s1: "+s1+"; s2: "+s2+"; s3: "+s3);
    }
}

[ece-76-55:codew2b/java/overLoad] smidkiff% java Test
User::foo was called with a string and an int
User::foo was called with an int and an int
User::foo was called with a string and a string
s1 hello 0, s2: -1-1, s3: super man
[ece-76-55:codew2b/java/overLoad] smidk
What if no overloaded function matches exactly?

Class C {
    . . .
    public void foo(int i) { . . . }
    public void foo(long l) { . . . }
    . . .
}

public static void main(String args[]) {
    short s = -4;
    C c = new C( );
    c.foo(s);
}
What are the widening primitive conversions?

- byte to short, int, long, float, or double
- short to int, long, float, or double
- char to int, long, float, or double
- int to long, float, or double
- long to float or double
- float to double

This gives us a clue as to what parameters may be closest to the calling argument types.
What if no overloaded function matches exactly?

• Form the set $M$ of all visible methods $m$ that match the call exactly or with only widening conversions.

• Examining the set $M$
  – If there is an exact match in $M$, use it
  – For each $m_i$ and $m_j$, if the parameters of $m_i$ can match $m_j$ with widening conversions, discard $m_j$. If after doing this only one method remains, use it

• Otherwise, declare an error

• blue text above is basically looking for the method whose parameters are closest to the argument types of the call.
Example 1

visible methods:

public void foo(int i, double d) {...}
public void foo(int i, float d) {...}
public void foo(char c, double d) {...}
public void foo(short i, float d) {...}

The call to be matched:

int j; double z;
foo(j, z);

• $M$ is blue method definition
• public void foo(int i, float d) {...} not included because cannot widen double $z$ to float
• foos with char and short not included because cannot widen int $j$ to short or double
• public void foo(int i, double d) {...} is only member of $M$ and so it is used
Example 2

Class C {
    
    public void foo(int i) { . . . }  
    public void foo(long l) { . . . }  
    
}

public static void main(String args[]) {
    short s = -4;
    C c = new C();
    c.foo(s);
}

• $M$ is all blue definitions
  – short widens to int
  – short widens to long
  – Nothing matches exactly

• let $m_i = \text{foo}(\text{int} \ i)$ and $m_j = \text{foo}(\text{long} \ l)$
  – int widens to long
  – get rid of $m_j = \text{foo}(\text{long} \ l)$
Example 3

visible methods:

public void foo(int i, double d) {...}  
public void foo(char c, double d) {...}  
public void foo(short i, double d) {...}  

The call to be matched:

int j; double z; float f; short s;  
foo(s, f);

• $M$ is all blue definitions  
  – short matches short exactly, widens to int  
  – float widens to double  
  – Nothing matches exactly

• let $m_i = \text{foo}(\text{short } i, \text{double } d)$ and $m_j = \text{foo}(\text{int } i, \text{double } d)$  
  – short widens to int  
  – double matches double  
  – get rid of $m_j = \text{foo}(\text{int } i, \text{double } d)$
Example 3

• One function left -- use it.

visible methods:

public void foo(int i, double d) {...}
public void foo(char c, double d) {...}
public void foo(short i, double d) {...}

The call to be matched:

int j; double z; float f; short s;
foo(s, f);
Example 4

visible methods:

```java
public void foo(int i; float f) {...}
public void foo(char c; double d) {...}
```

The call to be matched:

```java
char k; float x;
foo(k, x);
```

Intuitively, one visible method is closer match on one argument, another is closer on another argument. Neither is overall closer.

- $M$ is all blue definitions
  - nothing matches directly
- let $m_i = \text{foo}(\text{int } i; \text{ float } f)$ and $m_j = \text{foo}(\text{char } c; \text{ double } d)$
  - $\text{int}$ cannot widen to $\text{char}$
  - cannot get rid of $\text{foo}(\text{char } c; \text{ double } d)$
- let $m_i = \text{foo}(\text{char } c; \text{ double } d)$ and $m_j = \text{foo}(\text{int } i; \text{ float } f)$
  - $\text{char}$ widens to $\text{int}$
  - $\text{double}$ does not match or widen to $\text{float}$
  - cannot get rid of $\text{foo}(\text{short } s; \text{ float } f)$
- Multiple matches, error!
Simple constructors and using constructors and *final* to control inheritance
Constructors

