class Person
{
    public Person( String n, int ag, String ad, String p )
    { name = n; age = ag; address = ad; phone = p; }

    //accesspr (getter) methods
    public String getName( ) { return name; }
    public int getAge( ) { return age; }
    public String getAddress( ) { return address; }
    public String getPhoneNumber( ) { return phone; }

    //mutator (setter) methods
    public void setAddress( String newAddress )
    { address = newAddress; }
    public void setPhoneNumber( String newPhone )
    { phone = newPhone; }

    public String toString() {return "Name: " + getName() + ", Age: " + getAge() + ", Address: " + getAddress() + ", Phone: " + getPhoneNumber(); }

    private String name, address, phone;
    private int age;
}
class Person
{
    public Person( String n, int ag, String ad, String p )
    { name = n; age = ag; address = ad; phone = p; }

    //accesspr (getter) methods
    public String getName() { return name; }
public int getAge() { return age; }
public String getAddress() { return address; }
public String getPhoneNumber() { return phone; }

    //mutator (setter) methods
    public void setAddress( String newAddress )
    { address = newAddress; }
public void setPhoneNumber( String newPhone )
    { phone = newPhone; }

    public String toString() {return "Name: " + getName() + ", Age: " + getAge() + ", Address: " + getAddress() + ", Phone: " + getPhoneNumber(); }

    private String name, address, phone;
    private int age;
}

GUI Basics

Public methods

Private instance variables
class Student extends Person
{
    private double gpa;

    public Student( String n, int ag, String ad, String p, double g )
    {
        super( n, ag, ad, p );
        gpa = g;
    }

    public String toString( )
    {
        return super.toString( ) + " " + getGPA();
    }

    public double getGPA( )
    {
        return gpa;
    }
}

Calls superclass constructor with four parameters

Can call superclass methods using super

Partial overriding: use super to call a superclass method, when we want to do what the base class does plus a bit more, as in this example
class Employee extends Person {
    private double salary;

    public Employee( String n, int ag, String ad, String p, double s ) {
        super( n, ag, ad, p );
        salary = s;
    }

    public String toString( ) {
        return super.toString( ) + " " + getSalary();
    }

    public double getSalary( ) {
        return salary;
    }
}

Calls superclass constructor with four parameters
Partial overriding
Expanded Person family tree
Type Compatibility

• Because a **Student** IS-A **Person**, a **Student** object can be accessed by a **Person** reference

```java
Student s = new Student( "Joe", 26, "1 Main St", "845-555-1212", 4.0 );
Person p = s;
System.out.println("age is " + p.getAge());
```

• **p** may reference any object that IS-A **Person**
• Any method defined in the **Person** class or defined in the **Person** class and overridden in the **Student** class can be invoked through the **p** reference
Type Compatibility

```java
Student s = new Student ( "Joe", 26, "1 Main St", "845-555-1212", 4.0 );
Person p = s;
System.out.println( "age is " + p.getAge() );
System.out.println( "GPA is " + p.getGPA() );
// LINE ABOVE CAUSES ERROR
```

- But we can’t call methods defined *only* in class **Student** by using the reference **p** as it appears above. This is because a **Person** is not necessarily a **Student**.
Type Compatibility

• if p is cast as a **Student** the code works

• RULE: If a superclass identifier references a subclass object, then you need to cast the identifier using (subclass) cast when calling a subclass method.

```java
Student s = new Student ( "Joe", 26, "1 Main St", "845-555-1212", 4.0 );
Person p = s;
System.out.println( "age is " + p.getAge() );
System.out.println( "GPA is " +
    ((Student)p).getGPA() );
// LINE ABOVE IS OK if we cast p as a Student
```
Inheritance Hierarchies

superclass

subclass

superclass

subclass

subclass
Assignments to Variables

• RULE: You can assign an object of a derived class to a variable of any ancestor type, but not the other way around

```java
Person p1, p2;
p1 = new Student();
p2 = new Undergraduate();
```

LEGAL

```java
Student s = new Person();
Undergraduate ug = new Person();
Undergraduate ug2 = new Student();
```

ILLEGAL
Converting Wrapper Classes to Primitive Types (and vice versa)

• To convert an int to an Integer object, we need to do the following:

```java
int y = 5;
Integer num = new Integer(y);
// Integer num = new Integer("5"); works too
```

• To convert the above Integer object num back to an int (assuming the above declarations), use the following command:

```java
y = num.intValue();
```
Interfaces (WN Ch. 20, GT 2.4)

• In order for objects to interact, they must "know" about the public methods each supports.

• Java application programming interface (API) requires classes to specify the interface they present to other objects.

• The major structural element in Java that supports an API is the interface

  ==> Collection of method declarations and/or constants with no variables and no method bodies. Class full of empty methods!
Interfaces

**Interface**: a collection of *constants* and *method declarations*. An interface can't be instantiated.

Methods in an interface do not have any code within statement body. Has a ';' after declaration (signature).

```java
public interface Speaker {
    public void speak ();
    public void announce (String str);
}
```

speak and announce are method declarations in interface Speaker
Implementing Interfaces

```java
public class Philosopher implements Speaker {

    public Philosopher (String thoughts) {
        philosophy = thoughts;
    }

    public void speak() {
        System.out.println(philosophy);
    }

    public void announce(String announcement) {
        System.out.println(announcement);
    }

    private String philosophy;
}
```

Any class that extends Philosopher now is subtype of Speaker.

