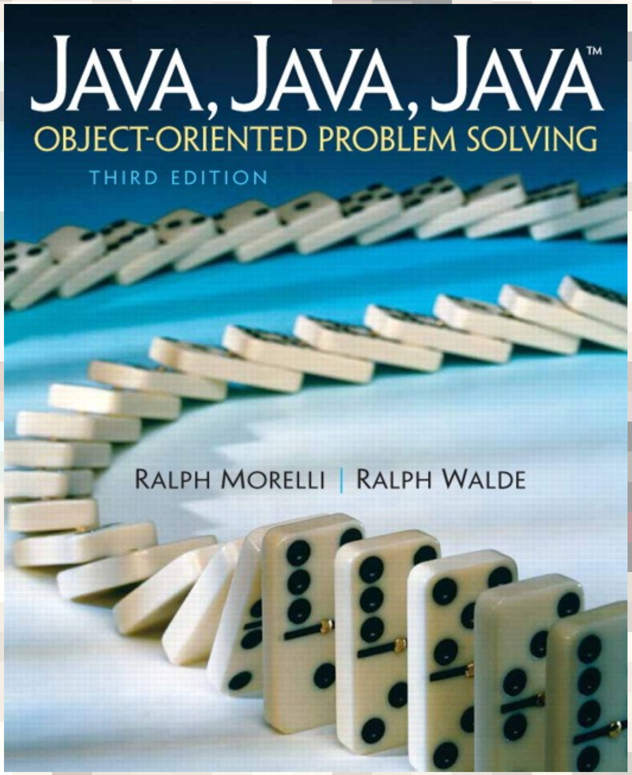


presentation slides for

JAVA, JAVA, JAVA

Object-Oriented Problem Solving

Third Edition

Ralph Morelli | Ralph Walde
Trinity College
Hartford, CT

published by Prentice Hall

Java, Java, Java

Object Oriented Problem Solving

Lecture 02: Inheritance and Polymorphism

Objectives

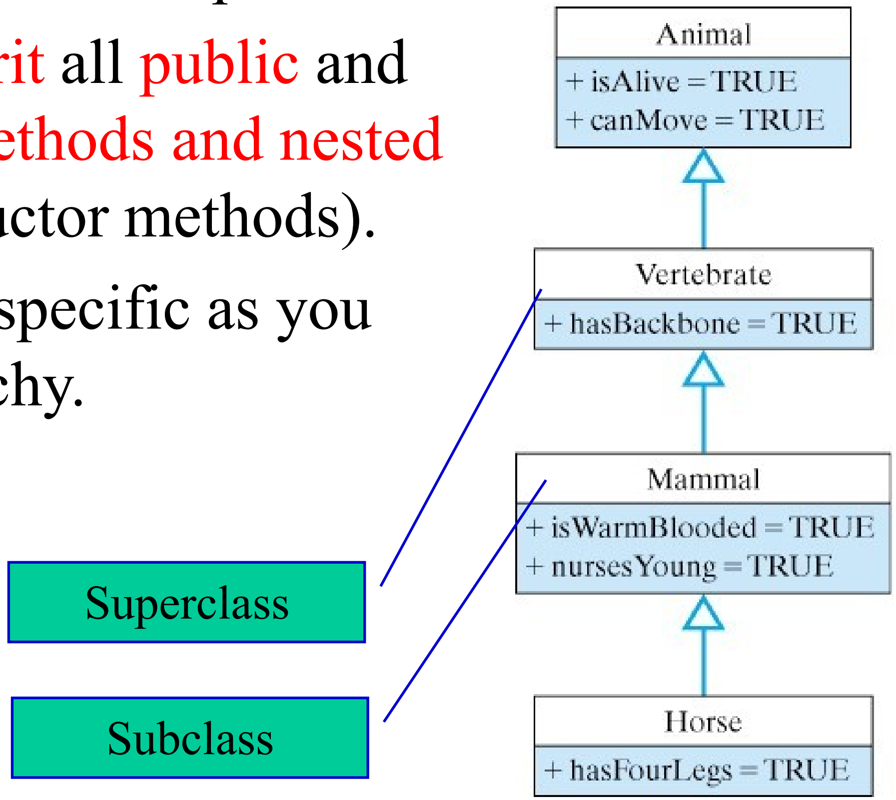
- Understand the concepts of inheritance and polymorphism.
- Know how Java's dynamic binding mechanism works.
- Be able to design and use abstract methods and classes.
- Be able to design and use polymorphic methods.
- Gain a better understanding of object-oriented design.

