Java Programming

Objects and Classes

1

Overview

- Object-Orientation
- Historical perspective
- Objects and classes
- Constructors
- Terminology

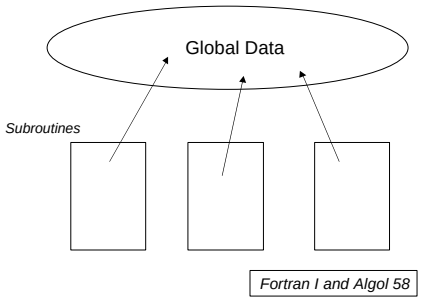
2

Object-Orientation

- OOP replaces the structured programming techniques of the early 70's
- In structured programming the focus was on the steps of the code
- With object-oriented programming it's entities with interfaces

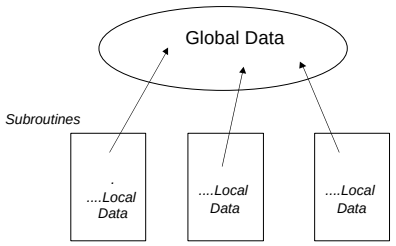
3

First Generation (1954-1958)



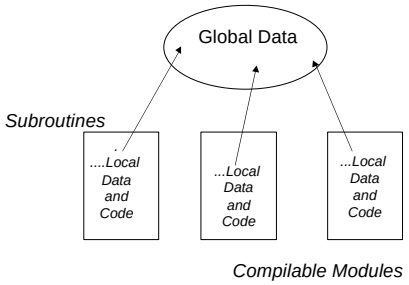
4

Second Generation (1960's)



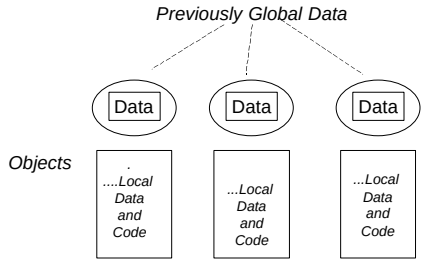
5

Third Generation Languages



6

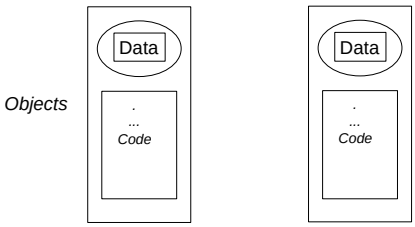
Object-Oriented Languages



7

Java is object-oriented

Everything is part of some object



8

Object Oriented Languages

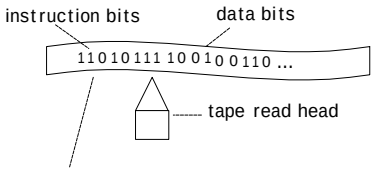
- Simula
- Smalltalk
- C++
- Ada95
- Eiffel
- Object Cobol
- Java

9

Objects and the Code-Data Dichotomy

10

Turing Machine



instructions and data encoded on infinite binary tape

11

THEORY

Turing Machine

von Neumann architecture

REAL

real computers: EDSAC, UNIVAC

assembly languages

Fortran Cobol

HIGH-LEVEL LANGUAGES

12

Language Evolution

CODE-Oriented Languages

Fortran → Algol → Pascal → Modula → Ada

C

DATA-Oriented Languages

Cobol → Cobol-74 → Cobol-85 → Cobol-85

SQL

13

Language Evolution

CODE-Oriented Languages

Fortran → Algol → Pascal → Modula → Ada → Ada-95

C → C++

DATA-Oriented Languages

Cobol → Cobol-74 → Cobol-85 → Cobol-85 → OO-Cobol

SQL → (JDBC)

Java

Object-Oriented Languages

Object Vocabulary

- Class
- Object
- Instance
- inheritance
- encapsulation
- messages

15

class, object, instance

- Class = the template from which objects are created

```
AudioClip meow = new AudioClip()
```

class name

object name

meow is an instance of the class AudioClip -- it's an object instance

16

class, object, instance

- Class = the template from which objects are created

```
AudioClip meow = new AudioClip()
```

constructor
(a special method)

17

inheritance

- A class may stand alone as in:

```
class Hello { …
```

- A class may inherit functionality:

```
class MyApplet extends Applet {…
```

- the class inherited from is called the base class

18

inheritance

- All classes in Java inherit functionality from the class Object

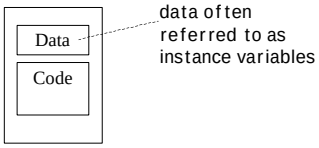
```
class Hello { …
```

extends Object

19

encapsulation

- data is encapsulated within objects
- also called data-hiding

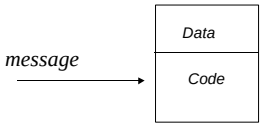


Object - a package of code and data

20

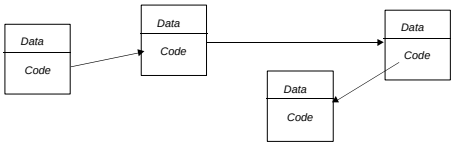
messages

- code is packaged in methods (old word is functions)
- we send messages to an object (to execute a method -- call a function)



21

An OO program



- Forces the allocation of functionality (code) to objects

Obejct-oriented programs are collections of objects sending messages to one another

22

what is an object?

23

Object Definition

– An object is something that has:
  - Behavior
  - State
  - Identity

24

Object Behavior

- All objects of the same class support the same behavior (methods) -
  - String s1 = “hello”;
  - String s2 = “java world”;
  - s1.length();
  - s2.length();

25

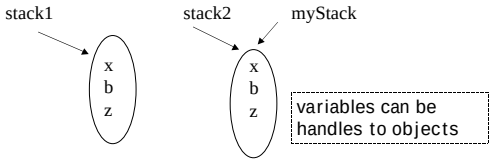
Object State

- The state of an object refers to the data an object contains
- The state of an object normally changes over time - as it’s data values change

26

Object Identity

- Each object has a unique identity, even if its state is identical to that of another object



27

Objects and Classes

- A class is a way to define commonality across a group of objects with:
  - common data (attributes or properties)
  - common behavior (methods or operations)
- An object is an instance of a class

28

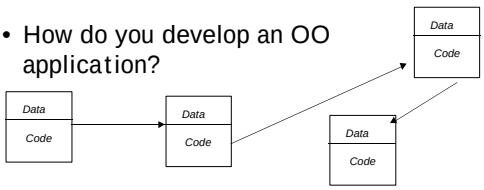
Style Guide for Naming Classes

- Classes are named using singular nouns
- Class names start with an upper case
- Underscores are not used
  - Names composed of multiple words are pushed together and the first letter of each additional word is capitalized
  - Example: Student, Professor, BillingSystem

29

Object-Oriented Design

- How do you develop an OO application?



The allocation of code and data to objects is a difficult design issue

30

Simple OO Design Strategy

- Nouns in the problem description often describe classes
  - Account, Customer, Budget
- Verbs often map to methods
  - compute overtime
  - cancel order
  - estimate project duration

31

Class Relationships

- Uses
  - sends a message to
- Containment (“has-a”)
  - one object defined as a variable reference within another object
- Inheritance (“is-a”)
  - one object extends another

32

Classes without instances

- A class may simply exist without instances
  - The Console class (Horstmann)
  - The Math class (java)
- Most common is using object instances of a class

33

Classes with Instances

Defining an object variables

```
AudioClip meow;
String s;
Rectangle r;
Socket sock;
```

NO objects exist yet!

class name

instance variable name

34

Object Instances

- Define variable

```
AudioClip meow;  //no object yet
meow.play();     //runtime error
```

- Create an object

```
meow = new AudioClip();
```

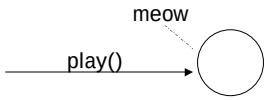
constructor (method)
--same name as class

35

Object Instances

- Create an object

```
AudioClip meow;
meow = new AudioClip();
meow.play();     // ok now
```

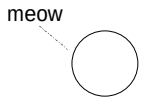


36

One Step Define/Create

- **One Step define/create**

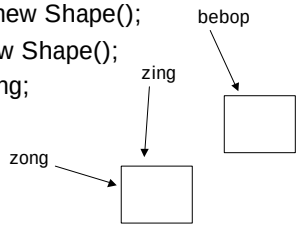
```
AudioClip meow = new AudioClip();
```



37

Multiple Object Instances

- Shape bebop, zing, zong;
- bebop = new Shape();
- zing = new Shape();
- zong = zing;

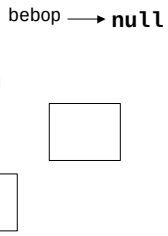


38

Null Object references

- bebop = null;
- if (bebop != null)
  bebop.show();

otherwise
runtime error



39

java.awt.Rectangle

40

java.awt.Rectangle class

public int x public int y public int width public int height	data
public boolean contains (int x, init y) public void setSize(int width, int ht) public String toString();	methods

```
Rectangle r1 = new Rectangle(); // init to all zeros
Rectangle r2 = new Rectangle(1,1, 10,10);
```

41

java.awt.Rectangle class

public int x public int y public int width public int height	data
public boolean contains (int x, init y) public void setSize(int width, int ht) public String toString();	methods

```
Rectangle r2 = new Rectangle(1,1, 10,10);
int k = r2.x + r2.y;     //x and y are public
boolean b = r2.contains(4,4);
```

42

Design Rule

- The data in your classes should be declared private
- This encourages information hiding
- Reduces effects of software changes

43

MyRectangle

private int x private int y private int width private int height	data (instance variables)
public boolean contains (int x, init y) public void setSize(int width, int ht) public String toString();	methods

Rectangle r3 = new Rectangle();
int k = r3.x + r3.y; //illegal

44

Options for working with private data members

1. Initialize

```
class Rectangle {
  private int x      = 20;
  private int y; //defaults to zero
  private int width  = 20;
  private int height = 40;

  public int computeArea () {
   return (width * this.height);
  }
}
```

optional

45

Working with private data members...

2. Accessors (get) & Mutators (set)

```
class Rectangle {
  private int x      = 20;
  private int y;
  private int width  = 20;
  private int height = 40;

  public int  getX () {return x;}
  public int  getY () {return y;}
  public void setY (int y) {
    this.y = y;
  }
```

46

MyRectangle with get & set

```
private int x
private int y
private int width
private int height
```

data (instance variables)

```
public int getX { return x;}
public int setX (int xin) { x = xin;}
pulbic int getWidth { return width; }
pulbic int setWidth(int w) {width = w;}
public boolean contains (int x, init y)
public void setSize(int width, int ht)
public String toString();
```

methods

47

Accessor Methods

- get and set methods that read and write private data
- example:

```
public int getWidth ( ) {...
```

capitalize variable name

having get but NOT set means read-only

48

Working with private data members...

