Data Structures

OOP and Class Hierarchies

CS284

Objectives

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

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:

```
1  public interface ATM {
2
3    /** Verifies a user's PIN.
4        @param pin The user's PIN
5    */
6    boolean verifyPIN(String pin);
7
8    /** Allows user to select account.
9        @return a String representing
10                the account selected
11   */
12   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:

```
1  public interface ATM {
2
3    /** Verifies a user's PIN.
4        @param pin The user's PIN
5    */
6    boolean verifyPIN(String pin);
7
8    /** Allows user to select account.
9        @return a String representing
10                the account selected
11   */
12   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:

```
1  public interface ATM {
2
3    /** Verifies a user's PIN.
4        @param pin The user's PIN
5    */
6    boolean verifyPIN(String pin);
7
8    /** Allows user to select account.
9        @return a String representing
10                the account selected
11   */
12   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:

```
1  public interface ATM {
2
3    /** Verifies a user's PIN.
4        @param pin The user's PIN
5    */
6    boolean verifyPIN(String pin);
7
8    /** Allows user to select account.
9        @return a String representing
10                the account selected
11   */
12   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:

```
1  /** Withdraws a specified amount
2        of money
3        @param account The account
4               from which the money
5               comes
6        @param amount The amount of
7               money withdrawn
8        @return whether or not the
9               operation is
10               successful
11     */
12   boolean withdraw(String account,
13                    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:

```
1   /** Displays the result of an
2       operation
3        @param account The account
4               from which money was
5              withdrawn
6        @param amount The amount of
7              money withdrawn
8        @param success Whether or not
9              the withdrawal took
10              place
11    */
12  void display(String account,
13                double amount,
14                 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:

```
1   /** Displays an account balance
2       @param account The account
3               selected
4   */
5   void showBalance(String account);
6 }
```