Outline

- Introduction
- Java’s Inheritance Mechanism
- Abstract Classes, Interfaces, and Polymorphism
- Example: A Toggle Button
- Example: The Cipher Class Hierarchy
- Case Study: A Two-Player Game Hierarchy
- Principles of Object-Oriented Design

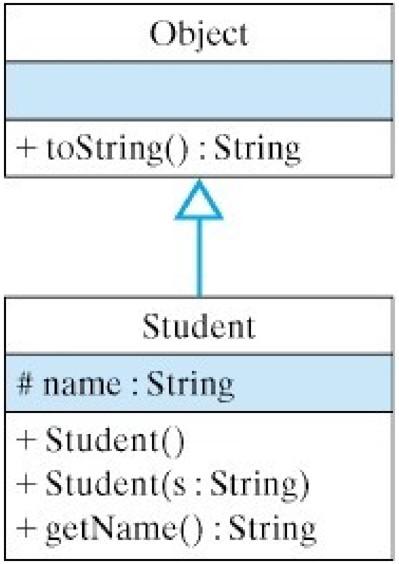
Java's Inheritance Mechanism

- *Class inheritance* is the mechanism whereby a class acquires (inherits) the methods and variables of its superclasses.
- **Rule**: Subclasses inherit all public and protected variables, methods and nested classes (except constructor methods).
- Classes become more specific as you move down the hierarchy.



Using an Inherited Method

- All classes are subclasses of the Object class. Use keyword "extends" to inherit.
- Any subclass of Object *inherits* the toString() method, meaning it can use it as its own.



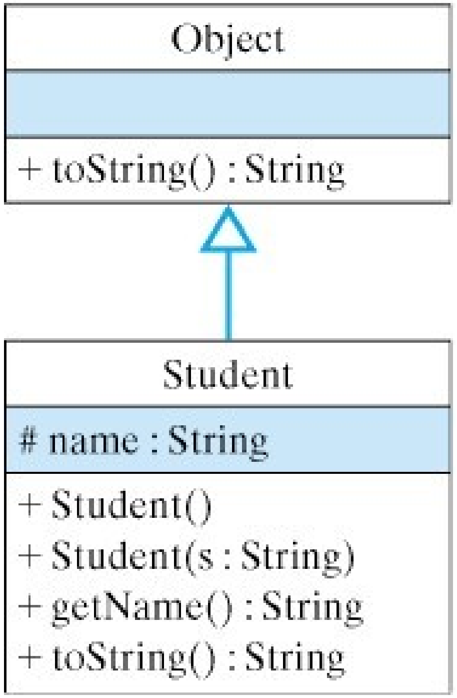

```
public class Student extends Object{
    protected String name;
    public Student(String s) {
        name = s;
    }
    public String getName() {
        return name;
    }
}
```

```
Student stu = new Student("Stewart");
System.out.println(stu.toString());   // Prints Student@cde100
```

Invoking the inherited toString() method.

Overriding an Inherited Method

- To *override* an inherited method means to give it a new definition in the subclass.
- Overriding a method (e.g., toString()) allows you to customize its behavior in the subclass.




```
public String toString() {
    return “My name is “ + name +
           “and I am a Student.”);
}
```

```
Student stu = new Student(“Stu”);
System.out.println(stu.toString()); // My name is Stu and I am a Student
```

Invoking the overridden toString() method.

Dynamic Binding and Polymorphism

- In *dynamic binding* (also called *late* or *runtime* binding) *a* method call is bound to the correct implementation of the method at runtime by the Java Virtual Machine.
- In *static binding* (for final and private methods) the Java compiler binds the method call to the correct method definition.
- *Polymorphism* (*poly* = many, *morph* = shape) : calling the same method can lead to different behaviors depending on the type of object on which the method is invoked.

The toString() method is polymorphic.

```
Object obj;                         // Static (declared) type: Object
obj = new Student("Stu");           // Actual (dynamic) type: Student
System.out.println(obj.toString()); // Print "My name is Stu…"
obj = new OneRowNim(11);            // Actual (dynamic) type: OneRowNim
System.out.println(obj.toString()); // Prints "nSticks = 11,player = 1"
```

Polymorphic Methods

- A *polymorphic method* behaves differently when called on different objects.
- Consider the following method definition.

```
public void polyMethod(Object obj) {
    System.out.println(obj.toString());  // Polymorphic
}
```

- In the following code segment, the first time polyMethod() is called, Java uses dynamic binding to associate toString() with the Student method. On the second call, it is associated with the OneRowNim toString() method.

```
Student stu = new Student("Stu");
polyMethod(stu);
OneRowNim nim = new OneRowNim();
polyMethod(nim);
```

Polymorphism and Object-Oriented Design

- Most print() and println() method use the *method signature* to determine which version of the method to invoke.
- So, print(10) matches the signature of print(int).