• Constructors are the functions that assign values to an object, i.e. an instance of the class.
• Storage is allocated on the heap for Java objects by \textit{new}.
• Constructors initialize the storage for an object’s storage.
• Initialization can happen by default, and thus constructors perform both system and programmer specified actions.
• Let’s look at some Java examples.
Java Constructors
import java.io.*;

class User {
    private String name;
    private int age;

    public User(String str, int yy) {
        name = str;
        age = yy;
    }

    public void print() {
        System.out.println("name: " + name + " age: " + age);
    }
}
Sample derived class constructor

class StudentUser extends User {
    private String schoolEnrolled;

    public StudentUser(String nam, int y, String sch) {
        super(nam, y); // call base class constructor
        schoolEnrolled = sch;
    }

    public void print() {
        super.print();
        System.out.println("School: " + schoolEnrolled);
    }
}
Creating a derived class object

import java.io.*;

class Test {

    public static void main(String args[]) {
        StudentUser student = new StudentUser("Ralph", 54, "Bug Tech");
        student.print();
    }
}

The StudentUser constructor calls the User constructor (passing "Ralph" and 54) before initializing the rest of the StudentUser object.
Java zero-arg constructors

• Java is well-defined
  – Java is required to always have a defined state for objects
  – Java was a clean slate design - no backward compatibility

• Java calls a zero-arg constructor if no explicit call provided. You may not need to define this constructor.

• Like C++, an error results if no zero arg constructor supplied and non-zero arg constructors are supplied
  – This is done to catch typos typos and unexpanded constructor skeletons supplied by IDEs
  – If no zero-arg constructor supplied, Java initializes fields according to the table on the right.
  – This table also applies to default variable initializations

<table>
<thead>
<tr>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>FALSE</td>
</tr>
<tr>
<td>char</td>
<td>\u0000</td>
</tr>
<tr>
<td>all integer types</td>
<td>0</td>
</tr>
<tr>
<td>float</td>
<td>0.0f</td>
</tr>
<tr>
<td>double</td>
<td>0.0</td>
</tr>
<tr>
<td>object reference</td>
<td>null</td>
</tr>
</tbody>
</table>
import java.io.*;

class User {
    private String name = "Default Name";
    private int age = -1;

    // public User(String str, int yy) {
    //     name = str; age = yy;
    // }

    public void print( ) {
        System.out.println("name: " + name + " age: " + age);
    }
}

A default for name and age are defined. Note that C++ does not allow this to be done in class declarations.

This code can be found in java/zeroarg
Java zero-arg example

class StudentUser extends User {
    private String schoolEnrolled;

    public StudentUser(String nam, int y, String sch) {
        // super(nam, y);
        schoolEnrolled = sch;
    }

    public void print() {
        super.print();
        System.out.println("School: " + schoolEnrolled);
    }
}

ejavac Test.java
[ece-76-55:codew2b/java/zeroInit] smidkiff% java Test
name: Default Name age: -1
School: Bug Tech
The `DFoo` constructor does not specify a base class constructor to be called. Therefore the default zero arg constructor for `Foo` is called. This constructor sets `fooString = null`. 

The `DFoo` constructor does not specify a base class constructor to be called. Therefore the default zero arg constructor for `Foo` is called. This constructor sets `fooString = null`. 

```java
import java.io.*;

public class Foo {

    private final String fooString;

    public Foo() {fooString = null;}
    public Foo(String ln) {fooString = ln;}

    public void print() {
        System.out.println("Foo: " + fooString);
    }
}

import java.io.*;

public class DFoo extends Foo {

    private final String dfooString;

    public DFoo(String ln) {dfooString = ln;}

    public void print() {
        System.out.println("DFoo, printing super: ");
        super.print(); // invokes print in base (super) class
        System.out.println("DFoo: " + dfooString);
    }
}
```
import java.io.*;

class Test {
    public static void main(String args[]) {
        Foo f = new Foo("Foo object");
        f.print();
        DFoo d = new DFoo("DFoo object");
        d.print();
        ((Foo) d).print();
        f = d;
        f.print();
    }
}

Note call to Base class print.  But why does it print Foo: null for DFoo objects?
The answer to why $Foo$ is null lies in how the constructors are written.

