# Programming in Java

Classes, Inheritance, Interfaces



# Array Declaration

- Array Declaration
  - Example
    ```
    int[] scores = new int[10];
    ```
  - Variable type is "int[]"
  - Elements range from scores[0] … scores[9]
  - Automatic bounds checking
  - Each array has a public constant, **`length`**
    ```
    scores.length
    ```
    *- this evaluates to 10*
  - Alternate declarations
    ```
    float[] prices;
    float prices[];
    ```



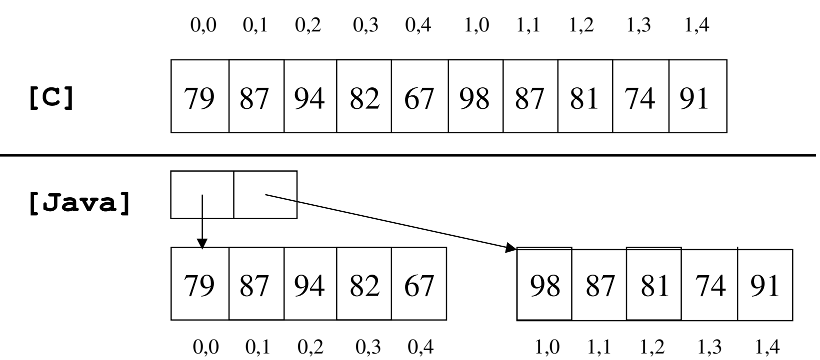
# Arrays

- Initializer lists can be specified
  ```
  int[] units = { 147, 323, 89 };
  ```
  – No new operator used
  – No size specified
- Elements of an array can be object references
  ```
  Strings[] words = new String[25];
  ```
  – This reserves space to store 25 references to String objects
  – String objects are not automatically created
- Arrays can be passed to methods
  – The reference is passed (true for all objects)
- Size of array is not part of type
  – A variable of type String[] can store ANY array of Strings, of any size



# Multi-Dimensional Arrays

- Multi-dimensional arrays
  - A two dimensional array is an array of arrays
    ```
    byte my_array[][] = new byte[2][5]
    ```
    – Memory is NOT necessarily contiguous



# Multi-Dimensional Arrays

- Each row (inner array) is independent of others
  - Can reassign part of the array
    ```
    my_array[1] = new byte[22];
    ```
  - Each row can have a different size than other rows
    ```
    int[][] table = { {28, 84, 47, 72}, {69, 26}, {91, 40, 28},
                      {42, 34, 37}, {13, 26, 57, 35} };
    ```
  - Can have partial declarations
    ```
    byte my_partial_array[][] = new byte[3][];
    String lots_of_strings[][][][] = new String[5][3][][];
    ```
  - All missing dimensions MUST be at the end
    ```
    String more_strings[][][] = new String[5][][3]; // Not valid
    ```



# Vectors

- Arrays
  - Once memory is allocated, the array size cannot change
    ```
    String[] my_array = new String[20];
    my_array = new String[10]; // OK; loses pointer to old array
    my_array.length = 30; // Not allowed
    ```
- java.util.Vector
  - Another way to represent list of values
    - Size of a Vector can change dynamically
    ```
    Vector courses = new Vector(20); // Compare to array decl.
    courses.addElement ("Lisp");
    courses.addElement ("Perl");
    // courses.addElement (220); // Not allowed
    courses.addElement (new Integer(220));  // OK with wrapper
    courses.size ()  // Returns 3
    courses.capacity() // returns 20
    ```



# Arrays vs. Vectors

- Differences between Arrays and Vectors
  - Vectors are dynamic (size can change)
  - Vector is more like a typical java class
    - No special syntax or operators (for ex. No [ ] operator for accessing)
  - Can only have Vectors of Objects
    - Any object, but… no primitive types
    - No strong type checking (No mechanisms for templates in Java)
    - Compiler cannot help prevent "Line objects" from getting into a vector that should only contain "Point objects"
- Vectors are implemented using an array of Object
  ```
  protected Object[] elementData;
  ```
  - Implications of insert, remove, resize operations