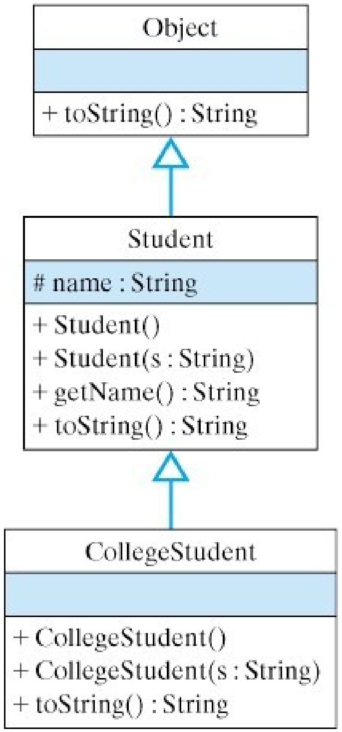
```
print(char c);           println(char c);
print(int i);            println(int i);
print(double d);         println(double d);
print(float f);          println(float f);
print(String s);         println(String s);
print(Object o);         println(Object o);
```

- But all objects are printed using print(Object o) which (likely) uses polymorphism and dynamic binding:

```
public void print(Object o) {
    System.out.print(o.toString());
}
```

Inheritance and Constructors

- Constructor methods are not inherited by subclasses.
- A subclass constructor can explicitly invoke a superclass method, including a superclass constructor, by using the super keyword.




```
public class CollegeStudent extends Student {
    public CollegeStudent() { }
    public CollegeStudent(String s) {
        super(s); // check out page 7
    }
    public String toString() {
       return "My name is " + name +
              " and I am a CollegeStudent.";
    }
}
```

In this context super() calls the superconstructor of the same signature to set the CollegeStudent's inherited name variable.

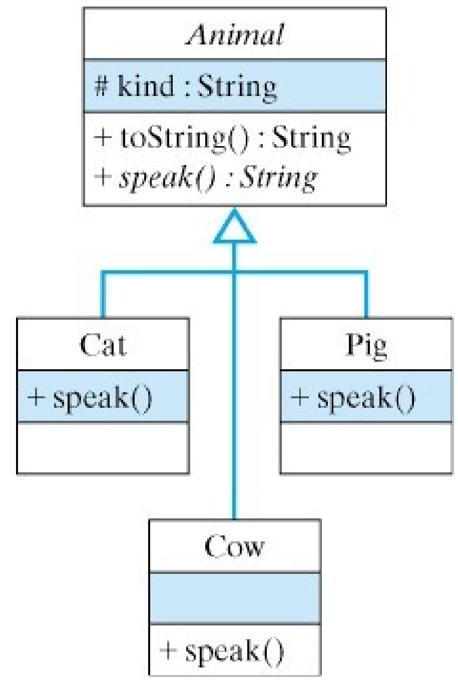
Abstract Classes and Java Interfaces

- In Java there are three forms of polymorphism:
  - Overriding an inherited method. (abovementioned)
  - Implementing an abstract method.
  - Implementing a Java interface.
- All three forms are based on the *dynamic binding mechanism*.

Implementing an Abstract Method

- Define the speak() method abstractly in the superclass.

```
public abstract class Animal { // keyword
    protected String kind; // Cow, pig, cat, etc.
    public Animal()  {  }
    public String toString() {
        return "I am a " + kind + " and I go " + speak();
    }
    public abstract String speak(); // Abstract method
}
```



- Implement speak() differently in each subclass.

```
public class Cat extends Animal {
    public Cat() {
        kind = "cat";
    }
    public String speak() {
        return "meow";
    }
}
```

```
public class Cow extends Animal {
    public Cow() {
        kind = "cow";
    }
    public String speak() {
        return "moo";
    }
}
```

Dynamic Binding and Extensibility

- Given the definitions of Animal, Cow, and Cat, we can get each kind of animal to speak in its own distinctive way:

```
Animal animal = new Cow();    // Animal's dynamic type is Cow
animal.speak();               // A cow goes moo (dynamic binding)
animal = new Cat();           // Animal's dynamic type is now Cat
animal.speak();               // A cat goes meow (dynamic binding)
```

- **Extensibility**: Given the definitions of Animal we can add new kinds of Animals to the hierarchy and program each to speak in their own distinctive way.

