

Introduction to Java Programming

Class Hierarchies

CS501

Objectives

- ▶ Inheritance, class hierarchies and code reuse
- ▶ ADTs and Interfaces

Inheritance and Class Hierarchies

ADTs and Interfaces

Inheritance by Example

- ▶ A computer has
 - ▶ manufacturer
 - ▶ processor
 - ▶ RAM
 - ▶ disk

Computer
String manufacturer String processor int ramSize int diskSize double processorSpeed
int getRamSize() int getDiskSize() double getProcessorSpeed() Double computePower() String toString()

Inheritance by Example (cont.)

```
/** Class that represents a computers */  
public class Computer {  
    // Data fields  
    private String manufacturer;  
    private String processor;  
    private double ramSize;  
    private int diskSize;  
    private double processorSpeed;
```

Inheritance by Example (cont.)

```
// Methods
/** Initializes a Computer object with all properties specified
    @param man The computer manufacturer
    @param processor The processor type
    @param ram The RAM size
    @param disk The disk size
    @param procSpeed The processor speed
 */
public Computer(String man, String processor, double ram, int disk, double procSpeed) {
    manufacturer = man;
    this.processor = processor;
    ramSize = ram;
    diskSize = disk;
    processorSpeed = procSpeed;
}
```

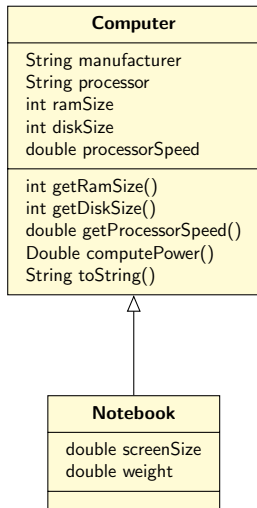
Inheritance by Example (cont.)

```
public double computePower()
{ return ramSize * processorSpeed; }
public double getRamSize() { return ramSize; }
public double getProcessorSpeed()
{ return processorSpeed; }
public int getDiskSize() { return diskSize; }
// insert other accessor and modifier methods here

public String toString() {
    String result = "Manufacturer: " + manufacturer +
        "\nCPU: " + processor +
        "\nRAM: " + ramSize + " megabytes" +
        "\nDisk: " + diskSize + " gigabytes" +
        "\nProcessor speed: " + processorSpeed +
            " gigahertz";
    return result;
}
}
```

Inheritance by Example (cont.)

- ▶ A Notebook has all the properties of Computer,
 - ▶ manufacturer
 - ▶ processor
 - ▶ RAM
 - ▶ Disk
- ▶ plus,
 - ▶ screen size
 - ▶ weight



Inheritance by Example (cont.)

```
/** Class that represents a notebook computer */  
public class Notebook extends Computer {  
    // Data fields  
    private double screenSize;  
    private double weight;  
    . . .  
}
```

- ▶ The data fields declared in `Computer` are also available to `Notebook`: they are **inherited**
- ▶ The methods declared in `Computer` are also available to `Notebook`: they are **inherited**
 - ▶ But `Notebook` still needs its own constructor for initializing its notebook-specific data
 - ▶ Lets take a closer look at this

Constructors in a Subclass

- ▶ They begin by initializing the data fields inherited from the superclass(es)

```
super(man, proc, ram, disk, procSpeed);
```

- ▶ This invokes the superclass constructor with the signature

```
Computer(String man, String processor, double ram, int
```

- ▶ They then initialize the data specific to their class, in this case to notebooks

```
screenSize = screen;  
weight = wei;
```

Constructors in a Subclass (cont.)

```
// methods
/** Initializes a Notebook object with all properties specified
 * @param man The computer manufacturer
 * @param processor The processor type
 * @param ram The RAM size
 * @param disk The disk size
 * @param procSpeed The processor speed
 * @param screen The screen size
 * @param wei The weight
 */
public Notebook(String man, String processor, double ram, int disk,
                double procSpeed, int screen, double wei) {
    super(man, processor, ram, disk, procSpeed);
    screenSize = screen;
    weight = wei;
}
```

The No-Parameter Constructor

- ▶ If the execution of any constructor in a subclass does not invoke a superclass constructor – an explicit call to `super()` – Java automatically invokes the no-parameter constructor for the superclass
- ▶ If no constructors are defined for a class, the no-parameter constructor for that class is provided by default
- ▶ However, if any constructors are defined, you must explicitly define a no-parameter constructor