The $DFoo$ constructor does not specify a base class constructor to be called. Therefore the default zero arg constructor for $Foo$ is called. This constructor sets $fooString = null$.

**import java.io.*;**

**public class Foo {**

  **private final String fooString;**

  **public Foo() {fooString = null;};**

  **public Foo(String ln) {fooString = ln;}**

  **public void print() {**
  
  **System.out.println("Foo: "+fooString);**
  
  **}**

  **}**

**import java.io.*;**

**public class DFoo extends Foo {**

  **private final String dfooString;**

  **public DFoo(String ln) {dfooString = ln;}**

  **public void print() {**
  
  **System.out.print("DFoo, printing super: ");**

  **super.print(); // invokes print in base (super) class**

  **System.out.println("DFoo: "+dfooString);**

  **}**

  **}**

The $DFoo$ constructor does not specify a base class constructor to be called. Therefore the default zero arg constructor for $Foo$ is called. This constructor sets $fooString = null$. 

From java/SuperInvoke/
This fixes the problem and calls the right constructor

```java
public class DFoo extends Foo {
    private final String dfooString;

    public DFoo(String ln) {
        super(ln);
        dfooString = ln;
    }

    public void print() {
        super.print();
    }
}
```

```java
class Test {
    public static void main(String args[]) {
        Foo f = new Foo("Foo object");
        f.print();

        DFoo d = new DFoo("DFoo object");
        d.print();

        ((Foo) d).print();

        f = d;
        f.print();
    }
}
```

Foo: Foo object
DFoo, printing super: Foo: DFoo object
DFoo: DFoo object
DFoo, printing super: Foo: DFoo object
DFoo: DFoo object
```

From java/SuperConstInvoke/
This fixes the problem and calls the right constructor

public class DFoo extends Foo {

    private final String dfooString;

    public DFoo(String ln) {
        dfooString = ln;
    }

    public void print() {
        super.print();
    }
}

Note that the call is within the constructor body.

class Test {

    public static void main(String args[]) {
        Foo f = new Foo("Foo object");
        f.print();
    }

    public DFoo d = new DFoo("DFoo object");
    d.print();
    ((Foo) d).print();

    f = d;
    f.print();
}

From java/SuperConstInvoke/
An example of polymorphic calls in a constructor

```java
public class B {
    private int i;
    public B( ) {i=1; foo( );}
    public void foo( ) {System.out.println("i: "+i);}
}

class D1 extends B {
    private int i;
    public D1( ) {i=2;}
    public void foo( ) {System.out.println("i: "+i);}
}

class Test {
    public static void main(String args[]) {
        D1 d1 = new D1( );
    }
}
```

**Diagram:**
- **D1 class**
  - B object
    - i = 0
  - D1 object
    - i = 0

**Code Explanation:**
- The class `B` has a default constructor that sets the `i` variable to 1 and calls the `foo` method.
- The class `D1` extends `B` and overrides the default constructor to set `i` to 2.
- In the `main` method of the `Test` class, a new `D1` object `d1` is created.

The `foo` method in both classes prints the value of `i`. When the `main` method creates a `D1` object, the `foo` method is called, printing `i: 2`.

---

**Additional Information:**
- The image also includes a diagram illustrating the inheritance and object creation from `B` to `D1`.
- The diagram shows the creation of an `D1` object and the value of `i` (initially 0) in both `B` and `D1` objects.
- The diagram also shows the call to the `foo` method in both contexts, demonstrating the polymorphic behavior.
An example of polymorphic calls in a constructor

```java
public class B {
    private int i;
    public B() {i=1; foo();}
    public void foo() {System.out.println("i: "+i);}
}

public class D1 extends B {
    private int i;
    public D1() {super(); i=2;}
    public void foo() {System.out.println("i: "+i);}
}

public class Test {
    public static void main(String args[]) {
        D1 d1 = new D1();
    }
}
```
An example of polymorphic calls in a constructor

```java
public class B {
    private int i;
    public B( ) {i=1; foo( );}
    public void foo( ) {System.out.println("i: "+i);}  
}

public class D1 extends B {
    private int i;
    public D1( ) {super( ); i=2;}
    public void foo( ) {System.out.println("i: "+i);}  
}

public class Test {
    public static void main(String args[]) {
        D1 d1 = new D1( );
    }
}
```
An example of polymorphic calls in a constructor