Implementing a Java Interface

- An *interface* (like ActionListener) is a Java class that contains only abstract methods and constants.
- Alternatively, we can define speak() as part of the Speakable interface.

```
public interface Speakable { // keyword
    public String speak(); // Abstract method
}
```

- Because speak() is no longer defined in Animal, we must use the cast operator when we invoke it.

```
public class Animal {
    protected String kind; // Cow, pig, cat, etc.
    public Animal()  {  }
    public String toString() {
        return "I am a " + kind + " and I go " + ((Speakable)this).speak();
    }
}
```

Implementing the Interface

- Subclasses of Animal can now implement the Speakable interface in their own distinct ways.

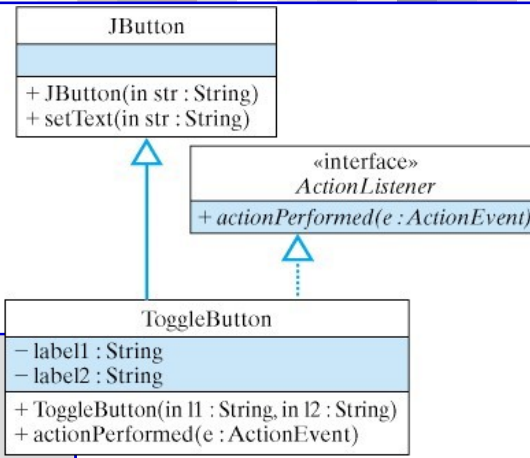
```
public class Cat extends Animal implements Speakable { // keyword
    public Cat() {
        kind = "cat";
    }
    public String speak() {
        return "meow";
    }
}
```

```
public class Cow extends Animal implements Speakable {
    public Cow() {
        kind = "cow";
    }
    public String speak() {
        return "moo";
    }
}
```

- Inheritance: A Cat is both an Animal and a Speakable.

Example: A Toggle Button

- Let’s use inheritance to create a special type of button.
- A ToggleButton toggles its label when it performs an action.

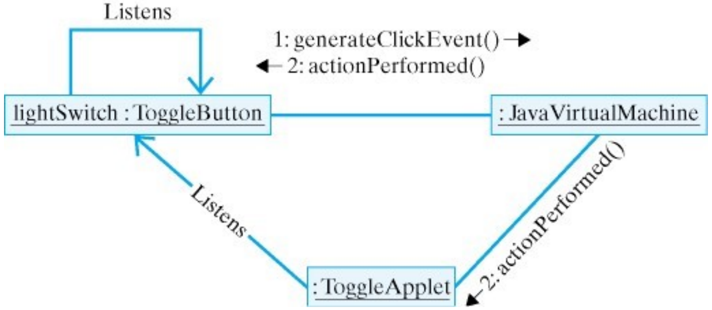



```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ToggleButton extends JButton implements ActionListener {
  private String label1;   // Toggle between two labels
  private String label2;

  public ToggleButton(String l1, String l2) { // Constructor
    super(l1);          // Use l1 as the default label
    label1 = l1;
    label2 = l2;
    addActionListener(this);
  }
  public void actionPerformed(ActionEvent e) {
    String tempS = label1;  // Swap the labels
    label1 = label2;
    label2 = tempS;
    setText(label1);
  } // actionPerformed()
} // ToggleButton
```

Clicking a ToggleButton toggles its own label.

Using the Toggle Button

- Both the Applet and the ToggleButton are listeners.

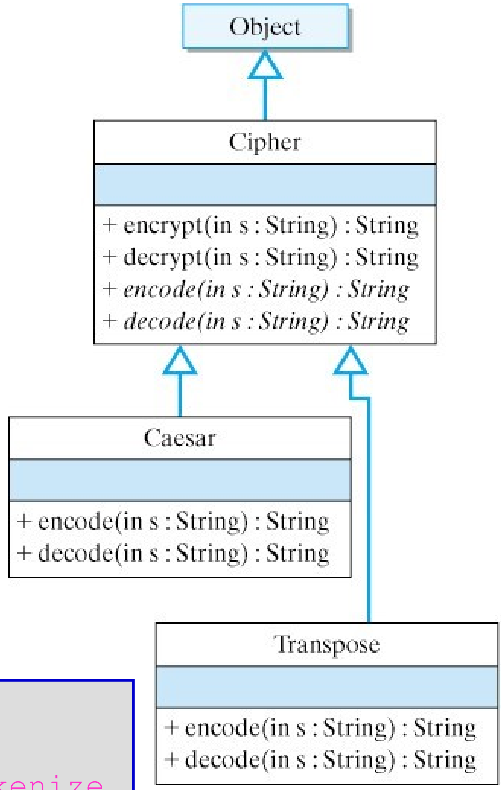



```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ToggleApplet extends Japplet implements ActionListener {
  private ToggleButton lightSwitch;

  public void init() {
    lightSwitch = new ToggleButton ("off","on");
    getContentPane().add(lightSwitch);
    lightSwitch.addActionListener(this);
  } // init()

  public void actionPerformed(ActionEvent e)  {
    showStatus("The light is " + lightSwitch.getText());
  } // actionPerformed()
} // ToggleApplet
```