Protected vs Private Data Fields

- ▶ Variables with private visibility cannot be accessed by a subclass
 - ▶ They are still there (they are inherited)
 - ▶ Just that to access them we have to use the methods defined in class `Computer`
 - ▶ An alternative is to declare them **protected** rather than **private**
- ▶ Variables with protected visibility (defined by the keyword `protected`) are accessible by any subclass or any class in the same package
- ▶ In general, it is better to use private visibility and to restrict access to variables to accessor methods

Is-a versus Has-a Relationships

- ▶ In an **is-a** or inheritance relationship, one class is a subclass of the other class
- ▶ In a **has-a** or aggregation relationship, one class has the other class as an attribute

Is-a versus Has-a Relationships

```
public class Computer {  
    private Memory mem;  
    ...  
}  
  
public class Memory {  
    private int size;  
    private int speed;  
    private String kind;  
    ...  
}
```

- ▶ A Computer has only one Memory
- ▶ But a Computer is not a Memory (i.e. not an is-a relationship)
- ▶ If a Notebook extends Computer, then the Notebook is-a Computer

Inheritance and Class Hierarchies

ADTs and Interfaces

Abstract Data Types

- ▶ An encapsulation of data and methods
- ▶ Allows for reusable code
- ▶ The user
 - ▶ need not know about the implementation of the ADT
 - ▶ interacts with the ADT using only public methods
- ▶ ADTs facilitate storage, organization, and processing of information
- ▶ The [Java Collections Framework](#) provides implementations of common ADTs

Interfaces

- ▶ A Java interface specifies or describes an ADT to the applications programmer:
 - ▶ the methods and the actions that they must perform
 - ▶ what arguments, if any, must be passed to each method
 - ▶ what result the method will return
- ▶ The interface can be viewed as a contract which guarantees how the ADT will function

Interfaces

- ▶ A class that implements the interface provides code for the ADT
- ▶ As long as the implementation satisfies the ADT contract, the programmer may implement it as he or she chooses
- ▶ In addition to implementing all data fields and methods in the interface, the programmer may add:
 - ▶ data fields not in the interface
 - ▶ methods not in the interface
 - ▶ constructors (an interface cannot contain constructors because it cannot be instantiated)

Example: ATM Interface

- ▶ An automated teller machine (ATM) enables a user to perform certain banking operations from a remote location.
- ▶ It must provide operations to:
 - ▶ verify a user's Personal Identification Number (PIN)
 - ▶ allow the user to choose a particular account
 - ▶ withdraw a specified amount of money
 - ▶ display the result of an operation
 - ▶ display an account balance
- ▶ A class that implements an ATM must provide a method for each operation

Example: ATM Interface

Interface:

- ▶ verify a user's PIN
- ▶ allow the user to choose a particular account
- ▶ withdraw a specified amount of money
- ▶ display the result of an operation
- ▶ display an account balance

Code:

```
public interface ATM {  
  
    /** Verifies a user's PIN.  
     * @param pin The user's PIN  
     */  
    boolean verifyPIN(String pin);  
  
    /** Allows user to select account.  
     * @return a String representing  
     *         the account selected  
     */  
    String selectAccount();  
}
```

Example: ATM Interface

Interface:

- ▶ verify a user's PIN
- ▶ allow the user to choose a particular account
- ▶ withdraw a specified amount of money
- ▶ display the result of an operation
- ▶ display an account balance

Code:

```
public interface ATM {  
  
    /** Verifies a user's PIN.  
     * @param pin The user's PIN  
     */  
    boolean verifyPIN(String pin);  
  
    /** Allows user to select account.  
     * @return a String representing  
     *         the account selected  
     */  
    String selectAccount();  
}
```

Example: ATM Interface

Interface:

- ▶ verify a user's PIN
- ▶ allow the user to choose a particular account
- ▶ withdraw a specified amount of money
- ▶ display the result of an operation
- ▶ display an account balance

Code:

```
public interface ATM {  
  
    /** Verifies a user's PIN.  
        @param pin The user's PIN  
    */  
    boolean verifyPIN(String pin);  
  
    /** Allows user to select account.  
        @return a String representing  
            the account selected  
    */  
    String selectAccount();  
}
```