3. Constructors

A class may specify several constructors to initialize objects

Rectangle r = new Rectangle (12, 33);
Rectangle s = new Rectangle (10,20, 15, 34);

compiler figures out which one by matching types

49

Constructor Methods

- Used with keyword new to create objects
- Have same name as the class
- May take parameters or have none
- Have NO return values

no return value

```
public Rectangle () {
  x=50; y=50;
}
```

50

Constructor Gotcha!

```
public void Rectangle () {
    x=50;
    y=50;
}
```

this is not a constructor. the compiler treats it as a regular method!

51

Object Creation and Constructors

- Rectangle r = new Rectangle(11,22);

1. The java runtime attempts to allocate memory for the new object
2. If insufficient free memory an **`OutOfMemoryError`** is thrown -- program aborts
3. If ok memory, the constructor is called to initialize new object

52

Default Constructors

53

Default no-arg Constructor

- All classes have a default no argument constructor

```
class Foo {
    int x;
}
```

initializes data to default values

Foo f = new Foo();

54

Constructor Gotcha!!

Rectangle r = new Rectangle (10, 20);
Rectangle s = new Rectangle (40, 50);
Rectangle t = new Rectangle ( );

You cannot use the default constructor if you define your own constructor!

If you want a no-arg constructor, you must explicitly define it!

55

this

- keyword used in non-static methods
- refers to object in which the method is used
- Example:

```
void setWidth (int width) {
  this.width = width;
}
```

instance variable

parameter

56

this...

- also used to refer to a constructor within the same class

```
public Rectangle (int x) {
  this(x, 100);
}
```

refers to Rectangle(int, int)

57

Constructor Design

- Less specific constructors should call more specific constructors

```
class Rectangle {
  ...

  public Rectangle (int x, int y) {
    this.x = x;  this.y = y;
  }

  public Rectangle ( ) {
    this(0, 0);
  }
```

58

method name overloading

```
class DataRenderer {
 void draw(String s) {
          . . .
 }
 void draw(int i) {
          . . .
 }
 void draw(float f) {
           . . .
 }
}
```

59

java.awt.Rectangle

- what data members?
- what constructors?
- what methods?

check out the documentation

60

Private Data

- Private data is not directly accessible to methods outside the class
- Private data is accessible to all methods within the class
  - all data objects within the class! (often a surprise but also the rule in c++)

61

not private within the class

```
class Rectangle {
  private int x;
  private int y;

  public boolean equals (Rectangle r)
  {
   return r.x == x && r.y == y;
  }
}
```

private data access

62

Passing Parameters

- Java parameters use pass by value"
- The value of the passed parameter is <u>copied</u> as the value of the parameter

```
class PassByValueTest {
   public static void main (String [] args) {
      double one = 1.0;
      System.out.println(one);
      halveIt(one);
      System.out.println(one); }
```

63

```
class PassByValueTest {
    public static void main (String [ ] args) {
        double one = 1.0;
        System.out.println(one);
        halveIt(one);
        System.out.println(one);
}

public static void halveIt (double arg) {
   arg = arg / 2.0;   // divide by two
    System.out.println(arg);
}
```

output:

1.0
0.5
1.0

64

Passing Object Parameters

- With object (reference) data types, an address is passed
- The receiving method has access to the object - the result is that an object parameter can be changed

65

```
class PassByValue2 {
    public static void main (String [ ] args) {

        Rectangle r = new Rectangle (20,20);
        System.out.println(r.x);
        moveX(r);
        System.out.println(r.x);
}

public static void moveX (Rectangle rect) {
        rect.x = rect.x / 2;   // divide x coordinate by two
         System.out.println(rect.x);
}
```

output:

20
10
10

the object reference is passed by value.

two object references (r and rect) refer to the same Rectangle object in memory

66

Class Employee

67

Employee

```
String name
String salary
Day hireDay
```

```
Employee (String n, double s, Day d)
void print()
void raiseSalary(double byPercent)
String getName()
Day getHireDay()
```

68

```
private String name;
private double salary;
private Day hireDay;
// accessor methods
public String getName { return name;}
public Day getHireDay {
    return hireDay;
```

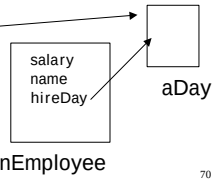
warning: returns an object reference!

69

Returning object references

- Violates encapsulation when the reference refers to private data
- A user can change the data via the object reference

```
Day dd =
 anEmployee.getHireDay();
dd.advance(20);
```



70

clone

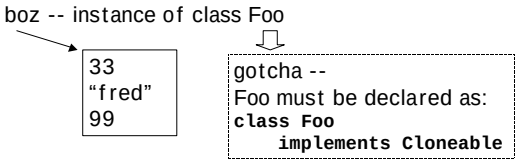
- All objects (potentially) can be cloned
- The method clone returns a copy of the object cloned

```
public Day getHireDay () {
  return (Day)hireDay.clone();
}
```

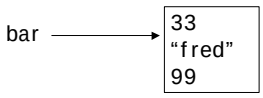
71

clone method

boz -- instance of class Foo



Foo bar = boz.clone();



72

Initialization Blocks

73

initialization block

- an arbitrary block may appear in a class definition -- outside a method
- the initialization block is executed before the constructor code
- initialization blocks are never necessary and almost always confusing!

74

initialization block…

```
class Rectangle {
  private int x;
  private int y;
   private int width;

  public Rectangle (int x, int y) {
   this.x = x; this.y = y;
  }

  {width = 50;}
}
```

initialization block

??

75

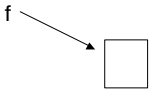
Garbage Collection and finalize

76

garbage collection

- objects no longer referenced are garbage collected -- their memory is returned to memory pool (the heap)

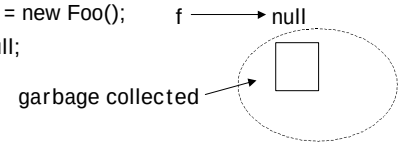
```
Foo f = new Foo();
f = null;
```



77

garbage collection

```
Foo f = new Foo();
f = null;
```



78

finalize method

- if you define a finalize method in your class, finalize() will be called before your object is garbage collected
- use finalize to free up system resources

79

Static Methods and Fields

80

Static Fields

```
Class MyClass {
 public static int BUFSIZE = 99;
 public static double [] buf =
                 new double[BUFSIZE];
```

how to initialize?

81

Static initializer

```
class MyClass {
 public static int BUFSIZE = 99;
 public static double [] buf =
                 new double[BUFSIZE];
 static {
  for(int i = 0; i < buf.length; i++)
    buf[i] = java.lang.Math.random();
}
```

82

Static methods

```
class MyClass {
 public static int BUFSIZE = 99;
 public static double [ ] buf =
                 new double[BUFSIZE];
 public int x = 10;
 private int y = 20;
 public static void foo () {
   … // can only access static vars & methods
}
```

83

Common use of main()

```
class MyClass {
 public static int BAR = 99;
 public void foo () {….}

 public static void main (String [] args) {
   MyClass myC = new MyClass();
   myC.foo();  // test an instance
 }
}
```

84

private methods

- Methods may be declared
  - public
  - private
  - protected -- visible to subclasses
  - "no keyword" -- visible to all classes in your package
- Use private for those methods:
  - not directly useful to class users
  - likely to change

85

Summary

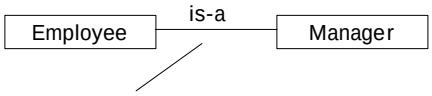
- Classes are the building blocks of Java
- Object instances are created using class constructors
- Classes may have static methods and fields
- Object instances have non-static methods and fields
- Rule: keep data private; use accessor methods

86

Inheritance

87

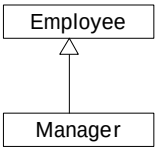
relationship?



is-a relationship
indicates inheritance

88

inheritance



superclass

subclass

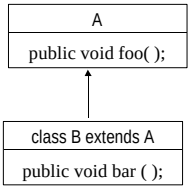
class Manager extends Employee

note: subclasses have more
functionality thant he superclass

89

Inheritance

```
B myB = new B();
myB.bar ();
myB.foo();
```



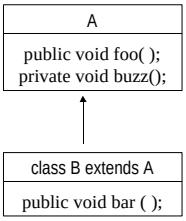
90

Inheritance - what about buzz?

B myB = new B();
myB.bar ();
myB.foo();

myB.buzz(); //??

NOT allowed!
***buzz** is private, visible only to methods within class A*



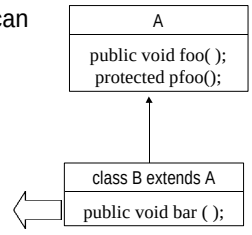
91

protected

protected means that code in a subclass can use it

A myA = new A();
B myB = new B( );
myA.pfoo( ); // legal ??
myB.pfoo ( ); // legal ??

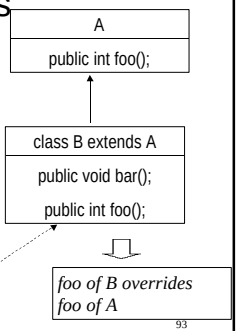
```
public void bar ( ) {
 foo ( );
 …...
 pfoo ( );
}
```



92

Inheritance - superclass & subclass

- The subclass inherits all public and protected fields and methods of the superclass
- What if two methods have the same name and parameter types??



*the subclass method **overrides** the superclass method*

B myB = new B();
myB.foo();

93

subclass constructor

```
public Manager (Sring n, double n, Day d)
{    super(n,d,s);
     secretaryName = "";
}
```

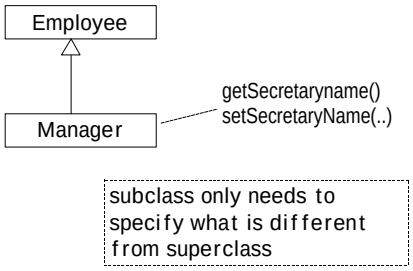
call to superclass constructor - must be first statement

94

Required: call to superclass constructor

- Subclasses always call their superclass constructor - either
- Explicitly
  - via super(..) in constructor
- Implicitly
  - system will call no-arg constructor
- Gotcha!! -- your no-arg constructor may not exist
  - when??