Clicking a ToggleButton performs an action.

Example: Cipher Class Hierarchy

- A cipher encrypts text.
- Design a hierarchy of classes to implement various historical ciphers.
- The Cipher superclass implements the encrypt() and decrypt() method.
- The abstract encode() and decode() methods are implemented in the subclasses.



The encrypt() method calls the polymorphic encode() method.

```
public String encrypt(String s) {  // Encrypt a sentence
    StringBuffer result = new StringBuffer("");
    StringTokenizer words = new StringTokenizer(s); // Tokenize
    while (words.hasMoreTokens()) {          // Encode each word
        result.append(encode(words.nextToken()) + " ");
    }
    return result.toString();                // Return result
} // encrypt()
```

The Cipher Class

- The Cipher class encapsulates the features that all ciphers have in common.

```
import java.util.*;

public abstract class Cipher {
  public String encrypt(String s) {
    StringBuffer result = new StringBuffer("");          // Use a StringBuffer
    StringTokenizer words = new StringTokenizer(s); // Break s into its words
    while (words.hasMoreTokens()) {                      // For each word in s
      result.append(encode(words.nextToken()) + " ");    //  Encode it
    }
    return result.toString();                             // Return the result
  } // encrypt()

  public String decrypt(String s) {
    StringBuffer result = new StringBuffer("");         // Use a StringBuffer
    StringTokenizer words = new StringTokenizer(s);     // Break s into words
    while (words.hasMoreTokens()) {                     // For each word in s
      result.append(decode(words.nextToken()) + " ");   //  Decode it
    }
    return result.toString();                       // Return the decryption
  } // decrypt()

  public abstract String encode(String word);             // Abstract methods
  public abstract String decode(String word);
} // Cipher
```

The Caesar Class

- The **Caesar** class implements the Caesar cipher.
- A Caesar cipher shifts every letter by 3. So, *a* becomes *d,* and *b* becomes *e,* and *z* becomes *c.*

```
public class Caesar extends Cipher {
  public String encode(String word) {
    StringBuffer result = new StringBuffer(); // Initialize a string buffer
    for (int k = 0; k < word.length(); k++) { // For each character in word
      char ch = word.charAt(k);               //  Get the character
      ch = (char)('a' + (ch -'a'+ 3) % 26);   //  Perform caesar shift
      result.append(ch);                    //  Append it to new string
    }
    return result.toString();               // Return the result as a string
  } // encode()

  public String decode(String word) {
    StringBuffer result = new StringBuffer(); // Initialize a string buffer
    for (int k = 0; k < word.length(); k++) { // For each character in word
    char ch = word.charAt(k);                 //  Get the character
       ch = (char)('a' + (ch - 'a' + 23) % 26); //  Perform reverse shift
       result.append(ch);                     //  Append it to new string
    }
    return result.toString();             // Return the result as a string
  } // decode()
} // Caesar
```

The Transpose Class

- The **Transpose** class implements a transposition cipher, which rearranges the letters in the text.
- In this case we simply reverse the letters in each word.

```
public class Transpose extends Cipher {

  public String encode(String word) {
    StringBuffer result = new StringBuffer(word); // Initialize a string buffer
    return result.reverse().toString();           // Just reverse the word
  } // encode()

public String encode(String word) {
return encode(word);                         // Just call encode()
  } // encode()

} // Transpose
```

Case Study: A Two-Player Game Hierarchy

- Redesign the OneRowNim game to fit within a hierarchy of two-player games.
- We use inheritance and polymorphism to design an extensible object-oriented game hierarchy.
- Goal: Generalize OneRowNim to create a superclass that contains features common to all two-player games.
- Goal: Allow both human and computer players.
- Goal: Allow games to be played with different user interfaces.