# Vector

- Constructors
  ```
  Vector(int initialCapacity)
  Vector(int initialCapacity, int capacityIncrement)
  Vector() // Default initial capacity is 10
  ```
- Size/Capacity methods
  ```
  public final int capacity();
  public final synchronized void ensureCapacity(int minCap);
  public final synchronized trimToSize();
  public final int size();
  public final synchronized void setSize(int newSize);
  ```
- Methods overridden from Object
  ```
  public synchronized Object clone();
  public final synchronized void toString();
  ```

# Vector

- Adding/Removing Elements

```
public final void addElement(Object arg);
```
*- Increases size; may increase capacity*
```
public final boolean removeElement (Object arg);
```
*- Reduces size; returns false if unsuccessful*

- Random Access

```
public final Object elementAt (int index)
```
*- Analogous to an array [ ] operator*
```
public final void insertElementAt (int index)
public final void removeElementAt (int index)
public final void setElementAt (Object arg, int index)
```

- Finding objects

```
public final int indexOf (Object arg)
public final int indexOf (Object arg, int index)
```
*- Return -1 if object is not found*



# StringTokenizer, StringBuffer

- StringTokenizer class
  - ../LewisLectures/chap06.ppt/#20
  - ../Src/Lewis/chap06/applications/Voltaire.java
  - ../Src/Lewis/chap06/applications/URL_Tokens.java
- StringBuffer class
  - Like String, but allows changes
    - append
```
StringBuffer text1 = new StringBuffer ("Hello");
text1.append (" World");
```
    - Other useful methods
```
insert (int index, char c);
charAt (int index);
setCharAt (int index, char c);
reverse();
length();
```



# StringBuffer Example

- StringBuffer.append is similar to String.concat

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

     String s = new String ("Hello ");
     String s2 = s.concat ("World");

     StringBuffer sb = new StringBuffer("Hello ");
     StringBuffer sb2 = sb.append ("World");

     System.out.println ("s:   " + s);
     System.out.println ("sb:  " + sb);
     System.out.println ("s2:  " + s2);
     System.out.println ("sb2: " + sb2);
   }
}  // class Concat
```

- What is the difference?



# Class Example

- Declaration

```
import java.lang.Math;
public class Circle {
  public int x, y, r; // (x,y) of Center; radius
  public double circumference () { return 2 * Math.PI * r; }
  public double area () { return  Math.PI * r * r; }
}
```

- Use

```
Circle c;
c = new Circle();
c.r = 2;
double a = c.area();
```

# Constructors

- Constructors
  - Perform initialization of objects
  - Declaration

```
public class Circle {
  private int x, y, r;
  public Circle (int ar) { this.x=0; y=0; r=ar; }
  … // Note the optional use of "this" above
}
```

  - Use

```
Circle c = new Circle (2);
double a = c.area();
```

  - Can have more than one constructor

```
public Circle (int ax, int ay, int ar) { x=ax; y=ay; r=ar; }
```

  - Can overload the default constructor

```
public Circle () { x=0; y=0; r=1; }
```

  - What if a class does not define ANY constructors?
  - What if a class defines constructors, but not one with NO arguments?



# Constructors

- One constructor can call another (unlike C++)
  - Uses "this"

```
public Circle (int ax, int ay, int ar) { x=ax; y=ay; r=ar; }
public Circle (int ar) { this(0, 0, ar); }
```

- Call to an alternate constructor MUST appear first
  - Before any other statements
  - Before variable declarations



# Class Variables

- Class variables

```
import java.lang.Math;
public class Circle {
  static int numCircle = 0;
  private int x=0, y=0, r=1; // Notice the initializers
  public Circle() { num_circles++; }
  public Circle (int ar) { this(); r=ar; }
  public double circumference () { return 2 * Math.PI * r; }
  public double area () { return  Math.PI * r * r; }
}
```

- Referencing Class variables
  - From within the class: this.numCircle (or just numCircle)

```
public Circle() { this.numCircle++; }
```

  - From outside the class: Circle.num_circle

```
Circle c = new Circle();
System.out.println ("# Circles= " + c.numCircle);
System.out.println ("# Circles= " + Circle.numCircle);
```



# Class Methods

- Class methods