95



96

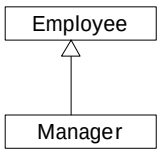
subclass (Manager) method

```
public void raiseSalary(double byPercent)
 {  // add 1/2% bonus for every year of service
    Day today = new Day();
    double bonus =
              0.5 * (today.getYear()- hireYear());
    super.raiseSalary(byPercent + bonus);
 }
```

superclass call

97

Inheritance and Typing



Because Manager is-a Employee, it can be used when the TYPE is Employee

98

Type safe assigment

```
Employee ed =
  new Manager("edward koch", 80000,
                    new Day(1999,9,9) );
```

can only use Employee methods on object "ed"

99

Arrays (of the superclass)

```
Employee[] staff = new Employee[20];
staff[0] = ed; // a manager object
staff[1] = new Employee ("bob", 45000,
   new Day(1996, 9, 9));
staff[2] = new Manager ("dahlia",
   65000, new Day (1997, 8,8) );
....
```

100

polymorphism

```
for (i=0; i< staff.length; i++) {
  staff[i].raiseSalary(8);
}
```

Employee objects have their methods called and Manager objects have their methods called

⇨ Late Binding

101

Late vs Static Binding

- Late Binding
  - the actual method that gets called is determined at run-time
- Static Binding
  - the method to be called is determined at compile time

102

Class Types

```
Manager ed =
  new Manager("edd", 20000,
                    new Day (1999,10,20) );
Employee[] staff = new Employee[20];
staff[0] = ed; // a manager object
...
```

```
edd.getSecretaryName()        -- OK!
staff[0].getSecretaryName() -- NO!!
```

103

Casting with Class Types

```
Manager ed =
  new Manager("edd", 20000,
                    new Day (1999,10,20) );
Employee[] staff = new Employee[20];
staff[0] = ed; // a manager object
...
Manager topGun = (Manager)staff[0];
```

```
topGun.getSecretaryName()   -- OK!
staff[0].getSecretaryName() -- NO!!
```

104

Casting

```
topGun.getSecretaryName()   -- OK!
staff[0].getSecretaryName() -- NO!!
```

⇧

The compiler checks that you do not promise too much

105

Bad Cast

```
staff[3] =
  new Employee("bob", 35000,
        new Day(1998, 12, 25));
Manager bogus = (Manager)staff[3];
bogus.getSecretaryName();
```

⇩

throws an Exception

106

Ridiculous Cast

```
staff[3] =
  new Employee("bob", 35000,
        new Day(1998, 12, 25));
Manager bogus = (Rectangle)staff[3];
```

⇩

compiler won't allow it!
Rectangle not in the Employee hierarchy

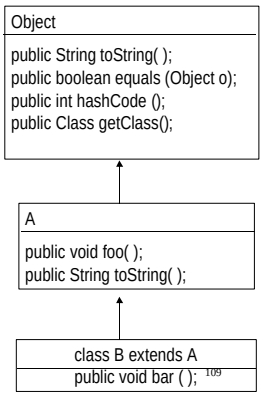
107

Protect with instanceof

```
staff[3] =
  new Employee("bob", 35000,
        new Day(1998, 12, 25));

if (staff[3] instanceof Manager) {
   Manager bogus = (Manager)staff[3];
   bogus.getSecretaryName();
}
```

108

Object - the ultimate superclass



Classes often override **`toString()`** and **`equals( )`**

109

method signature

return type *agument list*

```
public Point push (Rectangle r)
```

access level

public
private
protected
package

method name

110

member access

- Public
  - Any class has access;
- Private
  - most restrictive; accessible only within same class; good object-oriented practice is to define private data members
  - Protected
  - The class methods and subclass methods can access protected members
- Package
  - visible within the package

111

Default = package visibility

filename = A.java

```
class A {
 int x = 20;
 int foo( ) { return x*2; }
 public static void main (String args[ ] {
   B myB = new B();
   System.out.println
              ( myB.bar ( new A() ) );
}}

class B {
 int y;
 int bar (A myA ) {
  return (myA.foo() * myA.x );
}}
```

package visibility means public for all classes in the package

112

Package

- What you get if you don't specify
  - public, private or protected
- Allows classes in the same package to access members.
- Assumes that classes in the same package are "trusted friends"

113

Multiple classes in ONE file

filename = A.java

```
public class A {
 int x = 20;
 int foo( ) { return x*2; }
 public static void main (String args[ ] ) {
   B myB = new B();
   System.out.println ( myB.bar ( new A() ) );
}}

class B {
 int y;
 int bar (A myA ) {
  return (myA.foo() * myA.x );
}}
```

⇦ OK!

114

Only ONE can be public

filename = A.java

```
public class A {
 int x = 20;
 int foo( ) { return x*2; }
 public static void main (String args[ ] ) {
   B myB = new B();
   System.out.println ( myB.bar ( new A() ) );
}}

public class B {
 int y;
 int bar (A myA ) {
  return (myA.foo() * myA.x );
}}
```

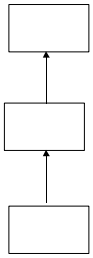
a public class must be in a file of the same name!!

NOT ALLOWED!

115

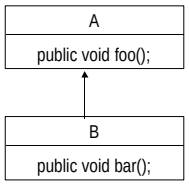
Subclasses & Contracts

- A class is extended by subclassing
  - adding new data attributes and methods to a subclass
- The collection of methods and fields accessible from outside a class + the class documentation defines the class' <u>contract</u>



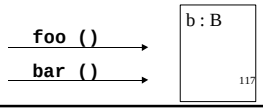
116

Class Contract



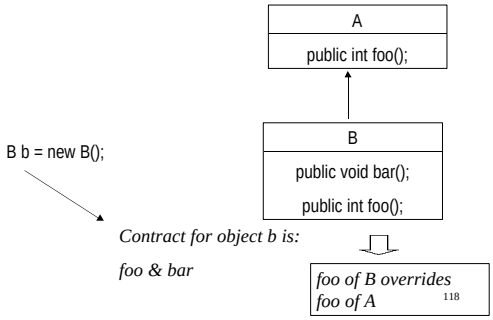
B b = new B();

Contract for object b is: foo & bar



117

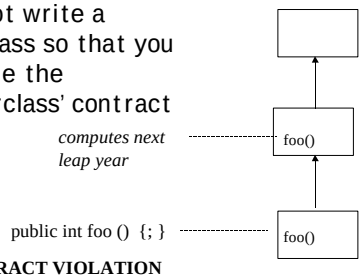
Polymorphism



118

Be true to your superclass

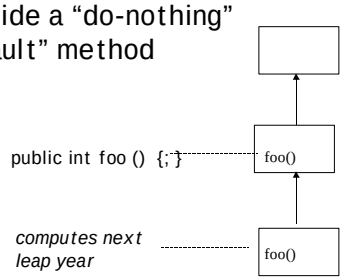
- Do not write a subclass so that you violate the superclass' contract



CONTRACT VIOLATION

119

It is OC (object-correct) to override a "do-nothing" or "default" method



NOT A CONTRACT VIOLATION

120

Example: overriding Object's **`equals( )`**

true if the two object references point to the same object

Object
public boolean equals(Object o);

MyRectangle
public MyRectangle(int x, int y, int w, int h);

```
MyRectangle r1 = new MyRectangle (10, 20, 30, 40);
MyRectangle r2 = new MyRectangle (10, 20, 30, 40);
false <-- r1.equals(r2);
```

121

Overriding Object's **`equals( )`**

```
if ((this.x==r.x) &&
    (this.y==r.y) &&
    (this.w==r.w) &&
    (this.h==r.h))
     then return true
     else return false;
```

Object
public boolean equals( );

MyRectangle
public MyRectangle(int x, int y, int w, int h); public boolean equals(MyRectangle r);

```
MyRectangle r1 = new MyRectangle (10, 20, 30, 40);
MyRectangle r2 = new MyRectangle (10, 20, 30, 40);
true <-- r1.equals(r2);
```

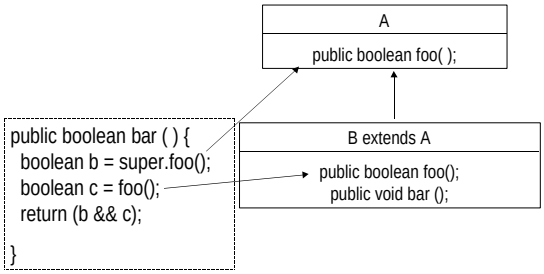
122

java.awt.Rectangle

- Overrides:
  - equals
    - returns true if two Rectangles have same x,y, height and width
  - toString
    - displays useful description
      - **java.awt.Rectangle[x=10,y=20,width=30,height=40]**
    - default (from Object) is:
      - Employee@afc6fab8

123

using super in methods



124

Class declarations

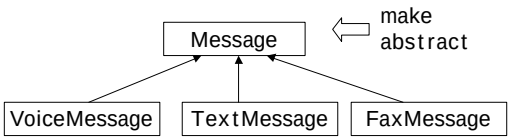
- **`class Rectangle {`**
  - visible to other class in the same package
- **`public class Rectangle {`**
  - visible to any class that imports Rectangle's package or is part of that package
- **`public final class Rectangle {`**
  - the class cannot be subclassed
- **`public abstract class Rectangle {`**
  - cannot create instances of the class
  - Rectangle r = new Rectangle() ; // ILLEGAL

125

abstract classes

126

Hierarchy Structure



we want all subclasses to have certain behavior -- each implemented its own way

127

abstract class

```
public abstract class Message {
  ...
  public abstract void play();
}
```

you cannot provide code in this class

128

abstract class

```
public abstract class Message {
  ...
  public abstract void play();
}
```

if you declare one method abstract you must declare the class abstract

129

abstract class

```
public abstract class Message {
  ...
  public abstract void play();
}
```

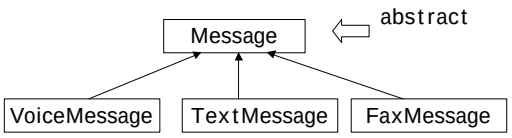
you cannot create an instance of an abstract class

130

```
public abstract class Message {
  private String sender;
  public Message (String from) {
    sender = from;
  }
  public abstract void play();
  public String getSender() {
   return sender;
  }
}
```

abstract class can have non-abstract methods and data

131



all non-abstract subclasses MUST implement any abstract methods defined in Message class

132

Abstract Classes

- A class may be declared abstract even though it has no abstract methods
- Abstract classes cannot be instantiated
- Variables can be declared as abstract class types but must refer to instance of concrete class

133

example

- Message m1 = new TextMessage("hi");

must conform to interface defined in Message class -- which is what we want in our design!