Example: ATM Interface

Interface:

- ▶ verify a user's PIN
- ▶ allow the user to choose a particular account
- ▶ withdraw a specified amount of money
- ▶ display the result of an operation
- ▶ display an account balance

Code:

```
public interface ATM {  
  
    /** Verifies a user's PIN.  
     * @param pin The user's PIN  
     */  
    boolean verifyPIN(String pin);  
  
    /** Allows user to select account.  
     * @return a String representing  
     *         the account selected  
     */  
    String selectAccount();  
}
```


Example: ATM Interface

Interface:

- ▶ verify a user's PIN
- ▶ allow the user to choose a particular account
- ▶ withdraw a specified amount of money
- ▶ display the result of an operation
- ▶ display an account balance

Code:

```
/** Withdraws a specified amount
    of money
    @param account The account
                from which the money
                comes
    @param amount The amount of
                money withdrawn
    @return whether or not the
                operation is
                successful
 */
boolean withdraw(String account,
                 double amount);
```

Example: ATM Interface

Interface:

- ▶ verify a user's PIN
- ▶ allow the user to choose a particular account
- ▶ withdraw a specified amount of money
- ▶ display the result of an operation
- ▶ display an account balance

Code:

```
/** Displays the result of an
    operation
    @param account The account
        from which money was
        withdrawn
    @param amount The amount of
        money withdrawn
    @param success Whether or not
        the withdrawal took
        place
    */
void display(String account,
             double amount,
             boolean success);
```

Example: ATM Interface

Interface:

- ▶ verify a user's PIN
- ▶ allow the user to choose a particular account
- ▶ withdraw a specified amount of money
- ▶ display the result of an operation
- ▶ display an account balance

Code:

```
/** Displays an account balance
    @param account The account
        selected
 */
void showBalance(String account);
}
```

Note: Interfaces may include declaration of constants; these are accessible in classes that implement the interface

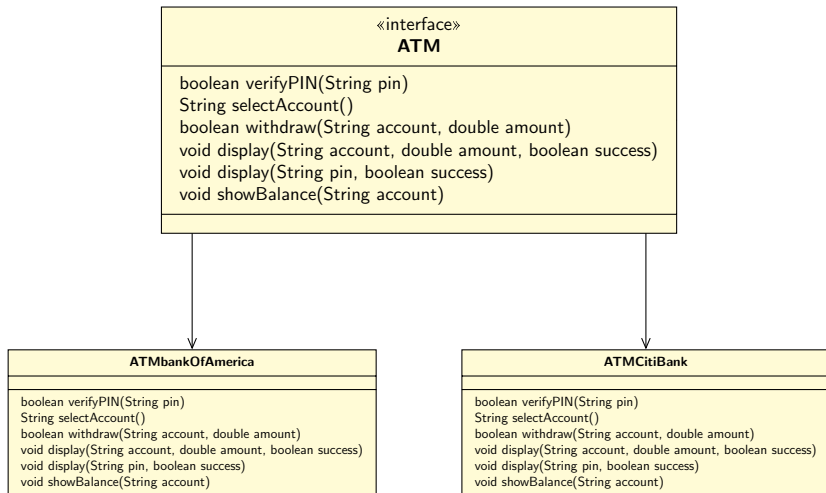
The `implements` clause

- ▶ For a class to implement an interface, it must end with the `implements` clause

```
public class ATMbankAmerica implements ATM  
public class ATMbankCiti implements ATM
```

- ▶ A class may implement more than one interface—their names are separated by commas

UML Diagram of Interface & Implementers



The `implements` Clause: Pitfalls

- ▶ The Java compiler verifies that a class defines all the abstract methods in its interface(s)
 - ▶ A syntax error will occur if a method is not defined or is not defined correctly
- ▶ You cannot instantiate an interface; it will cause an error

```
ATM anATM = new ATM();    // invalid statement
```

Declaring a Variable of an Interface Type

While you cannot instantiate an interface, you can declare a variable that has an interface type

```
/* expected type */  
ATMbankAmerica ATM0 = new ATMBankAmerica();  
  
/* interface type */  
ATM ATM1 = new ATMBankAmerica();  
ATM ATM2 = new ATMCitiBank();
```

The reason for wanting to do this will become clear when we discuss polymorphism