```java
public class B {
    private int i;
    public B() {i=1; foo();}
    public void foo() {System.out.println("i: "+i);}
}

public class D1 extends B {
    private int i;
    public D1() {super(); i=2;}
    public void foo() {System.out.println("i: "+i);}
}

public class Test {
    public static void main(String args[]) {
        D1 d1 = new D1();
    }
}
```
An example of polymorphic calls in a constructor

```java
class B {
    private int i;
    public B() {i=1; foo();}
    public void foo() {System.out.println("i: "+i);}
}

class D1 extends B {
    private int i;
    public D1() {super(); i=2;}
    public void foo() {System.out.println("i: "+i);}
}

class Test {
    public static void main(String args[]) {
        D1 d1 = new D1();
    }
}
```

`i: 0 is printed`
Static members and methods
class Base {
    int i;
    float z;

    public Base() {
    }
    public void print() {
        System.out.println(i + " " + z);
    }
}

• When a Base object is allocated, it is created on the heap
• When the print method is called, it needs to access the variables i and z associated with the object
• This is done by having a hidden parameter, this, passed into the method that points to the object so that the variables on the heap can be accessed.
The *this* reference

```java
class Base {
    int i;
    float z;

    public Base() {
    }
    public void print() {
        System.out.println(i + " " + z);
    }
}
```

- User code `r.print();` generates a call that looks like `r.print(this);`
- Since the reference `r` either contains the address of the object on which print is called, or allows the address to be found (`r` is, after all, a handle that allows the object to be accessed), coming up with a value of *this* is easy.
Static members

• Static members are variables that are associated with the class
• There is only one copy of them for the entire class. In the example, $B.x$ accesses class $B$’s $x$ variable.

```java
public class B {
    public static double x;
    ...
}
```

```java
B b = new B();
B.x = 2.0;
b.x = 1.0;
```
Static methods

• Static methods are associated with the class and *not* an object
• Static methods do not have access to a *this* pointer since they are not associated with any object
• Since static methods do not have access to a *this* pointer, they cannot access object members (since they are associated with no object, it is not clear whose object's members they would access.

```java
public class B {
    public static double x;
    public float y;
    static void foo() {
        // y = 1.0; *would be an error!*
        x = 1.0;
        ...  
    }
}
```

```java
B.foo();
B b = new B();
b.foo();
```

Both call the same static Foo in class B
Controlling inheritance in Java

• Java provides the final keyword

• When applied to a method, final keeps the method from being overridden, since to override the method it needs to be redefined, and final means that the given definition is the final definition.

• final public void print( ) { System.out.println(“this is a final, un-extendable method”);}
import java.io.*;

class User {
    private String name;
    private int age;

    public User(String str, int yy) {name = str; age = yy; }

    public void print( ) {
        System.out.println("name: " + name + " age: " + age);
    }
}

class StudentUser extends User {
    private String schoolEnrolled;

    public StudentUser(String nam, int y, String sch) {
        super(nam,y);
        schoolEnrolled = sch;
    }

    public void print( ) {
        super.print( );
        System.out.println("School: " + schoolEnrolled);
    }
}

[ece-76-55:codew2b/java/finalField] smidkiff% javac Test.java
./StudentUser.java:9: print() in StudentUser cannot override print() in User; overridden method is final
    public void print() {
        super.print();
        System.out.println("School: " + schoolEnrolled);
    }

1 error
One source of Java OO impurity - not everything is an object

• Java has object that are instantiations of classes, e.g. *Foo, DFoo, . . .*

• Java has primitives, e.g. *float, int, double*

• This causes problems when you try and write a function that takes either primitives or objects as arguments
  – The operations you can perform on each are often different (e.g. cannot invoke a method on an *int* or a *double*)
  – Have to be very careful when writing code like this
  – Can sometimes hide with other functions (We will do this as a mini programming assignment soon) but it requires effort on the part of the programmer
Abstract Classes
Abstract classes

- Abstract classes are classes for which objects cannot be constructed
- They can be derived from, however
- Abstract classes are a general OO concept, not a Java specific thing
Good for 3 things

1. Can lend organization to a class hierarchy,
2. Provides a common base class
3. Can represent a specialized behavior that when mixed with other classes gives a desired behavior

Can help build up an implementation

Let’s look at a concrete example to make these concepts clearer. In particular, let’s look at a shape class such as might be used in a drawing program.
A shape class

- It makes sense to construct a Circle, a Rectangle, etc., but not necessarily a shape
- It is useful to be able to refer to all shapes with a common class
abstract class Shape {
public:
    abstract public double area();
    abstract public double circumference;
    public double notAbstract() {
        System.out.println("Abstract classes can contain non-
        abstract methods!");
    }
    ...
}
A Shape interface in Java