134

TextMessage classes

```
class TextMessage extends Message {
 private String text;

 public TextMessage(String from, String t) {
   super(from);
   text = t;
 }

   public void play( ) { System.out.println(text); }
}
```

135

Vectors

136

the class Vector

- An "array" of objects that grows when necessary
- VERY useful when you don't know in advance how many elements you'll need
- To use: **`import java.util.*;`**

137

Vector

- Does <u>not</u> store primitive types (int, float, etc.)
- For primitive types use Wrapper classes
- Vector stores only instances of Object
- Vector elements can be null

138

Vector Declaration

```
Vector v;
```

But more common is...

139

Creating a Vector instance

Vector v = new Vector( );

140

Creating a Vector instance

Vector v = new Vector( );

Declaration *Allocation & Assignment*

Vector has initial capacity of 10

It's size (number of actual elements) is 0

141

Creating a Vector instance

Vector v = new Vector(20);

Constructor

creates a Vector with 20 object capacity

142

Creating a Vector instance

Vector v = new Vector(20,10);

Constructor

creates a Vector for 20 objects and when more space is needed, allocates new slots 10 at a time

143

addElement

- add an element
  - **void addElement(Object obj);**

or any subclass

```
Vector v = new Vector (20);
v.addElement( new Employee () );
v.addElement( new Manager()   );
```

144

elementAt

- add an element
  - **Object elementAt(int idx);**

```
Vector v = new Vector (20);
v.addElement( new Employee () );
v.addElement( new Manager()   );
Object myObj = v.elementAt(0); // OK!

Object myObj = v.elementAt(8); // NO!
```

Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 8 >= 2

145

elementAt returns TYPE Object

```
Employee ed = v.elementAt(0);
```

ILLEGAL!

```
Employee ed =(Employee)v.elementAt(0);
```

OK with CAST

but if you're wrong -- will throw EXCEPTION

146

Bad Cast is Possible

```
Manager ed =(Manager)v.elementAt(0);
```

Exception in thread "main"
java.lang.ClassCastException: Employee
at VectorTest.main(VectorTest.java:24)

147

instanceof

```
if (v.elementAt(0)
        instanceof Employee) {
  Manager ed
       =(Manager)v.elementAt(0);
  ….
}
```

148

Vector Methods

- boolean contains (Object obj)
  - determines if an object is in the vector
  - the two object references must refer to the SAME object
- Object removeElementAt (int idx)
  - removes the element at the specified index
  - if idx is not valid, throws ArrayIndexOutOfBounds exception

149

Vector Methods…

- boolean isEmpty ()
  - true if the vector contains no elments
- int capacity ()
  - how many elements can the Vector hold before expansion is necessary

Vector v = new Vector(20,10);

⇨ size=0, capacity=20

150

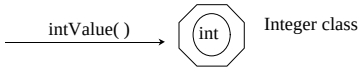
Vector Iteration

```
Vector vec = new Vector();
……
for (int i=0; i< vec.size(); i++)
  System.out.println(vec.ElementsAt(i));
```

151

Vectors and Primitive Types

- You can't create a vector of ints, floats or any other Java primitive type
- Java has wrapper classes specifically to turn your favorite primitive into an object



152

Wrapper Classes (one for each primitive type)

- Integer
- Float
- Double
- Long
- Character
- …

153

Using Wrapper Classes

- Vector v = new Vector( );
- Integer myInt = new Integer (33);
- v.addElement (myInt);

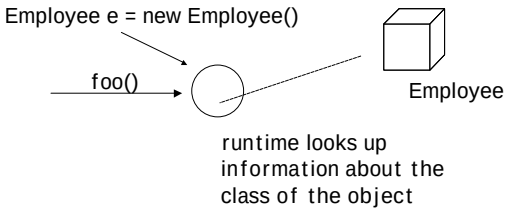
- Object oz = v.elementAt(0);
- int k = ((Integer)oz).intValue( );

154

The class Class

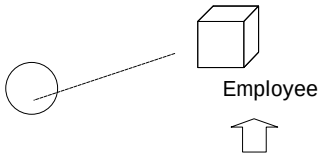
155

the java runtime



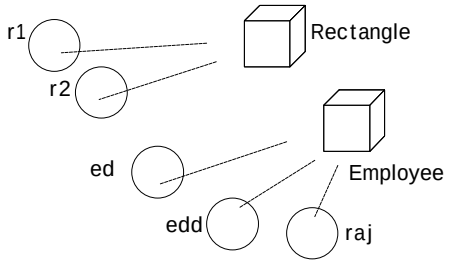
156

the class Class



programmers have access to this information about the class Employee thru the class Class

157

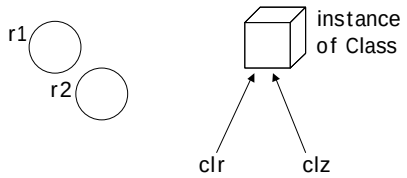


ONLY one instance of the class Class exists for each class in your program

you don't create it -- you access it!

158

use the getClass method to obtain a reference to a Class instance



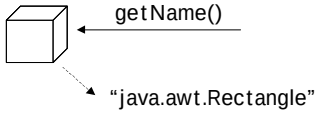

```
Class clr = r1.getClass();
Class clz = r2.getClass();
```

159

getName

- A class object can tell you its name

Rectangle class (of java.awt)



160

Create a Class instance using its name

- Class myc = Class.forName("Employee");

Employee class object

Will throw Exception if no such class exists

161

Create an Employee

- Class myC = Class.forName("Employee");

Employee class object

instance of Employee

- Employee raj = myC.newInstance();

must have no-arg constructor

162

Create instance from a String

- String s = “Manager”
- Object m
  = Class.forName(s).newInstance();

returns TYPE Object -- must cast to get Manager functionality

163

Other Class methods

- isInstance(Object obj)
  - returns true if obj is an instance of the class
- isAssignableFrom(Class other)
  - returns true if instance *other* is a subclass of the class receiving the message

164

Reflection

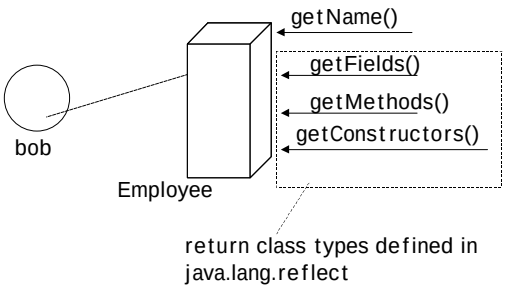
165

Reflection

- Reflective programs analyze the properties of other programs and classes
- What are the public methods of a class?
- What are the public fields?
- What are the constructors?

166

the class Class

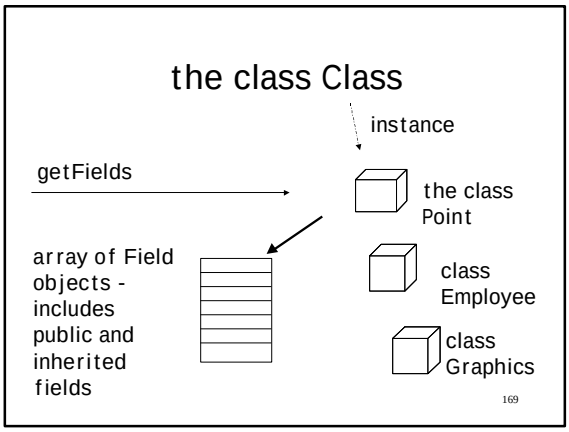


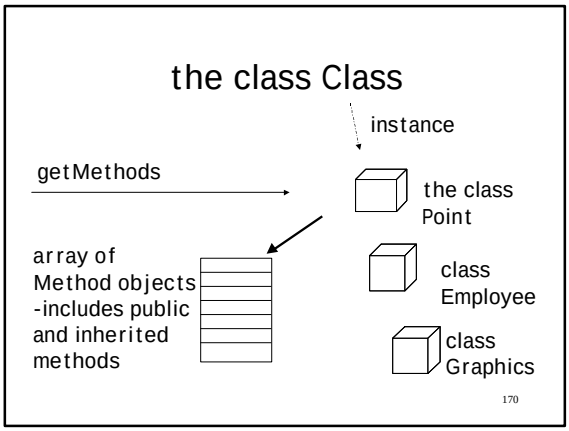
167

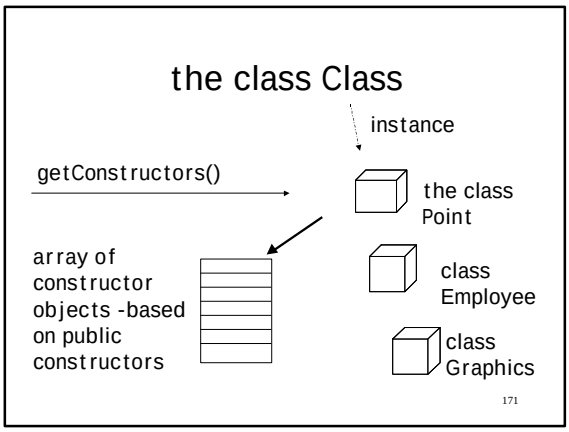
java.lang.reflect.*

- Field
  - a class that knows about fields
- Constructor
  - a class that knows about constructors
- Method
  - a class that knows about methods

168


the class Class
instance
getFields
the class Point
array of Field objects - includes public and inherited fields
class Employee
class Graphics
169


the class Class
instance
getMethods
the class Point
array of Method objects -includes public and inherited methods
class Employee
class Graphics
170


the class Class
instance
getConstructors()
the class Point
array of constructor objects -based on public constructors
class Employee
class Graphics
171

Basic Exception Handling

172

What's an Exception

- A signal that indicates an *exceptional condition* (something unexpected) has happened in your program
- To *throw an exception* is to signal that an exceptional condition has occurred
- To *catch an exception* is to handle the exception - to take whatever action is necessary
  - *sometimes you can't do anything*

173

Why Exceptions?