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

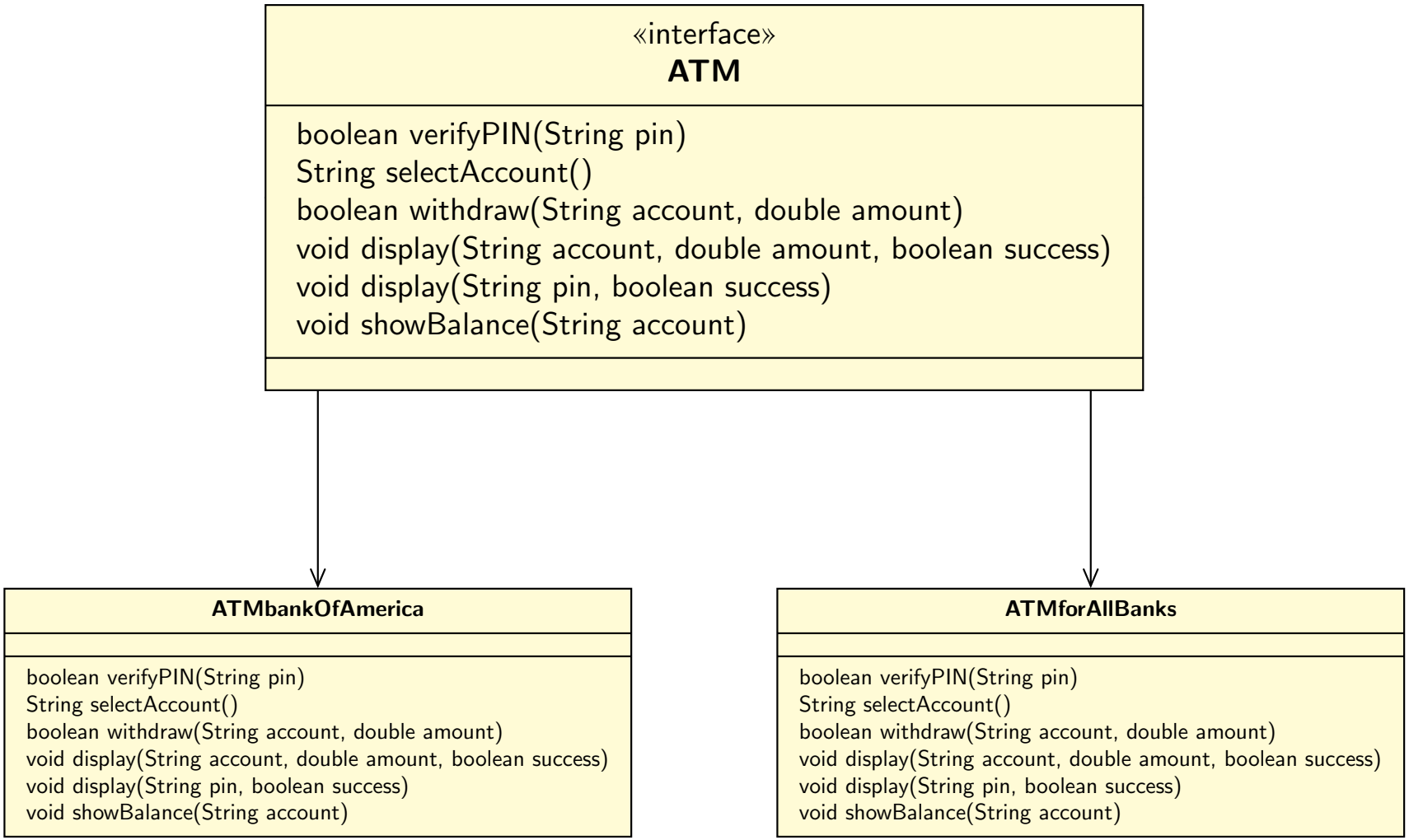
The `implements` clause

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

```
1 public class ATMbankAmerica implements ATM
2 public class ATMforAllBanks 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

```
1 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

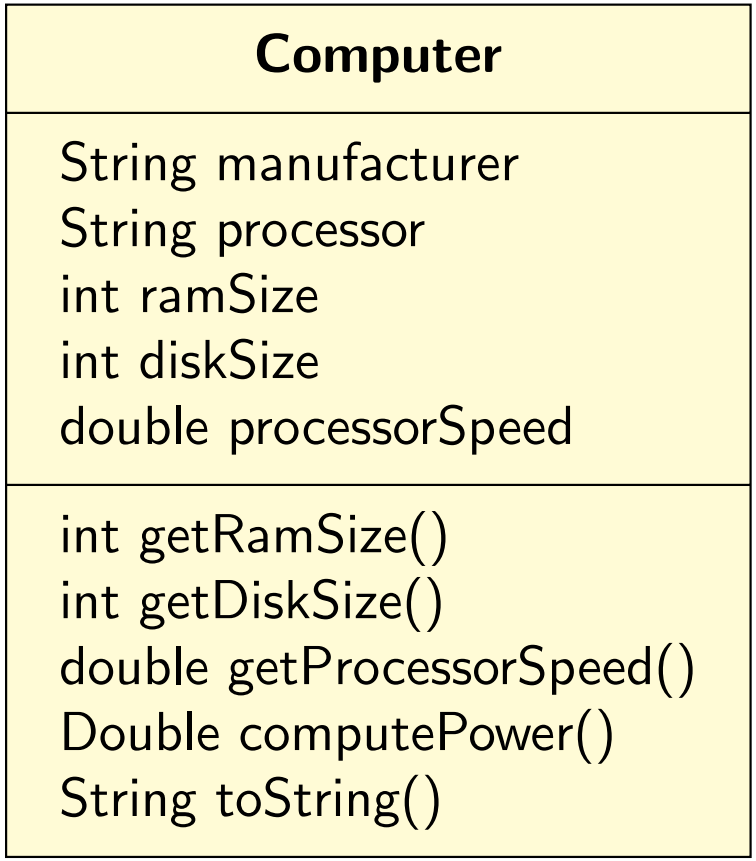
```
1 /* expected type */
2 ATMbankAmerica ATM0 = new ATMBankAmerica();
3
4 /* interface type */
5 ATM ATM1 = new ATMBankAmerica();
6 ATM ATM2 = new ATMforAllBanks();
```

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

Inheritance and Class Hierarchies

Inheritance by Example

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



Inheritance by Example (cont.)

```
1 /** Class that represents a computers */
2 public class Computer {
3   // Data fields
4   private String manufacturer;
5   private String processor;
6   private double ramSize;
7   private int diskSize;
8   private double processorSpeed;
```

Inheritance by Example (cont.)