```
import java.lang.Math;
public class Circle {
  private int x,y,r;
  int getX () { return this.x; }

  static int numCircle = 0;
  public static int getNumCircle() { return this.numCircle;}
}
```

- Calling class methods
  - From within the class

```
this.getNumCircle();
```

  - From outside the class

```
Circle c = new Circle();
int n1 = c.getNumCircle();
int n2 = Circle.getNumCircle();
```

# (Lack of) Globals

- Java does not allow global variables
  - Class variables can substitute for global variables
    - Advantage: no possibility of a collision in variable names
    - Example declaration in java.lang.Math:
      ```
      public final static double PI;
      ```
    - Example usage:
      ```
      public double circumference () { return 2 * Math.PI * r; }
      System.out.println ("Hello");
      ```
- Java does not allow global functions or methods
  - Class methods can substitute for global functions
    - Example declaration in java.lang.Integer:
      ```
      public static int parseInt(String str);
      ```
    - Example usage:
      ```
      int i = Integer.parseInt ("73");
      double sqrt_i = Math.sqrt(i);
      ```



# Inheritance

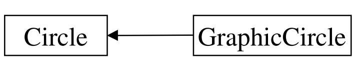
- Need a class with ability to draw Circles
  - Approach 1 (Not ideal)

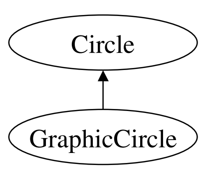
```
public class GraphicCircle {
  // Keep an instance var. to keep circle stuff
  public Circle c;

  // Delegate functionality to c
  public double area() { return c.area(); }
  public double circumference () {return c.circumference();}
  …

  // Add in GraphicCircle specific stuff
  public Color outline, fill;
  public void draw (Graphics page) { … }
}
```



# Inheritance

- Approach 2: Inheritance
  - A "GraphicCircle" isa (more specific version of) "Circle"
    ```
    public class GraphicCircle extends Circle {
      // Only need the 'additional' things
      Color outline, fill;
      public void draw (Graphics page) { … }
    }
    ```
- Terms and Concepts
  - Superclass, base class, parent class
  - Subclass, derived class, child class
  - isa, Class Hierarchy
  - Inheritance of instance variables and methods



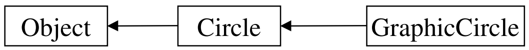



# Inheritance

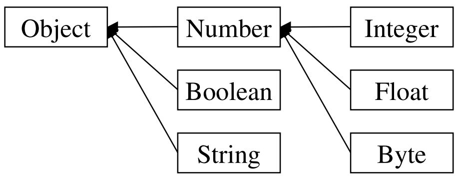
- GraphicCircle inherits all variables and methods
  ```
  GraphicCircle gc = new GraphicCircle ();
  gc.draw();   // Can invoke GraphicCircle methods
  gc.x = 5; // Can access Circle fields
  a = gc.area(); // Can invoke Circle methods
  ```
- GraphicCircle objects are also Circle objects
  ```
  Circle c;
  c = gc; // Assignment is legal
  a = c.area(); // Code can treat c as any other Circle
  c.draw(); // Illegal (draw is defined in GraphicCircle)

  boolean b1, b2;
  b1 = (c instanceof GraphicCircle);  // True
  b2 = (c instanceof Circle); // True
  ```

# Class Hierarchy

- All classes (except one) have a single superclass
  - No multiple inheritance
  - **`Object`** is the default superclass




- Classes and inheritance relationships form a Tree
  - Called Inheritance Hierarchy
  - Root of Tree is **`Object`**
  - All Java classes are part of this hierarchy






# Constructor Chaining

- A subclass invokes a superclass constructor
  - Explicitly - First line is a call to the superclass constructor

```
class GraphicCircle {
  …
  public GraphicCircle (int r, Color o, Color f) {
    super(r); // Must be first line
    this.outline = o;
    this.fill = f;
  }
```

  - Implicitly
    - If first line of constructor is not a call to a constructor, **`super()`** is automatically invoked
      - *What if supertype doesn't define a constructor with no arguments?*
      - *What if first line is a call to another constructor of the form* **`this(…)`** *?*
  - Note: Body of supertype constructor executes first (Like C++)!



# Overriding Methods

- Subclass can redefine method of superclass