Sample: Command-line Run of OneRowNim

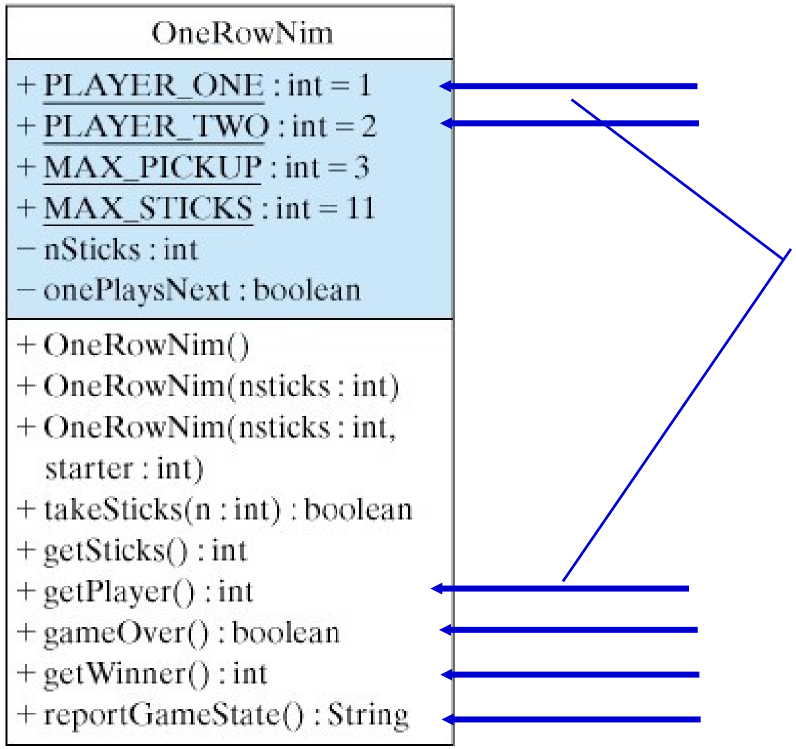
```
How many computers are playing, 0, 1, or 2? 1
What type of player, NimPlayerBad = 1, or NimPlayer = 2 ? 2

*** The Rules of One Row Nim ***
(1) A number of sticks between 7 and 11 is chosen.
(2) Two players alternate making moves.
(3) A move consists of subtracting between 1 and
         3 sticks from the current number of sticks.
(4) A player who cannot leave a positive
         number of sticks for the other player loses.

Player 2 is a  NimPlayer
Sticks left: 11 Who's turn: Player 1
You can pick up between 1 and 3 : 3

Sticks left: 8 Who's turn: Player 2 NimPlayer takes 3 sticks.

Sticks left: 5 Who's turn: Player 1
You can pick up between 1 and 3 : 2

Sticks left: 3 Who's turn: Player 2 NimPlayer takes 2 sticks.

Sticks left: 1 Who's turn: Player 1
You can pick up between 1 and 1 : 1

Sticks left: 0 Game over! Winner is Player 2 Nice game.
```

Careful: NimPlayer is an optimal player.

Human is player 1 and computer is player 2.

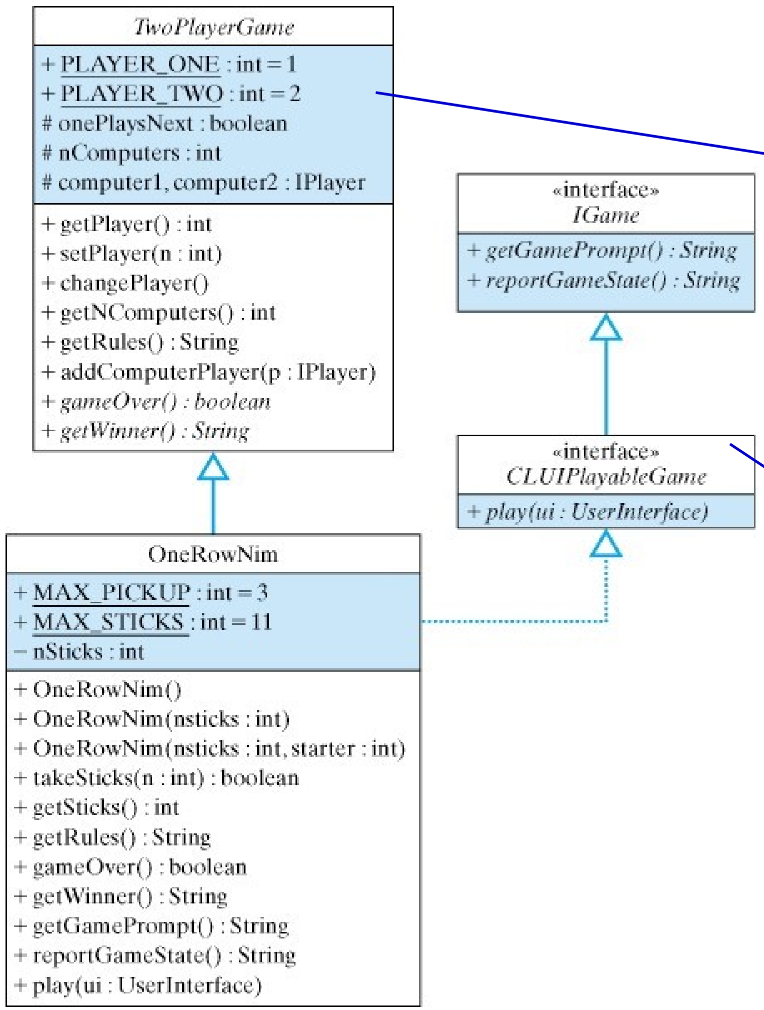
The computer wins.