- Exceptions allow the programmer to treat error conditions outside the main logic flow
- Most programming languages (without exceptions) handle errors by passing return codes as error indicators

174

example

```
public class ExceptionTest {
  public static void main
            (String [] args) {
    int k = 20;
    int m,j;

    m = k/j;
    System.out.println
            ("m is " + m);
  }
}
```

Div by Zero

Exception in thread main:
java.lang.ArithmeticException

175

exception handling

```
int m=0;
try {
       m = k/j;
}
catch (ArithmeticException e) { }

System.out.println("m is " + m);
```

176

Handling Exceptions the complete story

```
try {
  // code that might
  // throw an exception
} catch (ExceptionType variable) {
   // handle the exception if thrown
} finally {
  // .. always do this
}
```

177

Threads

178

What is a Thread?

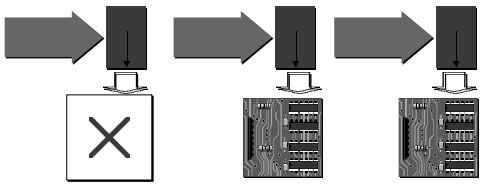
a single flow of control
within a program

sequential programs
have **one** thread

179

A Multi-Threaded Program

multiple flows of control

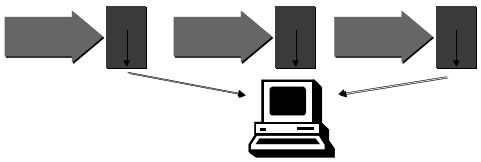


may run on multiple CPUs

180

A Multi-Threaded Program

multiple flows of control



may run on one CPU with time-sharing OS

181

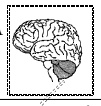
Multiple Threads in Browsers

Moonscape v 1.0

Java Net Home Page

Do all assignments and

you will be a JAVA



graphics load in a separate thread

182

Why Threads?

- Provides parallel computation with low overhead
- Use threads when a program may need to wait for some resource
  - disk access, network connection
- When one thread is waiting, other can continue processing

183

Threads vs Processes

- A process has its own address space
  - In a multitasking operating system, each program is run as a separate process
  - Process switching has overhead
- A thread shares the address space of the the program that created it
  - minimal overhead with thread switching

184

the “main” thread

- All java applications have a “main” thread -- started up when the static main method is executed
- To put the current thread to sleep send the sleep message to the Thread class as in:
  - Thread.sleep(millisecs)

185

Thread.sleep( )

```
for (int i=0; i<100; i++) {

  if (i == 10)
    try { Thread.sleep(2000); }
    catch (Exception e) {}

  else System.out.println(i);
}
```

186

Interfaces

187

Java Interface

- An alternative to abstract classes
- An interface specifies only method signatures:
  - method name
  - return value
  - parameters and types
- Abstract class can define:
  - data variables
  - concrete methods

188

interface

```
public interface Drawable {
  public void setColor(Color c);
  public void setPosition(double x, double y);
  public void draw(Graphics g);
}
```

189

interface

Keyword
OPTIONAL!

```
public abstract interface Drawable {
  public void setColor(Color c);
  public void setPosition(double x, double y);
  public void draw(Graphics g);
}
```

190

classes implement interfaces

```
public class Triangle implements  Drawable {
  public void setColor(Color c) {
    // code;
  }
  public void setPosition(double x, double y) {
    ....;}
  public void draw(Graphics g) {
    ....;}
}
```

191

extend only one class implement multiple classes

```
public class Triangle extends Shape
   implements  Drawable, Serializable {
  ....
}
```

the class Triangle must implement all methods in all interfaces -- but they can be as simple as { }

192

Interface as Data Type

```
Drawable myShape;
myShape = new Triangle();

Drawable [] shapes = new Drawable[5];
shapes[0] = new Triangle();
shapes[1] = new Circle();
```

Assumes Circle and Triangle both implement Drawable

193

```
Drawable [] shapes = new Drawable[5];
shapes[0] = new Triangle();
shapes[1] = new Circle();

shapes[1].setColor(Color.blue);
a1 = shapes[1].area();  // NOT OK!!
```

Can only execute methods defined as part of the interface

194

interface variables
(rarely seen)

```
public interface Drawable {
  private static final prefColor = Color.red;

  public void setColor(Color c);
  public void setPosition(double x, double y);
  public void draw(Graphics g);
}
```

Only static final variables are allowed in an interface

195

The Cloneable Interface

196

Copy vs Clone

- COPY
  Day bday = new Day(1960, 12,1);
- Day d = bday;
- d.advance(100);
  - // both are advanced! the two references refer to the same object

197

Clone

- Some classes implement the Cloneable interface - e.g. Day
- Day bday = new Day(1960,12,1);
- Day d = (Day)bday.clone();
- d.advance(100);

198

the method clone()

- Defined as a protected method in class Object
- It will copy all the data items and references into a new object
  - the actual references will remain the same
- If you want otherwise you must override clone() in your class

199

Interfaces and Callbacks

200

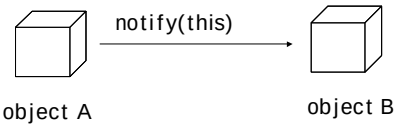
notation

object A

an instance of class A

201

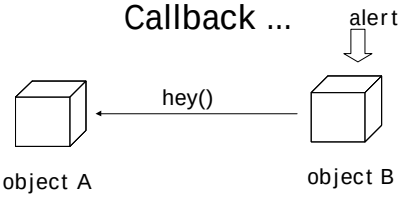
Callback Scenario



object A sends a message to object B -- please notify me (call me back) sometime in the future

202

Callback ...



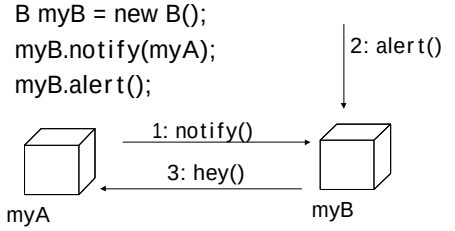
at some time in the future some event triggers B to send a message to A

203

```
class B {
  A myContact;

  void notify (A someObj) {
   myContact = someObj;
  }

  void alert () {
   myContact.hey();
  }
}

class A {
 void hey()
 {
    System.out.println("got the message");
 }
}
```

204

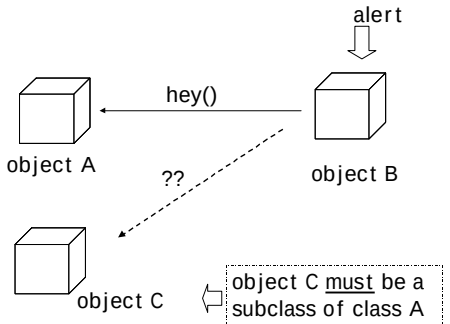
```
public class CallBackTest2 {
 public static void main (String [] args) {
   A myA = new A();
   B myB = new B();
   myB.notify(myA);
   myB.alert();
}
```



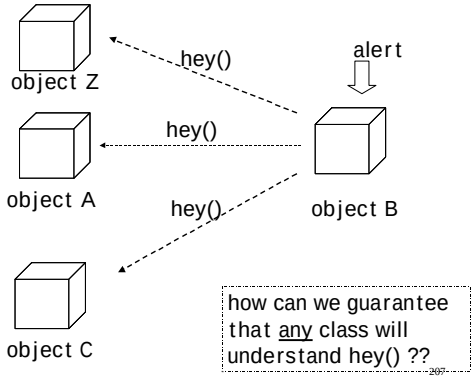
this works BUT….

205

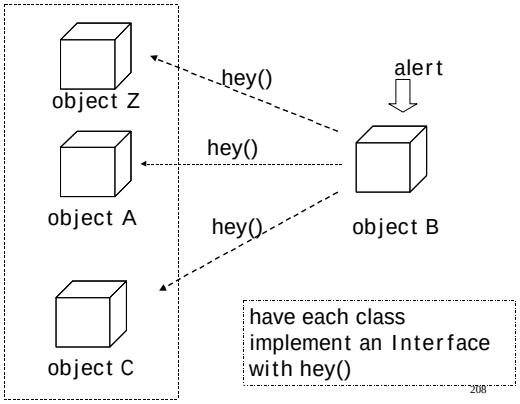
what about other classes?



206



207



208

KnowHey Interface

```
public interface KnowHey {
 public void hey ();
}
```

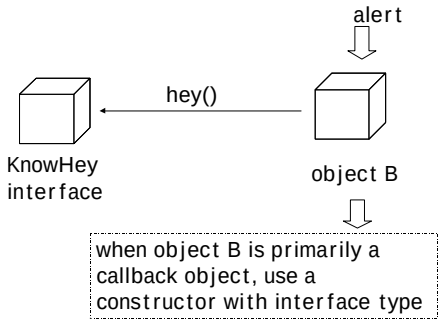
209

```
class B {
  KnowHey myContact;

  void notify (KnowHey someObj) {
   myContact = someObj;
  }

  void alert () {
   myContact.hey();
  }
}

class A implements KnowHey{
 public void hey()
 {
    System.out.println("got the message");
 }
}
```

class B can call back any object that implements the KnowHey interface

210

Constructor with interface type



211

```
class B {
  KnowHey myContact;

  public B (KnowHey obj) {
    myContact = obj;
  }

  void alert () {
     myContact.hey();
  }
}
```

constructor handles registration of object for callback

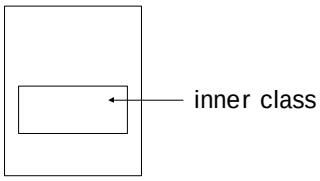
212

Inner Classes

213

Inner class

- A class defined within another class



214

Why Inner classes?

- Objects of the inner class can freely reference the data of enclosing class (incl private data)
- An inner class is hidden from other classes in the same package
- Anonymous inner classes useful for callbacks
- Convenient for event-driven programs

215

Sleeping Threads

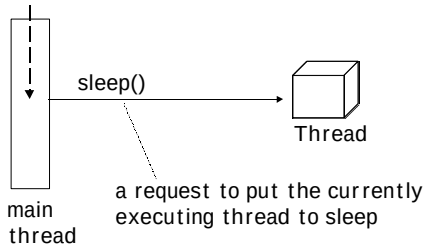
216

Thread.sleep( )

```
for (int i=0; i<100; i++) {

  if (i == 10)
    try { Thread.sleep(2000); }
    catch (Exception e) {}

  else System.out.println(i);
}
```

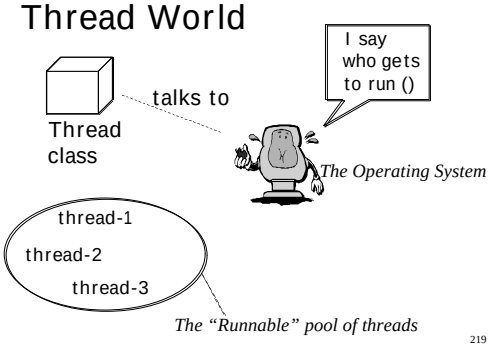
217

How does Thread.sleep() work?