```
class Circle { …
  public void reset () { x=0; y=0; r=1; }
}
class GraphicCircle { …
  public void reset () {
    x=0; y=0; r=1;
    fill = Color.getColor ("black");
  }
}
```

- Subclass method can call superclass method

```
class GraphicCircle { …
  public void reset () {
    super.reset();
    fill = Color.getColor("black");
  }
}
```



# Polymorphism; Final Modifier

- Actual method to call is determined at runtime
  - Depends on actual object's type (not variable type)

```
Circle[] c[2];
c[0] = new Circle();
c[1] = new GraphicsCircle();
for (int i=0; i<2; i++)
  c[i].reset();
```

  - C++ requires **`virtual`** keyword to implement polymorphism
    - C++ default (without keyword): resolution is done at compile time
    - Java: methods are "virtual" by default polymorphism
    - Can use **`final`** keyword modifier to enable compile time resolution

```
class Circle { …
  public final void reset () { x=0; y=0; r=1; }
}
class GraphicCircle { …
  public void reset () { } // No longer valid!
}
```

# Finalize Methods

- Finalize: Similar to C++ destructor
  - A place to clean up an object before memory is deallocated
    - Invoked before garbage collection
  - Typically used for closing files, releasing resources, etc.

```
public class FileOutputStream extends OutputStream {
  … // From java.io.FileOutputStream
  protected void finalize() throws IOException {
    if (fd != null) close(); // Closes file descriptor
  }
}
```

  - Not very common in Java (compared to C++)
    - Most cleanup is done automatically by the garbage collector
  - Not guaranteed to be called
    - Program may exit without *ever* calling the finalizer
    - Operating system must free any outstanding resources after exit



# Finalize Methods

- Java chains constructor methods (like C++
- Java does NOT chain finalize methods
  - If you define a finalizer, you should invoke the super's finalizer explicitly

```
class GraphicCircle extends Circle { …
  protected void finalize () {
    … local cleanup …
    super.finalize();
    … more local cleanup …
  }
}
```



# Visibility Modifiers

- Public, Private, Protected, Package

```
public class Circle { // With mixed visibility
  public int x;    // Public visibility
  protected int y; // Protected visibility
  int r;           // Package visibility (default)
  private int numCircle; // Private visibility
  int area() { … }
}
```

  - Package visibility is default
    - classes in same package are **`friend`**-ly to each other

| Accessible to | Public | Protected | Package | Private |
|---|---|---|---|---|
| Same Class | Y | Y | Y | Y |
| Class in same package | Y | Y | Y | N |
| Subclass in different package | Y | Y | N | N |
| Non-subclass, different package | Y | N | N | N |



# Visibility Modifier Guidelines

- Public
  - Use this for methods, constants that are part of the public API
  - Most variables should not be public (lecture notes don't follow this rule)
- Protected
  - For members that might be useful to subclasses (e.g. Circle's x,y,r)
  - But not for public use
- Package
  - For things that "cooperating classes" need access to
- Private
  - Fields and methods that should be hidden from everyone else

# Circle Class

```
public class Circle {
  protected int x=0, y=0, r=1;
  private static int numCircle=0; // No one has access

  /* Constructors */
  public Circle () { numCircle++; }
  public Circle (int ar) { this(); r=ar; }

  // Public way to get to variables (final for optimization)
  public final int getNumCircle() { return numCircle; }
  public final int getX() { return x; }
  public final int getY() { return y; }
  public final int getR() { return r; }

  // Methods to set variables
  public void moveTo (int newx, newy) { x=newx; y=newy; }
  public void move (int dx, int dy) { x+=dx; x+=dy; }
  public void setRadius (double newr) { r = newr; }
}
```



# Final Classes; Abstract Classes

- Final (no analogy in C++)
  - Method
    - Cannot be redefined by subclass
  - Class
    - Cannot be subclassed

```
public final class System extends Object { … }
public class MyClass extends System { … } // Not valid
```

- Abstract
  - Method (analogous to a pure virtual function in C++)
    - Must be redefined by subclass
  - Class
    - Cannot be instantiated
    - If a class has an abstract method, it must be declared an abstract class



# Abstract Class Example