Philosopher class **must** declare a method for each of the methods declared in the interface.
The Philosopher class could implement other methods for which there is no declaration in the Speaker interface, but it must have implementations of each interface method.

A class *implements* an interface by providing method implementations for each method defined in the interface. The implementing class is a subtype of the interface.

The keyword *implements* in the Philosopher class header says the class is defining bodies for each method in the interface.
Another Interface Example

```java
public interface Rollable
{
    // Reselect the upward-pointing face of object
    public void roll();
    // return the current value of the object
    public int value();
}
```

This interface specifies that there must be roll and value methods in each object that implements it.

```java
public class Die implements Rollable {...}
```

Now Die is a subtype of the Rollable type. We can use a Die object anywhere Rollable objects are required.
Multiple Inheritance

The ability to derive a class from more than one parent class is known as *multiple inheritance*.

```java
class Person {...}
class Democrat {...}
class Employee extends Person, Democrat {...}
```

Multiple inheritance is NOT ALLOWED in Java (i.e., a class can't extend more than one other class)
A Java Provision for Multiple Inheritance...

Classes can implement multiple interfaces

```java
interface Manager {...}

interface Democrat {...}

class Person {...}

class Employee extends Person implements Manager, Democrat {...}
```

The Employee class would be a subclass of Person and a subtype of Manager and Democrat. We could write a program that makes use of Employee objects anywhere Person, Manager, or Democrat objects are required!
Another Way Java Provides for Multiple Inheritance...

Interfaces can extend multiple interfaces

```
interface Democrat {...}
interface Farmer {...}

interface Senator extends Democrat, Farmer {...}
```

```
class Person {...}
```

```
class Employee extends Person implements Senator {...}
```

The Employee class would be a subclass of Person and a subtype of Senator.
Multiple Interfaces

When a class implements an interface that extends another interface, it must include all methods from each interface in hierarchy.

Faculty and Staff must include methods of both Retirement and StatePlan interfaces.
Dynamic Binding and Polymorphism

Even though you can't instantiate an object from an interface, you can assign an object of a subtype to a variable of its supertype (i.e., an interface the object implements).

```java
StatePlan s = new Faculty ( "Joe", 26, "1 Main St", "845-555-1212", 10000.0 );
StatePlan e = new Staff ( "Boss", 42, "4 Main St", "854-555-1212", 10000.0 );
StatePlan p = null;
if( ((int)(Math.random() * 10)) % 2 == 1 )
    p = s;
else
    p = e;
System.out.println( "Person is " + p.toString() );
```

Do not know until program runs whether to use Faculty's toString or Staff's toString
Casting in an Inheritance Hierarchy

- The "instanceof" operator allows us to test whether an object variable is referring to an object of a certain class, derived from a certain class, or implementing a certain interface.

```java
StatePlan s = new Faculty( "Joe", 26, "1 Main St", "845-555-1212", 10000.0 );
StatePlan e = new Staff( "Boss", 42, "4 Main St", "854-555-1212", 10000.0 );
StatePlan p = null;
if( ((int)(Math.random() * 10)) % 2 == 0 )
    p = s;
else
    p = e;
if (p instanceof Staff) Staff f = (Staff)p;
```
Abstract Classes

• Abstract classes lie between interfaces and complete classes.

  ==> Class that may contain empty method declarations as well as fully defined methods and instance variables.

  ➢ Not possible to instantiate an abstract class.
  ➢ Subclasses must provide an implementation for each abstract method in the parent class.
  ➢ "Partial" implementation of a class. Derived classes complete the definition.

    public abstract class Matrix implements Graph {...}
An Abstract Class

The purpose of an abstract class is to define inheritable, shared variables and methods and to impose requirements through abstract methods.

```java
public abstract class Attraction {
    public int minutes;
    public Attraction() {minutes = 75;}
    public Attraction(int m) {minutes = m;}
    public int getMinutes() {return minutes;}
    public void setMinutes(int m) {minutes = m;}
    public abstract int rating();
}
```

Any classes derived from Attraction would inherit the public members and would have to provide an implementation of the abstract method `rating`. 
A Class Derived from Attraction

```java
public class Movie extends Attraction {
    public int script, acting, direction;

    public Movie() {script=5; acting=5; direction = 5;}

    public Movie(int m) {super(m);}

    public int rating() {
        return script+acting+direction+getMinutes();
    }
}
```