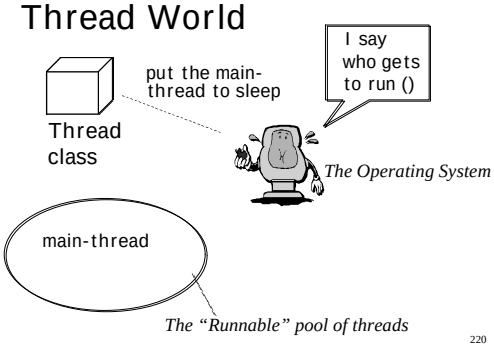
218

Thread World



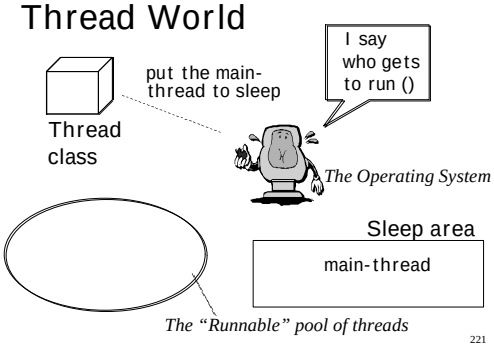
219

Thread World



220

Thread World



221

Creating a Thread with the run ( ) method

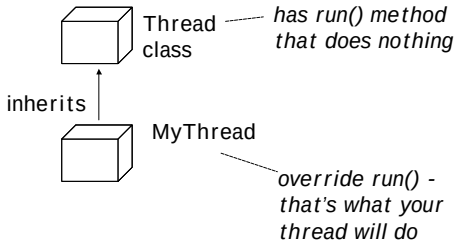
- Two ways to create a Java thread:
  - Extend the Thread class and override the run () method
  - Implement the Runnable interface
    - define run ( )

either way, there is a thread object

whatever code is in run ( ) will be run as a separate thread

222

Subclass Thread



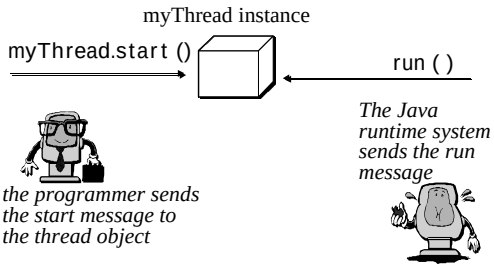
223

Subclassing Thread

```
class SimpleThread extends Thread {
   private String internalName;

   SimpleThread (String name) {
      internalName = name;
   }

public void run() {

  for (int i=0; i<5; i++) {
          System.out.println(internalName);
      }
   }
}
```

224

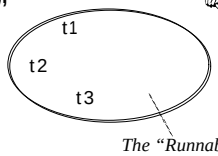
Activating your thread



225

```
public class SimpleThreadTest1 {
  public static void main (String argv[]) {
    SimpleThread t1 = new SimpleThread("sun");
    SimpleThread t2 = new SimpleThread("java");
    SimpleThread t3 = new SimpleThread("beans");

    t1.start();
    t2.start();
    t3.start();
  }
}
```

The Operating System



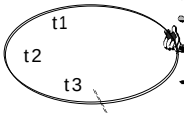
(Continued ➔)

The "Runnable" pool of threads

226

```
class SimpleThread extends Thread {
  String internalName;

  public void run() {

    for (int i=0; i<5; i++) {
      System.out.println(internalName);
    }
  }
}
```

Don't try to out-guess me!



The "Runnable" pool of threads

Output:
sun
sun
java
java
java
sun
beans
beans
sun
java
java
beans
sun
beans
beans

227

Creating our own thread that goes to sleep

228

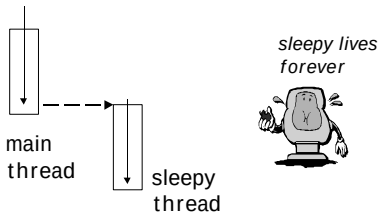
```
class Sleepy extends Thread {

  public void run ()  {
    while (true) {
     try { sleep (2000);
     }
     catch (Exception e) { }

     System.out.println("good morning");
    }
  }
```

since we're a thread we can put ourself to sleep

```
public static void main(String [] args) {
        Sleepy zzz = new Sleepy();
        zzz.start();
}
```

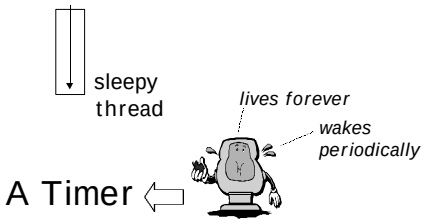
229

Multi-Threaded Program (kind-of)



230

Sleepy -- a closer look



231

Timer

Timer
thread

when it wakes up we want it
to notify another object

232

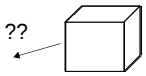
Object-Oriented Design Problem

```
class Timer extends Thread {

  public void run ()  {
    while (true) {
     try { sleep (2000);
     }
     catch (Exception e) { }

     System.out.println("good morning");

      // --- how to notify someone else ??
    }
  }
```

233

Design Problem

FooBar



Timer

Timer needs a
foobar reference

we want to alert a FooBar
object every 2 seconds

234

Timer has its own foobar

```
class Timer extends Thread {
  FooBar foobar;

  public void run () {
    while (true) {
      try { sleep (2000);
      }
      catch (Exception e) { }

      foobar.wakeup();

    }
}
```

who's foobar??

235

```
class Timer extends Thread {
  FooBar foobar;

  public Timer (FooBar f) { foobar =f; }

  public void run () {
    while (true) {
      try { sleep (2000);
      }
      catch (Exception e) { }

      foobar.wakeup();

    }
}
```

use constructor to set internal reference

OR...

236

```
class Timer extends Thread {
  FooBar foobar;

  public void register (FooBar f) { foobar =f; }

  public void run () {
    while (true) {
      try { sleep (2000);
      }
      catch (Exception e) { }

      foobar.wakeup();

    }
}
```

use method to set internal reference

237

```
class Timer extends Thread {
  FooBar foobar;

  public void register (FooBar f) { foobar =f; }

  public void run () {
    while (true) {
      try { sleep (2000);
      }
      catch (Exception e) { }

      foobar.wakeup();

    }
}
```

implications for class FooBar?

238

```
class Timer extends Thread {
  FooBar foobar;

  public void register (FooBar f) { foobar =f; }

  public void run () {
    while (true) {
      try { sleep (2000);
      }
      catch (Exception e) { }

      foobar.wakeup();

    }
}
```

implications for class Timer?

Timer ONLY works with FooBars

239

to make a Timer work with many different classes requires the use of interfaces

240

Java Graphics and Events

241

Console Applications

- Input from the keyboard
- Typically involves parsing of input strings
- Limited in interaction

242

Graphical User Interfaces

- User input through
  - Buttons
  - Pull down lists
  - Menus
  - Radio boxes
  - Choice boxes

243

Part C Assignment

- Use the Java GUI classes to draw several shapes on the screen
- Each shape has a x,y velocity
- Press a button and shapes will move on the screen according to:

  x = x + xVelocity

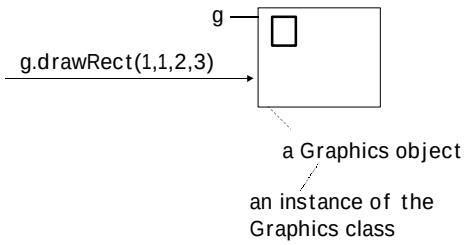
  y = y + yVelocity
- Each button press results in ONE update cycle

244

Basics of Screen Drawing

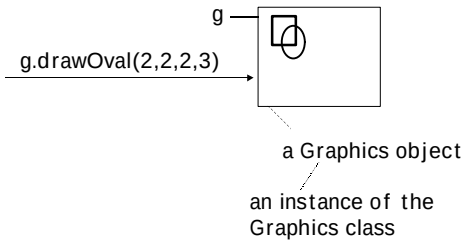
245

Display requires...



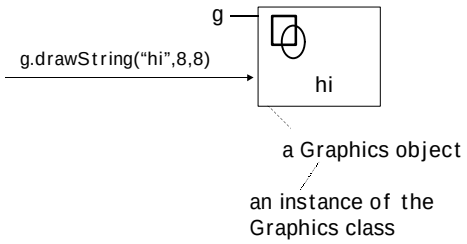
246

g.drawOval(...)



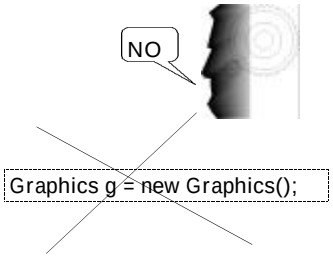
247

g.drawString(...)



248

how do you get a
graphics object?



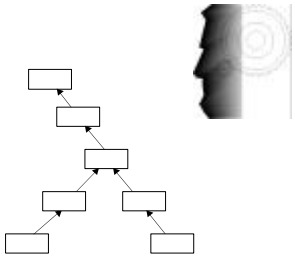
249

how do you get a graphics object?