```
public abstract class Shape {
  public abstract double area(); // Note: no definition
  public abstract double circumference();
}
public class Circle extends Shape {
  protected int x, y, r;
  public Circle(int ar) { r=ar; }
  public double area() { return Math.PI * r * r; }
  public double circumference() { return 2 * Math.PI * r; }
}
public class Rectangle extends Shape {
  protected int x, y, w, h;
  public Rectangle (int aw, int ah) { w=aw; h=ah; }
  public double area() { return w * h; }
  public double circumference() { return 2 * (w + h); }
}
```



# Abstract Class Example

- Example usage

```
public static void main () { …
  Shape[] shapes = new Shape[3];
  shapes[0] = new Circle(2);
  shapes[1] = new Rectangle (3,4);
  shapes[2] = new Rectangle (2,3);
  …
  double total_area = 0;
  for (int i=0; i<shapes.length; i++)
    total_area += shapes[i].area();
}
```

- Subclasses of Shape can be assigned to an array of Shape
- Area() method can be invoked on any kind of Shape
  - Declared as an abstract method in Shape
  - Not valid if area() method was not defined in Shape

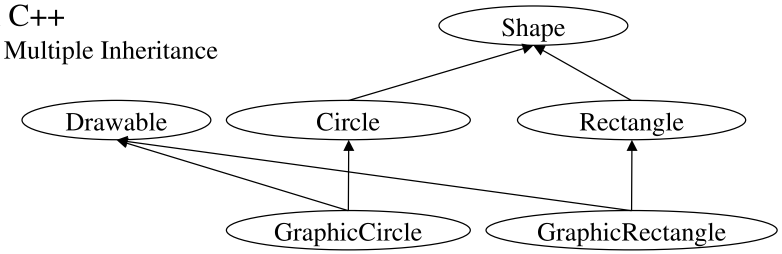
# Inheritance

- Example Hierarchy

```
Shape – abstract area(), circumference()
  Circle – area(), circumference()
    GraphicCircle – draw()

  Rectangle – area(), circumference()
    GraphicRectangle – draw()
```

  – Want to have a Drawable class, with an abstract draw()

- In C++
  – Multiple Inheritance






# Interface

- Java
  - No multiple inheritance
  - Java's solution: *interface*

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

- Interface
  - Looks like an abstract class; simulates some Multi-Inheritance
    – But uses keyword **interface** instead of **abstract** and **class**
    – All methods are abstract by default
    – All instance variables must be constants (**static** and **final**)
  - Other classes can *implement* an interface



# Interface

```
public class GraphicRectangle
  extends Rectangle implements Drawable
{
  private Color c;
  public GraphicRectangle (int w, int h) { super(w,h); }

  // Implement each method in Drawable
  public void setColor (Color ac) { c = ac; }
  public void setPosition (int ax, int ay) { x=ax; y=ay; }
  public void draw(Graphics dw) { ... }
}
```



# Using Interfaces

```
Shape[] shapes = new Shape[3];
Drawable[] drawables = new Drawable[3];

GraphicCircle dc = new GraphicCircle(1);
GraphicRectangle dr = new GraphicRectangle (3,4);
GraphicCircle dc2 = new GraphicCircle(3);

// Add them to arrays
shapes[0] = dc; drawables[0] = dc;
shapes[1] = dr; drawables[1] = dr;
shapes[2] = dc2; drawables[2] = dc2;

double total_area = 0;
for (int i=0; i<shapes.length; i++) {
  total_area += shapes[i].area();
  drawables[i].setPosition(i*10, i*10);
  drawables[i].draw(gc); // Assume gc is defined somewhere
}
```

# Multiple Interfaces

- Each user defined class
  - Extends exactly one other class
  - Implements 0, 1, or more interface

```
public class GraphicRectangle
  extends Rectangle
  implements Drawable, java.lang.Cloneable,
             java.lang.Serializable
{
  ...
}
```

- Interface
  - Provides a way to simulate multiple inheritance
  - Every class that implements an interface MUST define all methods of that interface



# Interface Hierarchy

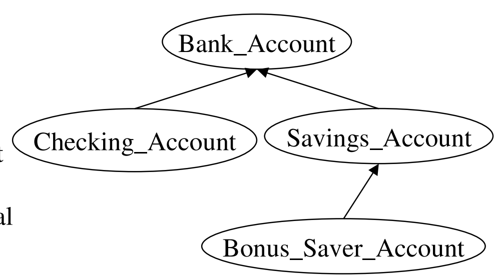
- Interfaces can be subtypes of other interfaces
  - Results in an interface hierarchy
    - Directed, Acyclic Graph (not a tree, like the class hierarchy)

