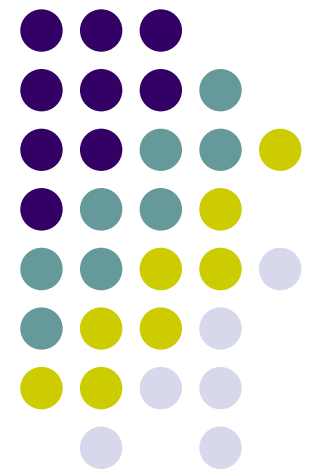
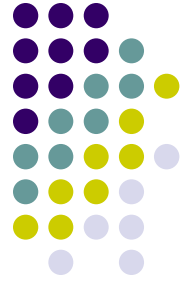


6.092: Thursday Lecture

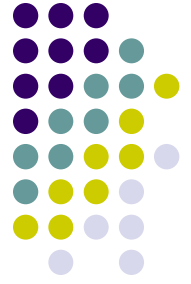
Lucy Mendel
MIT EECS





Topics

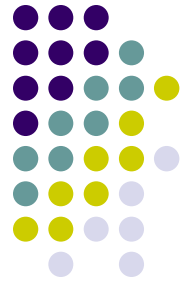
- Interfaces, abstract classes
- Exceptions
- Inner classes



Abstract Classes

- Use when subclasses have some code in common

```
abstract class Person {  
    private String name = "";  
    public String getName() { return name; }  
    public void setName(String n) { name=n; }  
    abstract public String sayGreeting();  
}  
  
class EnglishPerson extends Person {  
    public String sayGreeting() { return "Hello"; }  
}
```



Interfaces

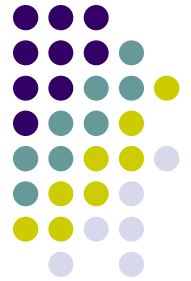
- Use to make distinct or unknown or unspecified components **plug-and-play**

```
public interface Dragable {  
    public void drag();  
}
```

```
public class Icon implements Dragable {  
    public void drag() { ... }  
}
```

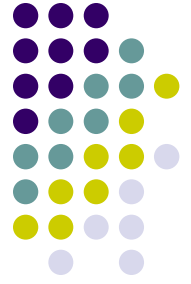
```
Public class Chair implements Dragable {  
    public void drag() { ... }  
}
```

Another reason to use Interfaces



```
Public abstract class Cowboy {  
    /*kill something*/  
    public void draw() {...}  
}
```

```
Public abstract class Curtain {  
    /*let in sunshine*/  
    public void draw() {...}  
}
```



Cowboy Curtain

```
public class CowboyCurtain extends
    Cowboy, Curtain {
    ...
}
```

```
CowboyCurtain cowboy = new CowboyCurtain();
Cowboy.draw();
// does it get brighter or does something die?
```

Multiple implements, single extends



```
interface Drawable {  
    public void draw();  
}
```

```
interface Clickable {  
    public void click();  
}
```

```
interface Draggable {  
    public void drag();  
}
```

class Icon implements
Drawable, Clickable, Draggable {

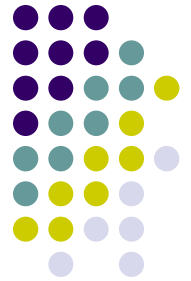
```
    public void draw() {  
        SOP("drawing..."); }  
    public void click() {  
        SOP("clicking..."); }  
    public void drag() {  
        SOP("dragging..."); }  
}
```



Subtyping

```
class Square {  
    public int width;  
}  
class Rectangle {  
    public int width,height;  
}  
...  
int calculateArea (Square x) {  
    return (x.width)*(x.width); }  
int calculateCircumference (Rectangle x) {  
    return 2*(x.width+x.height); }
```

Should:
Square extend Rectangle?
Rectangle extend Square?