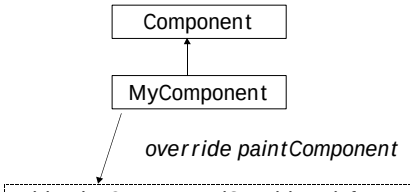

```
void paintComponent(Graphics g) {
   g.drawRect(1,1,2,3);
}
```

250

before you can draw anything you must understand the graphics class relationships

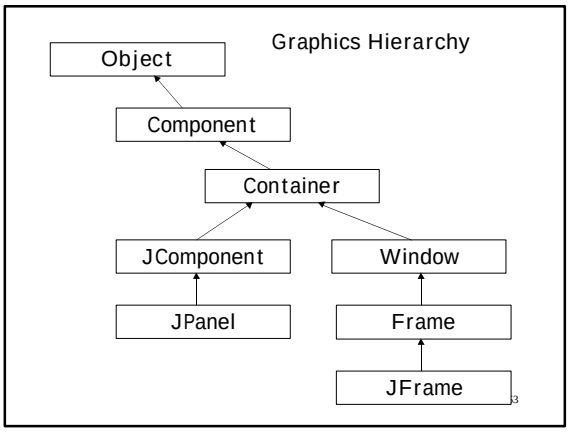


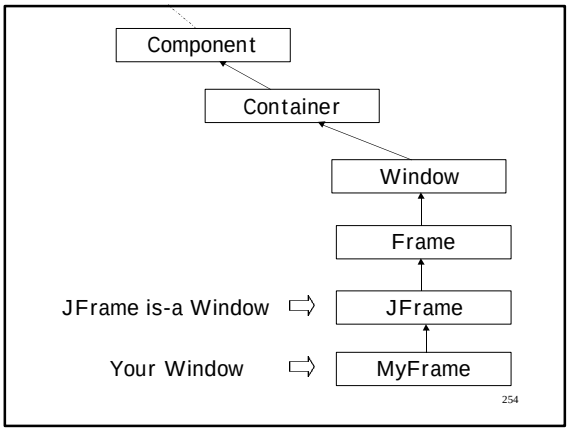
251

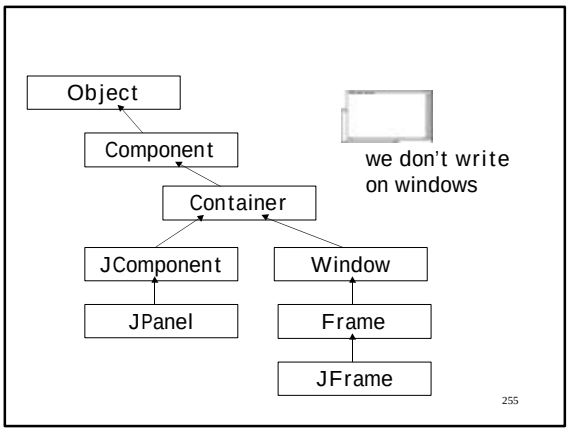


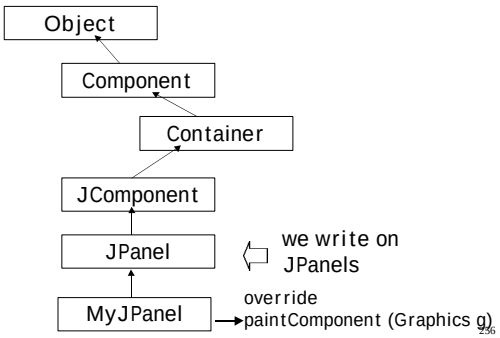

```
void paintComponent(Graphics g) {
   g.drawRect(1,1,2,3);
}
```

252


Graphics Hierarchy
Object
Component
Container
JComponent
Window
JPanel
Frame
JFrame
253


Component
Container
Window
Frame
JFrame is-a Window
JFrame
Your Window
MyFrame
254


Object
Component
Container
JComponent
Window
JPanel
Frame
JFrame
we don't write
on windows
255



256

GUI Drawing

- Create a Window
  - subclass JFrame
- Create a Panel to draw on
  - subclass JPanel
- Add the panel to the window
  - using the add method

257

GUI Drawing

- Create a Window
  - subclass JFrame

default JFrame windows are DUMB!

- they have zero size and won't go away

258

Example 7.1: FirstTest.java

```
import javax.swing.*;

class FirstFrame extends JFrame
{ public FirstFrame()
  { setTitle("FirstFrame");
    setSize(300, 300);
  }
}

public class FirstTest {
 public static void main(String[] args){
  JFrame frame = new FirstFrame();
  frame.show();
 }
}
```

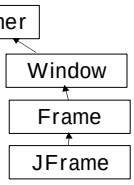
activates GUI thread

259

Example 7.1: FirstTest.java

```
import javax.swing.*;

class FirstFrame extends JFrame
{ public FirstFrame()
  { setTitle("FirstFrame");
    setSize(300, 300);
  }
}
```

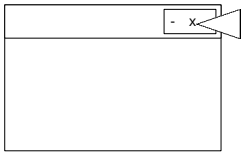
Default frame size is 0 x 0 pixels
Before you can see a frame, you need to give it a visible size



260

FirstFrame lives forever

- Clicking on the close box only *hides* the window, but *does not close the application*



261

Closing your App

- Options
  - Under Solaris, select Destroy from the system menu
  - Under Windows, press **`Ctrl+C`** or click on Close button in top right of shell window
  - Under Windows 95/98 or NT, carefully press **`Ctrl+Alt+Del`**

262

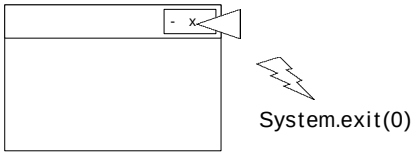
Why does FirstFrame lives forever?

- Clicking on the close box sends a message (triggers an event) that the user has clicked the X - …but nobody is receiving the message…

263

We want..

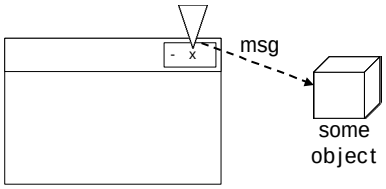
- Somebody to execute System.exit(0) when the X is pressed



264

We need a listener...

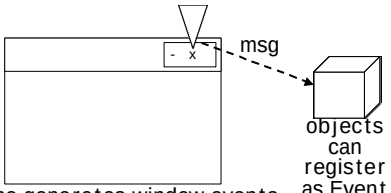
- An object that implements an agreed upon method and who then can shut down with System.exit(0)



265

Java Event Model

- Objects register with event sources (e.g. a Window) to receive notification that some event(s) have occurred

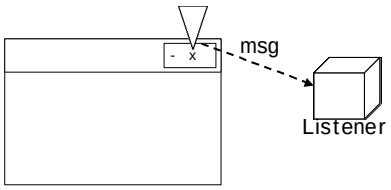


A JFrame generates window events

266

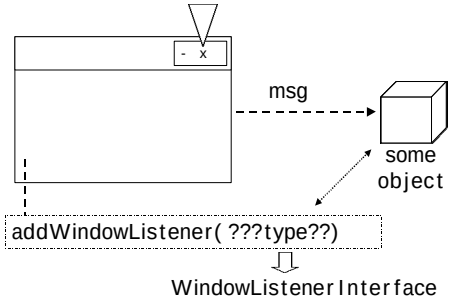
Window Event Registration

- A Window (JFrame) supports an addWindowListener(...) method



267

Window-listener relationship?

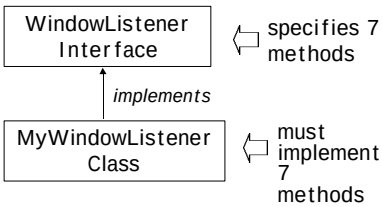


268

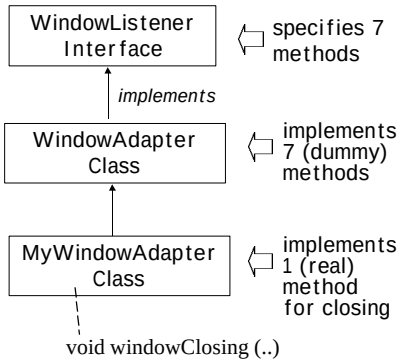
WindowListener Interface

- void windowActivated (..)
- void windowClosed (..)
- void windowClosing (..) ⇦ only one of interest right now
- void windowDeactivated (..)
- void windowDeiconified (..)
- void windowIconified (..)
- void windowOpened (..)

269



270

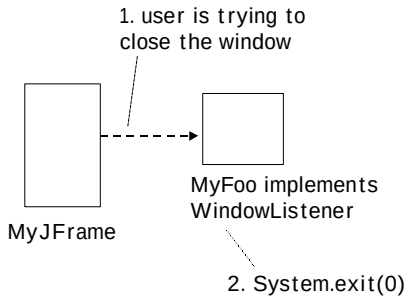


271

Options for Window Listener object

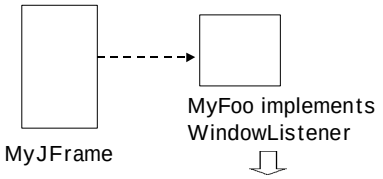
- Create separate class
  - seen in complex apps
  - many potential listeners
- Create anonymous inner class
  - seen when no one else is interested in events that may occur

272



273

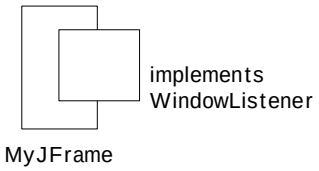
note: the WindowListener lives ONLY to service ONE object's Window events!



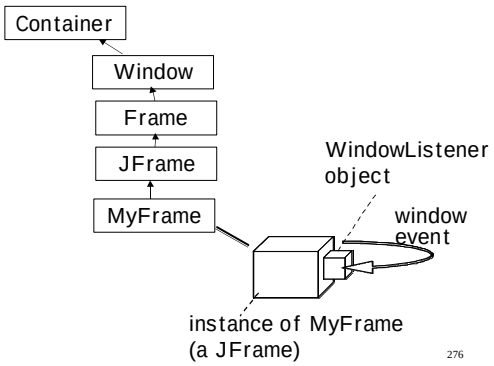
We, the programmer(s), don't need a reference to it

274

make it an <u>anonymous</u> <u>inner</u> class



275



276

Example 7.2: CloseableTest.java

```
import java.awt.event.*;
import javax.swing.*;

class MyJFrame extends JFrame {
 public MyJFrame()
  { setTitle("CloseableFrame");
    setSize(300, 200);
    addWindowListener(new
                  WindowAdapter(){
       public void
         windowClosing(WindowEvent e)
           { System.exit(0); } } );
  }
}
```

Inner class

277

Inner Class Syntax

Superclass

```
addWindowListener(new WindowAdapter( ) {
    public void windowClosing(WindowEvent e)
      { System.exit(0); }
 });
```

method that goes with new (un-named) subclass

278

```
public class CloseableTest {
 public static void main(String[] args){
  JFrame frame = new MyJFrame();
  frame.show();
 }
}
```

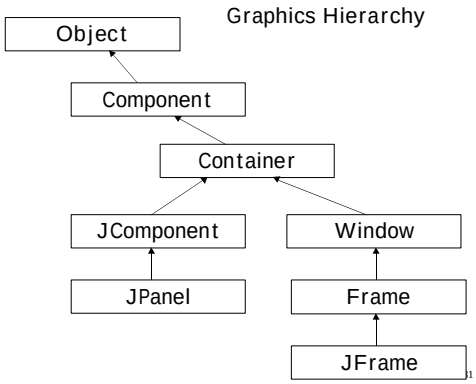
the Window that will create and add components to itself

279

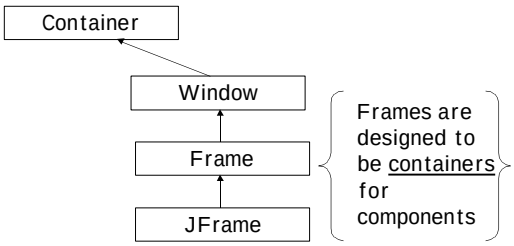
How do I draw?