```
public interface Transformable
  extends Scalable, Rotateable, Reflectable { ... }
public interface GraphicObject
  extends Drawable, Transformable { ... }
public class Shape implements GraphicObject { ... }
```



# Case Study on Inheritance

- Case Study
    - Solution: Lewis: ../chap08/applications/Accounts2.java
  - Bank_Account
    - Generic account with ability to make deposits and withdrawals
  - Savings_Account
    - A kind of Bank_Account
    - Collects interest
  - Bonus_Saver_Account
    - A kind of Savings_Account
    - Collects more interest
    - Has penalties for withdrawal
  - Checking_Account
    - A kind of Bank_Account
    - Has overdraft protection






# Define Class Hierarchy

```
public class Bank_Account {
  ...
}
public class Savings_Account extends Bank_Account {
  ...
}
public class Bonus_Saver_Account extends Savings_Account {
  ...
}
public class Checking_Account extends Bank_Account {
  ...
}

public class Accounts2 {
  public static void main (String[] args) {
    ... Create objects and test out class hierarchy ...
  }
}
```

# Define Methods

```
public class Bank_Account {
  public Bank_Account(int account_num,double init_bal) {...}
  public void deposit (double amount) { ... }
  public void withdrawal (double amount) { ... }
}

public class Savings_Account extends Bank_Account {
  public void add_interest () { ... }
}

public class Bonus_Saver_Account extends Savings_Account {
  public void withdrawal (double amount) { ... penalty ... }
  public void add_interest () { ... give bonus rate ... }
}

public class Checking_Account extends Bank_Account {
  public void withdrawal (double amount)
    { ... check for overdraft ... }
}
```



# Define Methods (Details)

```
public class Bank_Account {
  protected int account;
  protected double balance;

  public Bank_Account(int account_num,double init_bal) {...}
    Bank_Account aBA = new Bank_Account(4321, 100.00);

  public void deposit (double amount) { ... }
    aBA.deposit (50.00);

  public void withdrawal (double amount) { ... }
    aBA.withdraw (20.00);
}
```



# Define Methods (Details, cont.)

```
public class Savings_Account extends Bank_Account {
  protected double rate;

  public Savings_Account (int account_num,
                          double initial_balance,
                          double interest_rate) { ... }
    Savings_Account aSA = new Savings_Account (1234, 100.00, 0.05);
  public void add_interest () { ... }
    aSA.add_interest();
}

public class Bonus_Saver_Account extends Savings_Account {
  public Bonus_Saver_Account (int account_num,
                              double initial_balance,
                              double interest_rate) { ... }
    Bonus_Saver_Account aBSA = new Bonus_Saver_Account (1234, 100.00, 0.05);
  public void withdrawal (double amount) { ... penalty ... }
    aBSA.withdraw ( 20.00 );
  public void add_interest () { ... give bonus rate ... }
    aBSA.add_interest ();
}
```



# Define Methods (Details, cont.)

```
public class Checking_Account extends Bank_Account {
  private Savings_Account overdraft;

  public Checking_Account (int account_number,
                           double initial_balance,
                           Savings_Account protection) {...}
    Checking_Account aCA = new Checking_Account (87323, 75.00, aBSA);

  public void withdrawal (double amount) {
    if (checking account has enough funds)
      take funds out of checking account
    else if overdraft account has enough funds
      take funds out of overdraft account
    else
      print error "Insufficient funds"
  }
    aCA.withdraw (20.00);
}
```

# Bank_Account

```
class Bank_Account {
   protected int account;
   protected double balance;

   public Bank_Account (int account_num, double init_bal) {
     account = account_num;
     balance = initial_balance;
   }  // constructor Bank_Account

   public void deposit (double amount) {
     balance += amount;
     System.out.println("Deposit into account " + account);
     System.out.println("Amount: " + amount);
     System.out.println("New balance: " + balance);
     System.out.println();
   }  // method deposit

   // ... rest is on next slide
```



# Bank_Account (cont.)