```java
interface Shape {
    public:
        double pi = 3.14;
        abstract public double area();
        public double circumference; // note abstract is optional
        // ...
};
```

A Java interface - interfaces can only contain abstract methods and (non-abstract) constants.
Why two ways with Java?

- C++ allows multiple inheritance, java does not

- Thus, in C++ if a derived class Person needs to be both a Student and an Employee, the Person class can inherit from both Student and Employee.

- Things get ugly if both Student and Employee implement (directly or via inheritance) some method or field

- Which should be called in Person when not overridden? Implementation dependent with "modern" compilers flagging it as an error. *Don't do this!*
Multiple inheritance problem

- Student
  - foo()

- Employee
  - foo()

- Person
  - which foo()
Another problem

```
Base
foo()

Student
Base
foo()

Employee
Base
foo()

Person
which foo()

```
Interface or abstract?

• Use an abstract class A if the inheriting (derived) class D ISA A
• Use an interface I if the inheriting class D has the capabilities of an I
• When to use one or the other is not completely clear, however.
  • If you can define many, but not all, of the behaviors in the base class you might want to use an abstract class
  • If you want most or all methods overridden to have a specialized version in the derived class, you may want to use an interface
  • If most everything is overridden, there is not a strong ISA relationship
Java prohibits multiple inheritance

interface Shape {
    public double area ( );
    public double circumference( );
}

class Rectangle implements Shape {
    // ...
    // implement code for area and circumference
    // ...
}
Interfaces can extend other interfaces

- Only do this when you want to add to the functionality of the interface
- You cannot implement methods in the derived interface for the base methods of the interface
- There was/is an Eclipse bug that will claim the base class interface method's implementation is in the derived interface that extends it.
A longer Java example

Diagram:
- Shape
  - Polygon
    - Square
    - Rectangle
    - OtherPolygon
  - Other
  - CurvedShape
    - Circle
    - Elipse
    - OtherCurvedShape
A longer Java abstract class example

abstract class Shape {
    abstract protected double area();
    abstract protected double circumference();
}

abstract class Polygon extends Shape {
    abstract protected int numVertices;
    protected boolean starShape;
}

abstract class curvedShape extends Shape {
    protected void polygonalApprox();
}
class Circle extends CurvedShape {
    protected double r;
    protected static double PI = 3.14159;

    public Circle() {r = 1.0;}
    public Circle(double r) {this.r = r;}
    public double area() {return PI*r*r;}
    public double circumference() {return 2*PI*r;}
    public double getRadius() {return r;}
    public void polygonalApprox() {
        System.out.println("polygonalApprox code goes here");
    }
}
class Rectangle extends Polygon {
    double w, h;
    public Rectangle() {
        w = 0.0; h = 0.0; numVertices = 0; starShaped = true;
    }
    public Rectangle(double w, double h) {
        this.w = w;
        this.h = h;
        numVertices = 4;
        starShaped = true;
    }
    public double area() {return w*h;}
    public double circumference() {return 2*(w+h);}
    public double getWidth() {return w;}
    public double getHeight() {return h;}
}
class Test {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[3];
        shapes[0] = new Circle(2.9);
        shapes[1] = new Rectangle(1.0, 3.0);
        shapes[2] = new Rectangle(4.0, 2.0);

        double totArea = 0;
        for (int i = 0; i < shapes.length; i++)
            totArea += shapes[i].area();
        System.out.println("Total area = " + totArea);
    }
}
Interface example in Java

• Consider a *Drawable* class that implements methods for drawing
  • `setColor`
  • `setPosition`
  • `draw`
• In C++ we could implement a *DrawRectangle* class by inheriting from both *Rectangle* and *Drawable*
• Java only allows single inheritance
• Interfaces allows some of the flexibility of multiple inheritance without the problems with same signature methods in both inherited classes
interface Drawable {  
    public void setColor(Color r);  
    public void setPosition(double x, double y);  
    public void draw(DrawWindow w);  
}

class DrawableRectangle extends Rectangle implements Drawable {  
    private Color c;  
    private double x, y;  
    public DrawableRectangle(double w, double h) {super(w, h);}  
    // here are the implementations of the methods  
    // inherited from the interface Drawable:  
    public void setColor(Color c) {this.c = c;}  
    public void setPosition(double x, double y) {  
        this.x = x; this.y = y;  
    }  
    public void draw(DrawWindow dw) {  
        dw.drawRect(x, y, w, h, c);  
    }  
}
Interfaces and attributes

- In the previous example, could conceivably have an attribute (field in the class) that was a reference to a *drawable* object that handled the draw functions

  - Advantage: the drawable object might actually be able to implement many of the methods of the interface, saving classes from having to implement them

  - Disadvantage: if we need to treat circle, rectangle, etc., as a drawable objects, cannot do it