Generalizing OneRowNim



Elements common to all two-player games will move up to the superclass and be inherited by OneRowNim and other games.

The Two-Player Game Hierarchy



Elements common to all two-player games are defined in the superclass and inherited by OneRowNim.

Java interfaces are used to define the user interface. A CLUIPlayableGame (command-line game) is a subclass of IGame (interactive game).

The TwoPlayerGame Superclass

- The TwoPlayerGame class is simple to define.
- It contains a default getRules() method that is meant to be overridden by its subclasses.
- It contains abstract gameOver() and getWinner() methods that are meant to be implemented in its subclasses.
- Its players are defined as IPlayer objects, which can be either humans or computers. An IPlayer is any object that implements the makeAMove() method.
- Notice how general the makeAMove() method is:

```
public interface IPlayer {
    public String makeAMove(String prompt);
}
```

The CLUIPlayableGame Interface

- The purpose of the CLUIPlayableGame and IGame interfaces is to create a connection between any two-player game and a command-line interface.
- Their methods handle the interaction between a TwoPlayerGame and a UserInterface.
- Note that the play() method, which will contain the game's control loop, takes a UserInterface parameter.
- Implementation is game-dependent and is defined in the OneRowNim class.

```
public interface CLUIPlayableGame extends IGame {
    public abstract void play(UserInterface ui);
}

public interface IGame {
    public String getGamePrompt();
    public String reportGameState();
} //Igame
```

The UserInterface Interface

- The purpose of the UserInterface interface provides method signatures for objects that can serve as a user interface for our games.
- Note that the methods are the same as those in KeyboardReader.

```
public interface UserInterface {
    public String getUserInput();
    public void report(String s);
    public void prompt(String s);
}
```

- In fact, to turn our KeyboardReader into a UserInterface we can amend its defintion as follows:

```
public class KeyboardReader implements UserInterface …
```

Interfaces or Abstract Classes?

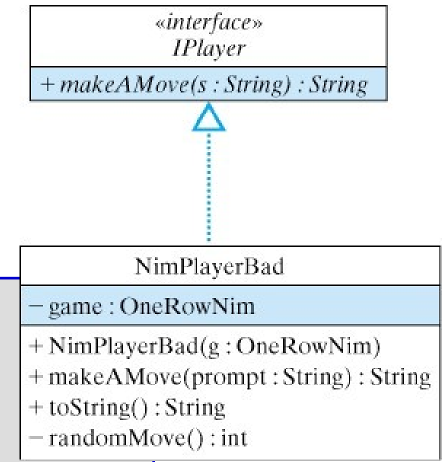
- Why do we use interfaces to define our method signatures rather than defining abstract methods in the TwoPlayerGame class?
- Using interfaces increases flexibility and extensibility. Interfaces can be attached to any class.
- Methods contained in the TwoPlayerGame class should be those that are necessary to define that *type* of object. The gameOver() and getWinner() methods are necessary to the definition of a game.
- Interface methods typically define certain *rules* that the objects will play. This seems to be true of play(), getGamePrompt(), and reportGameState().

The Revised OneRowNim Class

- The gameOver(), getWinner(), getGamePrompt(), reportGameState() methods are virtually unchanged.
- The play() method has major revisions.
  - Takes either computer or human players.
  - On each loop, one or the other player moves.
  - The makeAMove() method is used.