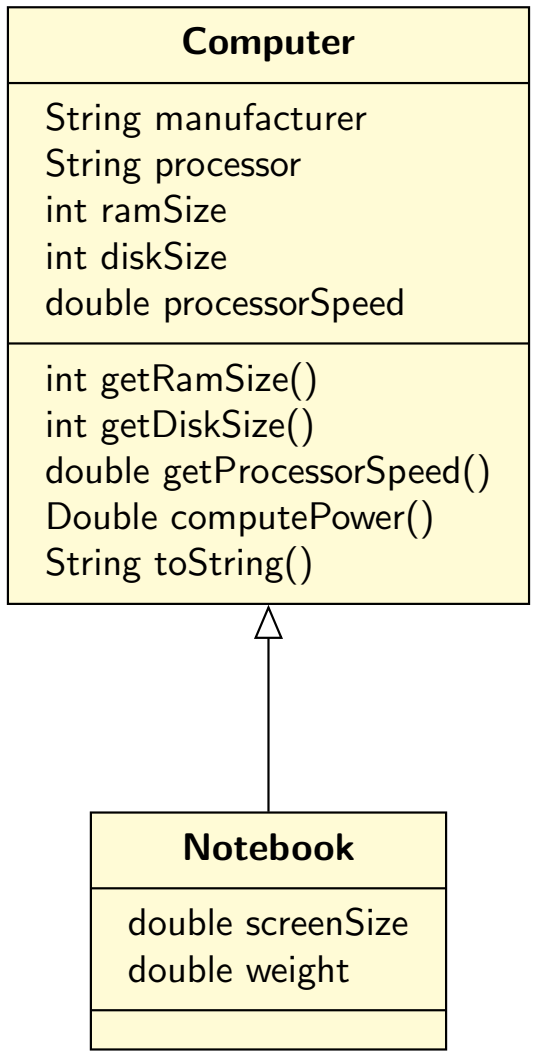
```
1  // Methods
2   /** Initializes a Computer object with all properties
      specified.
3     @param man The computer manufacturer
4     @param processor The processor type
5     @param ram The RAM size
6     @param disk The disk size
7     @param procSpeed The processor speed
8   */
9   public Computer(String man, String processor, double
      ram, int disk, double procSpeed) {
10    manufacturer = man;
11    this.processor = processor;
12    ramSize = ram;
13    diskSize = disk;
14    processorSpeed = procSpeed;
15  }
```

Inheritance by Example (cont.)

```
1   public double computePower()
2     { return ramSize * processorSpeed; }
3   public double getRamSize() { return ramSize; }
4   public double getProcessorSpeed()
5     { return processorSpeed; }
6   public int getDiskSize() { return diskSize; }
7   // insert other accessor and modifier methods here
8
9   public String toString() {
10    String result = "Manufacturer: " + manufacturer +
11        "\nCPU: " + processor +
12        "\nRAM: " + ramSize + " megabytes" +
13        "\nDisk: " + diskSize + " gigabytes" +
14        "\nProcessor speed: " + processorSpeed +
15            " gigahertz";
16    return result;
17  }
18 }
```

Inheritance by Example (cont.)

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



Inheritance by Example (cont.)

```
1 /** Class that represents a notebook computer */
2 public class Notebook extends Computer {
3   // Data fields
4   private double screenSize;
5   private double weight;
6     . . .
7 }
```

- 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 disk, double procSpeed)
```

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

```
1        screenSize = screen;
2        weight = wei;
```

Constructors in a Subclass (cont.)

```
1  // methods
2    //* Initializes a Notebook object with all properties
       specified.
3       @param man The computer manufacturer
4       @param processor The processor type
5       @param ram The RAM size
6       @param disk The disk size
7       @param procSpeed The processor speed
8       @param screen The screen size
9       @param wei The weight
10     */
11     public Notebook(String man, String processor, double
        ram, int disk, double procSpeed, double screen,
        double wei)
12     {
13        super(man, proc, ram, disk, procSpeed);
14        screenSize = screen;
15        weight = wei;
16     }
```

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

```
1  public class Computer {
2    private Memory mem;
3    ...
4  }
5
6  public class Memory {
7    private int size;
8    private int speed;
9    private String kind;
10   ...
11 }
```

- 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