Any classes derived from Attraction would inherit the public members and would have to provide an implementation of the abstract method `rating`. 
class GenericArray {
    public static void main (String[] args) {
        Object[] array = new Object[4];
        array[0] = "String 1";
        array[1] = new Integer(1);
        array[2] = new Person();
        array[3] = new Integer("57");
        for (int i = 0; i < array.length; i++) {
            if (array[i] instanceof String) {
                String temp = (String)array[i];
                System.out.println(temp);
            }
            else if (array[i] instanceof Integer) {
                int x = ((Integer)array[i]).intValue();
                System.out.println(x);
            }
            else if (array[i] instanceof Person) {
                int y = ((Person)array[i]).getAge();
                System.out.println(y);
            }
        }
    }
}
File Input & Output (File I/O)

- File I/O in Java can be accomplished by using one of many built-in Java classes.
  - Reading input from a file:
    - `BufferedReader inFile = new BufferedReader(new FileReader("input.dat"));`
      Function: Reads text as a stream of characters.
    - `StreamTokenizer token = new StreamTokenizer(inFile);`
      Function: Takes an input stream and parses it into "tokens", allowing the tokens to be read one at a time.
  - Writing output to a file:
    - `PrintWriter outFile = new PrintWriter(new FileWriter("output.dat"));`
      Function: Prints formatted representations of objects to text output stream.
import java.io.*;

public class TestReadWrite {
    public static void main (String[] args) throws IOException{
        BufferedReader inFile = new BufferedReader
            (new FileReader("input.dat"));
        PrintWriter outFile = new PrintWriter
            (new FileWriter ("output.dat");
        StreamTokenizer token = new StreamTokenizer(inFile);

        token.nextToken();

        int number = ((int) token.nval);

        outFile.println("The number read is "+number);

        outFile.close();
        inFile.close();
    } // end of main
} // end of class
File I/O example using Command line arguments

```java
import java.io.*;

public class TestReadWrite {
    public static void main (String[] args) throws IOException{
        BufferedReader inFile = new BufferedReader
            (new FileReader (args[0]));
        PrintWriter outFile = new PrintWriter
            (new FileWriter (args[1]));
        StreamTokenizer token = new StreamTokenizer(inFile);
        token.nextToken();
        int number = ((int) token.nval);
        outFile.println("The number read is "+number);
        outFile.close();
        inFile.close();
    } // end of main
} // end of class
```
import java.io.*;

public class TestReadWrite {
    public static void main (String[] args) throws IOException{
        BufferedReader inFile = new BufferedReader
                           (new FileReader (args[0]));
        PrintWriter outFile = new PrintWriter
                          (new FileWriter (args[1]));
        StreamTokenizer token = new StreamTokenizer(inFile);
        token.nextToken();
        int number = ((int) token.nval);
        outFile.println("The number read is "+number);
        outFile.close();
        inFile.close();
    } // end of main
} // end of class

args[0] and args[1] are Strings read from the command line.

token.nextToken() looks for characters separated by white space.
token.nval returns the value of the token as a double (cast as an int here).
import java.io.*;

public class TestReadWrite {
    public static void main (String[] args) throws IOException{
        BufferedReader inFile = new BufferedReader(new FileReader(args[0]));
        PrintWriter outFile = new PrintWriter(new FileWriter(args[1]));
        StreamTokenizer token = new StreamTokenizer(inFile);
        token.nextToken();
        int number = ((int) token.nval);
        outFile.println("The number read is "+number);
        outFile.close();
        inFile.close();
    } // end of main
} // end of class

Calling methods to read and write from files can generate IOExceptions. You must include throws statement in declaration of any method that reads and/or writes to files.

outFile.println prints to a file, not the screen

Always call the close method in files when finished using them.
Reading Command-Line Arguments

- Command-line arguments are read through the main method's array of Strings parameter, args (or whatever you call it).

- For example, to run the TestReadWrite program shown on the 2 previous slides, we would type

  ```java
  java TestReadWrite input.dat output.dat
  ```

  **IMPORTANT:** You must have a non-empty file in the same directory as the TestReadWrite.class file called "input.dat" when you run this program!! Also, any file called "output.dat" in the current directory will be overwritten.

- In the TestReadWrite program, args[0] = "input.dat" and args[1] = "output.dat" during execution of the program.
import java.io.*;

public class ChangeCoinsApp {

    public static void main (String args[]) throws IOException {

        BufferedReader stdin =
            new BufferedReader(new InputStreamReader(System.in));

        System.out.println("Enter a number: ");

        String line = stdin.readLine();
        int num = Integer.parseInt(line);
    }
}

Taking Input from the Console

Add 2 words to main method declaration

Declare an object that takes input from the keyboard.

Read the String and convert it to an int using the static parseInt method of the Integer class.
import java.io.*;
public class TestReadWrite {
    public static void main (String[] args) throws IOException{
        BufferedReader fileIn = new BufferedReader(new FileReader("pal.txt"));
        PrintWriter outFile = new PrintWriter(new FileWriter("palindrome.txt"));
        String line, lcString;
        StringTokenizer str;
        while ((line = fileIn.readLine()) != null) {
            str = new StringTokenizer(line.toLowerCase()," ,:.;-!?"");
            while(str.hasMoreTokens()) {
                lcString = str.nextToken();
                outFile.println(lcString);
            } // end inner while
        } // end outer while
        fileIn.close(); // close input file
        outFile.close(); // close output file
    } // end main
} // end class