```java
Drawable d[] = new Drawable[4]; . . . initialize d
for (i=0; i < 4; i++) {d[i].setPosition = f(i);}
```
Multiple Interfaces/Constants

• Multiple interfaces can be implemented

• All abstract methods (i.e. all methods) declared in the interface must be implemented (defined)

  • That each method is defined in the implementing class removes any ambiguity as to which interface’s method is called - there is one implemented method that covers all interfaces!

• Constants can be declared in interfaces and the constants become visible to the implementing class

• The possibility of ambiguity exists with constants
Can variables be declared in interfaces?

- Yes, but . . .

- They must be a public static final variable

```java
public interface foo {
    int i = 0;
}
```

- The declaration above is the same as `public static int i = 0;` even if it is not explicitly declared as such.
What if a static final is declared twice and used?

interface Bar {
    static final int c = 10;
}

interface Foo {
    static final double c = 5.0;
}

class Test implements Foo, Bar {
    void main(String[ ] args) {
        lc = c;
    }
    int lc;
}

smidkiff% javac Test.java
Test.java:3: reference to c is ambiguous, both variable c in Foo and variable c in Bar match
    lc = c;
   ^
1 error
What if a static final is declared twice and not used?

interface Bar {
    static final int c = 10;
}

interface Foo {
    static final double c = 5.0;
}

class Test implements Foo, Bar {
    void main(String[] args) {
        lc = 10;
    }
    int lc;
}

smidkiff% !javac
javac Test.java
smidkiff%
Packages and access modifiers

• Often we have multiple files that work together to provide the same functionality

• For example, we could have a graphics package with Draggable interface, a shape abstract class, and Circle, Rectangle, Line and Point classes.

• They are more closely related than, e.g., a set of classes and interfaces that implements Math functions (sin, logarithmic functions, coordinate system transforms) and a set of classes and interfaces that implement home address validation or SSN validation.

• Packages give us a way to group closely related interfaces and classes together
Some classes in a package

// in the Draggable.java file
package graphics;
public interface Draggable {
...
}

// in the Graphic.java file
package graphics;
public abstract class Graphic {
...
}

// in the Circle.java file
package graphics;
public class Circle extends Graphic implements Draggable {
...
}

// in the Rectangle.java file
package graphics;
public class Rectangle extends Graphic implements Draggable {
...
}

// in the Point.java file
package graphics;
public class Point extends Graphic implements Draggable {
...
}

// in the Line.java file
package graphics;
public class Line extends Graphic implements Draggable {
...
}
Benefits of packages

- Being in the same package indicates that the classes are related
- Documents where, in this case, graphics related code can be found
- The package creates a namespace and you can have a graphics.Rectangle and a Rectangle class in another (perhaps unnamed) package
- Can allow extra access to variables to members of the package while being protected from access outside of the package
## Access Modifiers

<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>Can access</td>
<td>Can access</td>
<td>Can access</td>
<td>Can access</td>
</tr>
<tr>
<td>Protected</td>
<td>Can access</td>
<td>Can access</td>
<td>Can access</td>
<td>No</td>
</tr>
<tr>
<td>None</td>
<td>Can access</td>
<td>Can access</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Private</td>
<td>Can access</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>