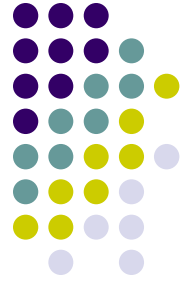


Rectangle extends Square

```
class Square {
    public int width;
    Square( int x ) { width = x; }
}

class Rectangle extends Square {
    public int height;
    Rectangle( int width, int height ) {
        super(width);
        this.height = height; }
}

...
Rectangle rect = new Rectangle( 2, 3 );
calculateArea( rect );    // returns 4, not 6 !
```



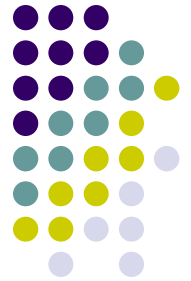
Square extends Rectangle

```
class Rectangle {  
    public int width, height;  
}
```

```
class Square extends Rectangle {  
    public int side;  
}
```

...

```
Square square = new Square( 3 );  
calculateCircumference( sq ); // w.t.f. no height!
```



Square extends Rectangle

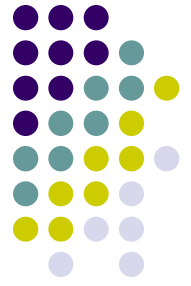
```
class Rectangle {
    public int width, height;
    Rectangle( int width, int height ) {
        this.width = width; this.height = height; }
}
class Square extends Rectangle {
    Square( int x ) { super( x, x ); }
}
...
Square square = new Square( 3 );
calculateCircumference( sq ); // 12, ok
```



True Subtyping

- Inheritance (extending classes) re-uses code
- A true subtype will behave the right way when used by code expecting its supertype.

```
class B {  
    Bicycle myMethod(Bicycle arg) {...}  
}  
class A {  
    RacingBicycle myMethod(Vehicle arg) {...}  
}
```

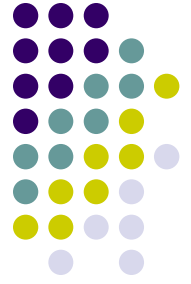


Composite

- Contain a class, rather than extend it

```
class ListSet { // might want to implement Set
    private List myList = new ArrayList();
    void add(Object o) {
        if (!myList.contains(o)) myList.add(o);
    }
}
```

...

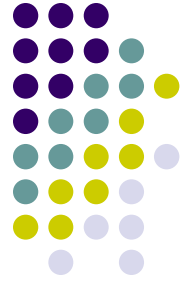


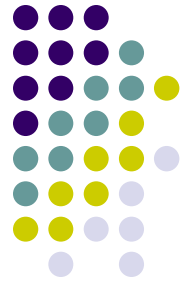
Exceptions

- Goal: help programmers report and handle errors
- What happens when an exception is thrown?
 - Normal program control flow halts
 - Runtime environment searches for handler:

```
try {  
    statement(s) that might throw exception  
} catch (exceptiontypeA name) {  
    handle or report exceptiontypeA  
} catch (exceptiontypeB name) {  
    handle or report exceptiontypeB  
} finally {  
    clean-up statement(s)  
}
```

Exceptions





Exception Catching

```
class Editor {  
    public boolean openFile( String filename ) {  
        try {  
            boolean fileOpen = true;  
            File f = new File(filename);  
            // do stuff with f  
            return true;  
        } catch (FileNotFoundException e) {  
            e.printStackTrace();  
            return false;  
        } finally {  
            fileOpen = false;  
        }  
        System.out.println(" bar ");  
    }  
}
```




Exception Throwing

```
public class File {  
    public File(String filename) throws  
        FileNotFoundException {  
        ...  
        if ( can't find file ) {  
            throw new FileNotFoundException();  
        }  
    }  
}
```

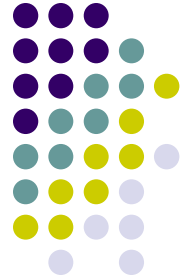


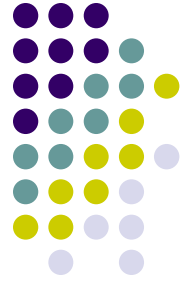
Nested (or Inner) Classes

```
class EnclosingClass {  
    ...  
    class ANestedClass {  
        ...  
    }  
    ...  
}
```

Why use nested classes?