280

Graphics Hierarchy

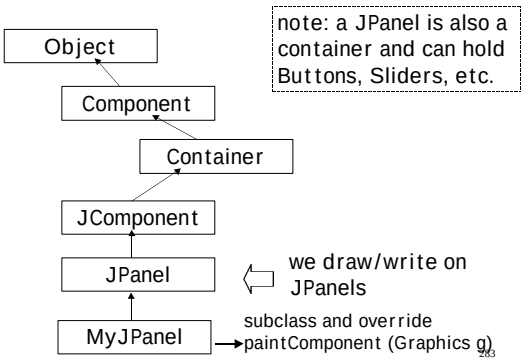


281

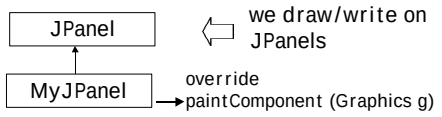


Drawing directly onto a frame is not considered "good programming practice," though it is possible

282

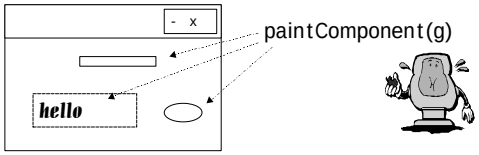


283




```
class MyJPanel extends JPanel {
 public void paintComponent(Graphics g) {
   super.paintComponent(g);
   g.drawString("Hello World", 50, 50);
 }
}
```

284

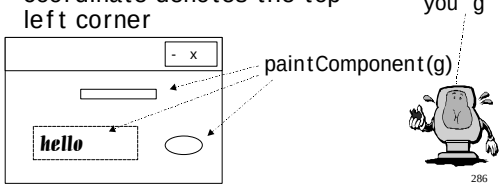


- Each time a window needs to be redrawn, no matter why, the event handler notifies the window. This causes **`paintComponent`** methods of <u>all components</u> to be executed.

285

paintComponent (Graphics g)

- All drawing in Java must go through a **Graphics** object
- All done in pixels -- (0, 0) coordinate denotes the top-left corner

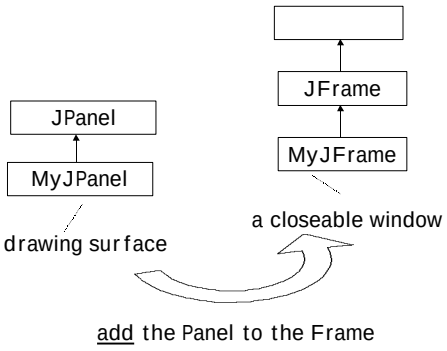


286

Important Tips to Remember

- Want to text messages or graphics in a panel -- override the **paintComponent** method.
- Never call the **paintComponent** method yourself --it's called automatically when:
  - the user increases the size of the window
  - if the user popped up another window that covered the existing one
  - the window is displayed for the first time
  - If you need to force repainting of the screen, call the **repaint** method instead of **paintComponent**.

287



288

Add-ing a Panel to a Window (Frame)

drawing surface

MyJPanel

a closeable window

MyJFrame

1. Ask the JFrame for its contentPane container

contentPane

```
Container contentPane =
        myJFrame.getContentPane();
```

289

drawing surface

MyJPanel

a closeable window

MyJFrame

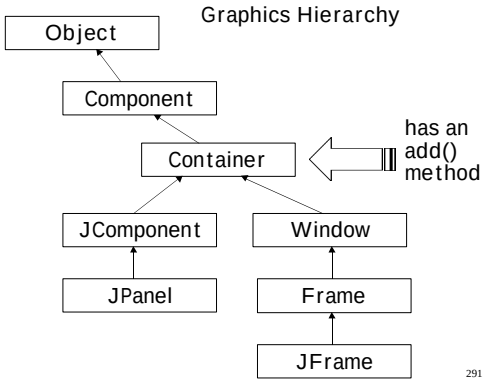
contentPane

2. Add Panel to container

```
MyJPanel jp = new MyJPanel();
contentPane.add(jp);
```

290

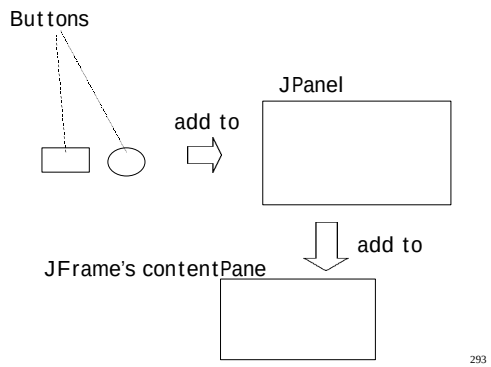
Graphics Hierarchy



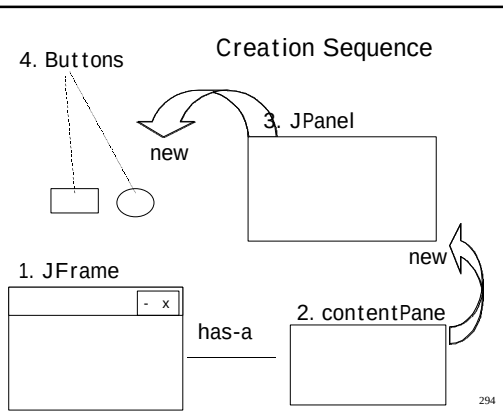
291

Buttons

292

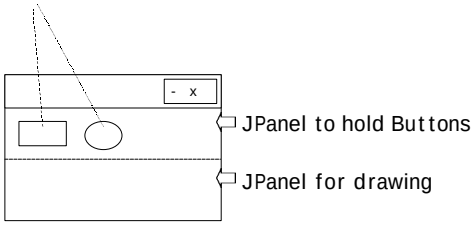


293



294

Convention: put Buttons in their own JPanel



BUT you can draw on ANY JPanel

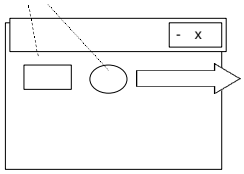
295

Responding to Button Clicks

ActionEvents and
ActionEventListeners

296

Buttons



generate ActionEvents i.e. objects that contain info about the button click

send the message:
`actionPerformed(ActionEvent evt)`

to registered `ActionListeners`

297

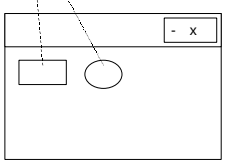
Java Event Handling

- A listener object implements a *listener interface*
- An event source is an object that can register listener objects and send them event objects
- The event source sends out event objects to all registered listeners
- The listener objects uses the information in the event object to react to the event

298

Buttons

1. create an instance of JButton



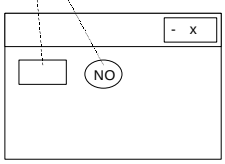
2. add it to a display surface

3. decide who will listen for button clicks

299

Buttons

1. create an instance of JButton

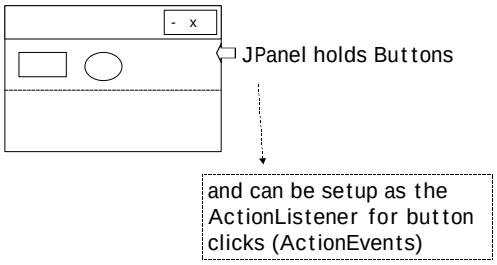



```
JButton myButton =
new JButton("NO");
```

2. add to a JPanel [container]
   buttonJPanel.add(myButton)
3. decide who will listen for button clicks
   usual choice is the JPanel (as container)

300

JPanel Double Duty



301

ActionListener Interface

- Requires method: **`actionPerformed()`**
- parameter: **`ActionEvent`** object

```
public class MyPanel extends JPanel
                    implements ActionListener {

  public void actionPerformed(ActionEvent evt)
   { // button click response
     . . .
   }
```

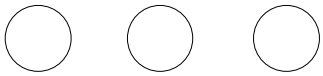
302

Example: Which Button Was Clicked?

```
class ButtonPanel extends JPanel {
 public ButtonPanel() {
   yellowButton = new JButton("Yellow");
   blueButton = new JButton("Blue");
   redButton = new JButton("Red");

   add(yellowButton);
   add(blueButton);
   add(redButton);
 }

private JButton yellowButton;
private JButton blueButton;
private JButton redButton;
```

303

Which Button Was Clicked???



Two ways to find out…

304

Technique One: The getSource Method

- Ask the event object (parameter) for a reference to the object that generated the event:

```
Object source = evt.getSource();

if (source == yellowButton) . . .
else if (source == blueButton) . . .
else if (source == redButton ) . . .
```

- Requires we keep references to the buttons

305

Technique Two: The getActionCommand Method

- Specific to ActionEvent class
- Use getActionCommand to returns the *string* associated with the button label

```
String command = evt.getActionCommand();
if (command.equals("Yellow")) . . .;
else if (command.equals("Blue")) . . .;
else if (command.equals("Red")) . . .;
```

306

Panel as Container & Listener

```
public ButtonPanel ()
{ yellowButton = new JButton("Yellow");
  blueButton = new JButton("Blue");
  redButton = new JBUtton("Red)";

  add(yellowButton);
  add(blueButton);
  add(redButton);

  yellowButton.addActionListener(this);
  blueButton.addActionListener(this);
  redButton.addActionListener(this);
}
```

Constructor

307

Broader Scope of ActionListener Interface

Used in different situations , e.g.:

- Button Pressed
- An item is selected from a list box with a double click
- A menu item is selected
- **ENTER** key is clicked in a text field

308