```
   public boolean withdrawal (double amount) {
     boolean result = false;
     System.out.println("Withdrawal from account " + account);
     System.out.println ("Amount: " + amount);

     if (amount > balance)
       System.out.println ("Insufficient funds.");
     else {
       balance -= amount;
       System.out.println ("New balance: " + balance);
       result = true;
     }
     System.out.println();

     return result;
   }  // method withdrawal
}  // class Bank_Account
```



# Savings_Account

```
class Savings_Account extends Bank_Account {
   protected double rate;

   public Savings_Account (int account_num,
                           double initial_balance,
                           double interest_rate) {
      super (account_num, initial_balance);
      rate = interest_rate;
   }  // constructor Savings_Account

   public void add_interest () {
      balance += balance * rate;
      System.out.println ("Interest added to account: " +
                          account);
      System.out.println ("New balance: " + balance);
      System.out.println();
   }  // method add_interest
}  // class Savings_Account
```



# Bonus_Saver_Account

```
class Bonus_Saver_Account extends Savings_Account {
   private final int PENALTY = 25;
   private final double BONUS_RATE = 0.03;

   public Bonus_Saver_Account (int account_num,
                               double initial_balance,
                               double interest_rate) {
      super (account_num, initial_balance, interest_rate);
   }  // constructor Super_Saver_Account

   public boolean withdrawal (double amount) {
      System.out.println ("Penalty incurred: " + PENALTY);
      return super.withdrawal (amount+PENALTY);
   }  // method withdrawal
```

# Bonus_Saver_Account (cont.)

```
    public void add_interest () {
        balance += balance * (rate + BONUS_RATE);
        System.out.println ("Interest added to account: " +
                            account);
        System.out.println ("New balance: " + balance);
        System.out.println();
     }  // method add_interest
}  // class Bonus_Saver_Account
```



# Checking_Account

```
class Checking_Account extends Bank_Account {

  private Savings_Account overdraft;

  public Checking_Account (int account_number,
                           double initial_balance,
                           Savings_Account protection) {
    super (account_number, initial_balance);
    overdraft = protection;
  }  // constructor Checking_Account

  // ... continued on next slide
```



# Checking_Account (cont.)

```
  //...
  public boolean withdrawal (double amount) {
    boolean result = false;
    if ( ! super.withdrawal (amount) ) {
      System.out.println ("Using overdraft...");
      if ( ! overdraft.withdrawal (amount-balance) )
        System.out.println("Overdraft funds insufficient.");
      else {
        balance = 0;
        System.out.println ("New balance on account " +
                            account + ": " + balance);
        result = true;
      }
    }
    System.out.println ();
    return result;
  }  // method withdrawal
}  // class Checking_Account
```



# Main

```
class Accounts2 {
  public static void main (String[] args) {
    Savings_Account savings =
      new Savings_Account (4321, 5028.45, 0.02);
    Bonus_Saver_Account big_savings =
      new Bonus_Saver_Account (6543, 1475.85, 0.02);
    Checking_Account checking =
      new Checking_Account (9876, 269.93, savings);

    savings.deposit (148.04);
```

*Deposit into account 4321*
*Amount: 148.04*
*New balance: 5176.49*

```
    big_savings.deposit (41.52);
```

*Deposit into account 6543*
*Amount: 41.52*
*New balance: 1517.37*

# Main (cont.)

```
savings.withdrawal (725.55);
```

*Withdrawl from account 4321*
*Amount: 725.55*
*New balance: 4450.94*

```
big_savings.withdrawal (120.38);
```

*Penalty incurred: 25*
*Withdrawl from account 6543*
*Amount: 145.38*
*New balance: 1371.9899999999998*

```
checking.withdrawal (320.18);
```

*Withdrawl from account 9876*
*Amount: 320.18*
*Insufficient funds.*

*Using overdraft...*
*Withdrawl from account 4321*
*Amount: 50.25*
*New balance: 4400.69*

*New balance on account 9876: 0.0*

```
  }  // method main
}  // class Accounts2
```