Nested Classes



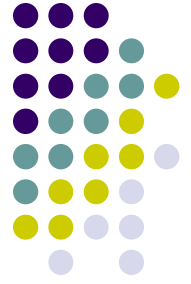


Nested Class Properties

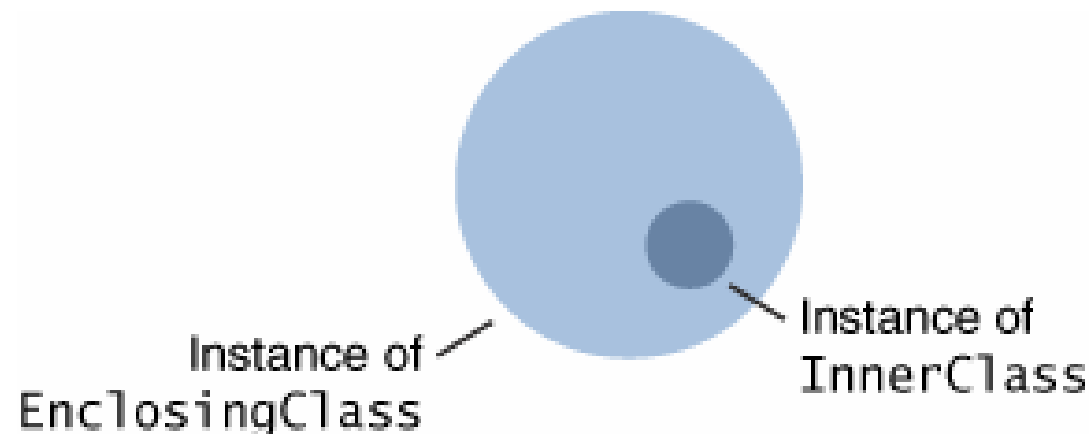
- Have access to **all** members of the enclosing class, even private members
- can be declared **static** (and final, abstract)
- Non-static (or instance) nested classes are called **inner classes**

```
class EnclosingClass {  
    static class StaticNestedClass { ... }  
    class InnerClass { ... }  
}
```

Inner Classes (non-static nested classes)



- Associated with an instance of the enclosing class
 - Cannot declare static members
 - Can only be instantiated within context of enclosing class

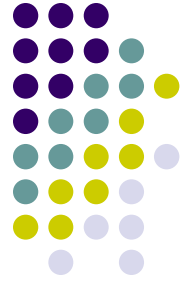


Local Anonymous Inner Class



```
public class Stack {
    private ArrayList items;
    public Iterator iterator() {
        private class StackIterator implements Iterator {
            int currentItem = items.size() - 1;
            public boolean hasNext() { ... }
            public ArrayList<Object> next() { ... }
            public void remove() { ... }
        }
        return new StackIterator();
    }
    /* or declare here */
}
```

Anonymous Inner Class



```
public class Stack {
    private ArrayList items;
    public Iterator iterator() {
        return new Iterator() {
            int currentItem = items.size() - 1;
            public boolean hasNext() { ... }
            public ArrayList<Object> next() { ... }
            public void remove() { ... }
        };
    }
}
```



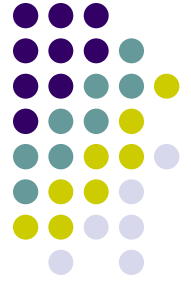
Floating Point Precision

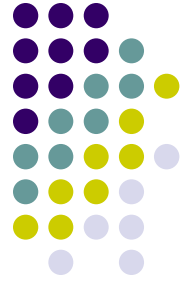
```
System.out.println(1.00 - .42);  
// 0.610000000000000000000001
```

```
System.out.println(1.00 - 9*.10) ;  
// 0.099999999999999999999995
```

- **BigDecimal**
 - Pain of using Objects
- **int or long**
 - keep track of decimal yourself, eg, put money in terms of pennies

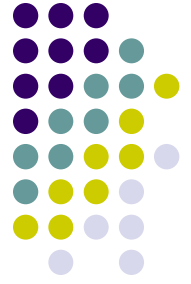
Defensive Programming





```
public class Man {
    private Wallet myWallet;
    public Money payPaperboy() {
        return myWallet.money;
    }
}

public class Paperboy {
    public Money payment;
    public void getPaid( Man m ) {
        payment += m.payPaperboy();
    }
}
```

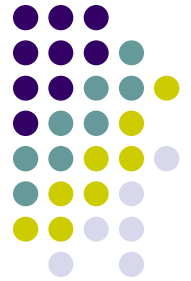


Minimize Accessibility

```
public class Man {
    public Wallet myWallet;
}

public class Paperboy {
    public Money payment;
    public void getPaid( Man m ) {
        payment += m.myWallet.money;
    }
}
```

Keep field references unique



- Copy parameters before assigning them to fields
- Copy fields before returning them

```
public final class Period {
    private final Date start;
    private final Date end;
    public Period(Date start, Date end) {
        if (start.compareTo(end) > 0)
            throw new IllegalArgumentException(start
                + " after " + end);
        this.start = start;
        this.end = end;
    }
    public Date start() { return start; }
    public Date end() { return end; }
}
```