An IPlayer for OneRowNim

- To write a computer player for OneRowNim we define a class that implements the IPlayer interface.

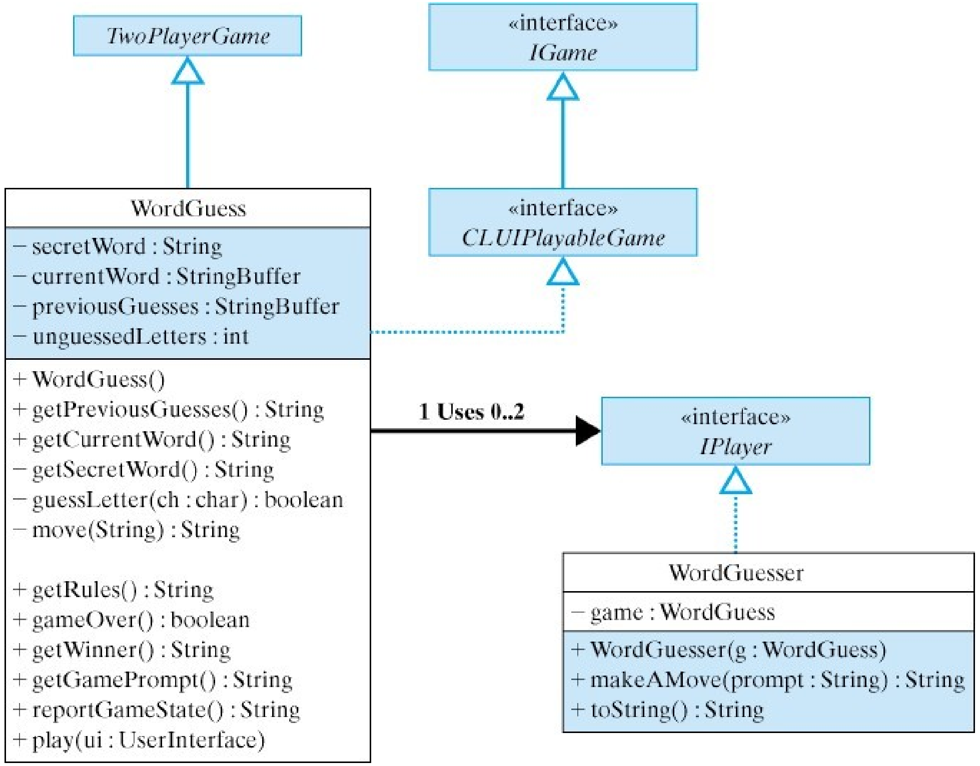



```java
public class NimPlayerBad implements IPlayer {
    private OneRowNim game;
    public NimPlayerBad (OneRowNim game) {
        this.game = game;
    } // NimPlayerBad()
    public String makeAMove(String prompt) {
        return "" + randomMove();
    } // makeAMove()
    private int randomMove() {
        int sticksLeft = game.getSticks();
        return 1 + (int)(Math.random() *
           Math.min(sticksLeft, game.MAX_PICKUP));
    } // randomMove()
    public String toString() {
      String className =
          this.getClass().toString(); // Gets 'class NimPlayerBad'
      return className.substring(5);  // Cut off the word 'class'
    } // toString()
} // NimPlayerBad
```

Just makes a random move.

Extending the TwoPlayerGame Hierarchy

- Let's extend TwoPlayerGame to play another game.
- WordGuess plays a form of Hang Man.



Technical Terms

abstract method
ciphertext
cryptography
interface
plaintext
polymorphism
static (declared) type
transpositions cipher

actual type (dynamic type)
class inheritance
dynamic (late) binding
overloaded method
polymorphic method
static (early) binding
substitution cipher

Summary Of Important Points

- *Inheritance* is an object-oriented mechanism whereby subclasses inherit the public and protected instance variables and methods from their superclasses.
- *Dyamic binding* (or late binding) is the mechanism that associates a method call with the correct implementation of the method at run time.
- *Static binding* associates the method call with the method implementation at compile time.
- *Polymorphism* refers to the fact that a method call can result in different behaviors depeding on object on which it is invoked.
- A *static type* is a variable's declared type. A *dynamic type* is the type of the object assigned to the variable.

Summary Of Important Points (cont)

- An abstract method is a method definition that lacks an implementation. Only its signature is given. An abstract class contains one or more abstract methods. It can be subclassed but not instantiated.
- A Java interface is a class that contains only method signatures and (possibly) constant declarations, but no variables.
- A Java interface can be implemented by any class that provides implementations to all of its abstract methods.