Data Structures

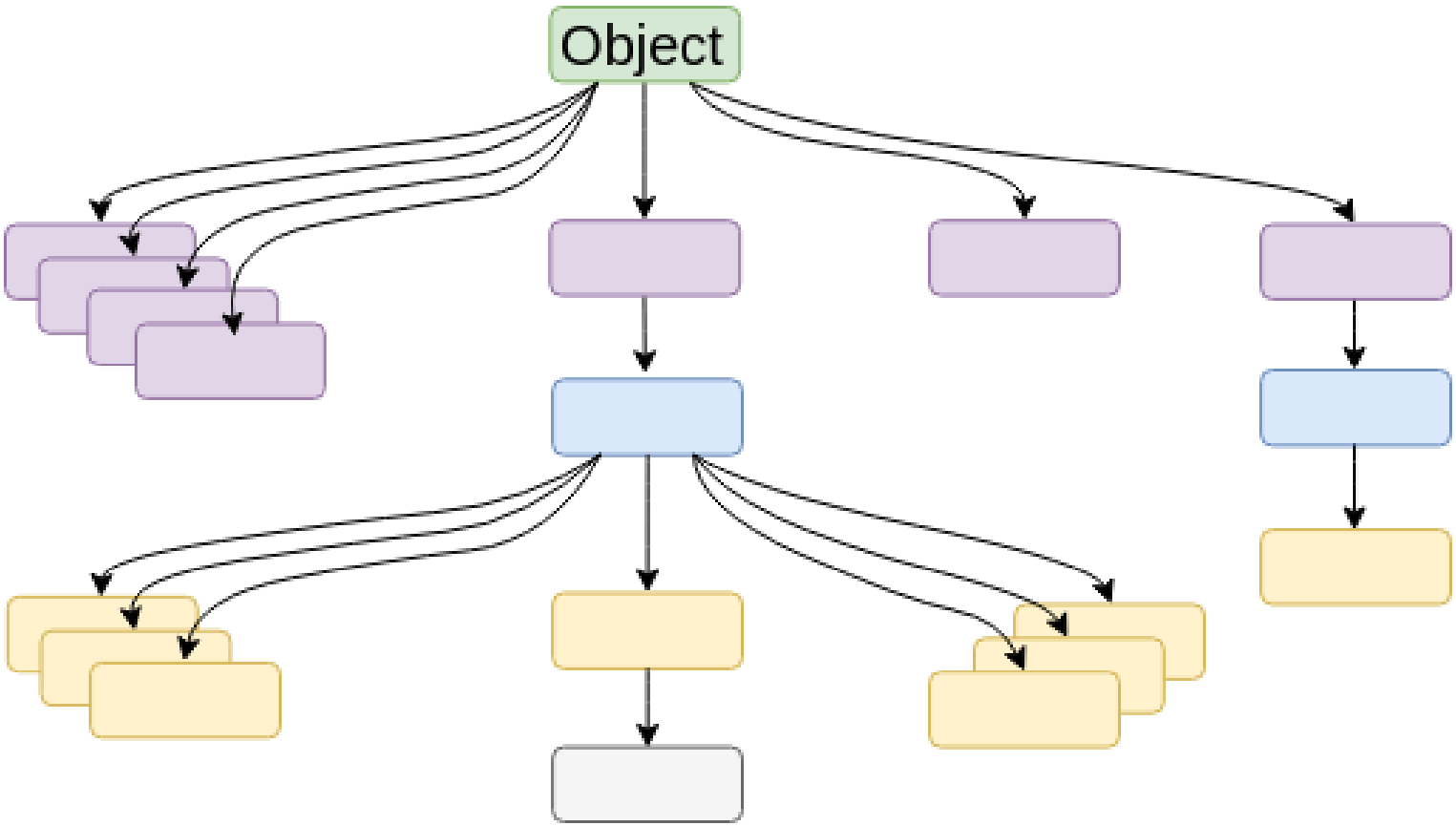
OOP and Class Hierarchies

CS284

Object-Oriented Programming

- Enables programmers to reuse previously written code
- To implement a new class similar to an existing class, programmer can *extend* the existing class, rather than *rewriting* the entire class
- This is called *inheritance*, the original class is called *superclass* (OOP: inheritance, encapsulation, abstraction, polymorphism)
- All Java classes are arranged in a *class hierarchy*

Java Class Hierarchy

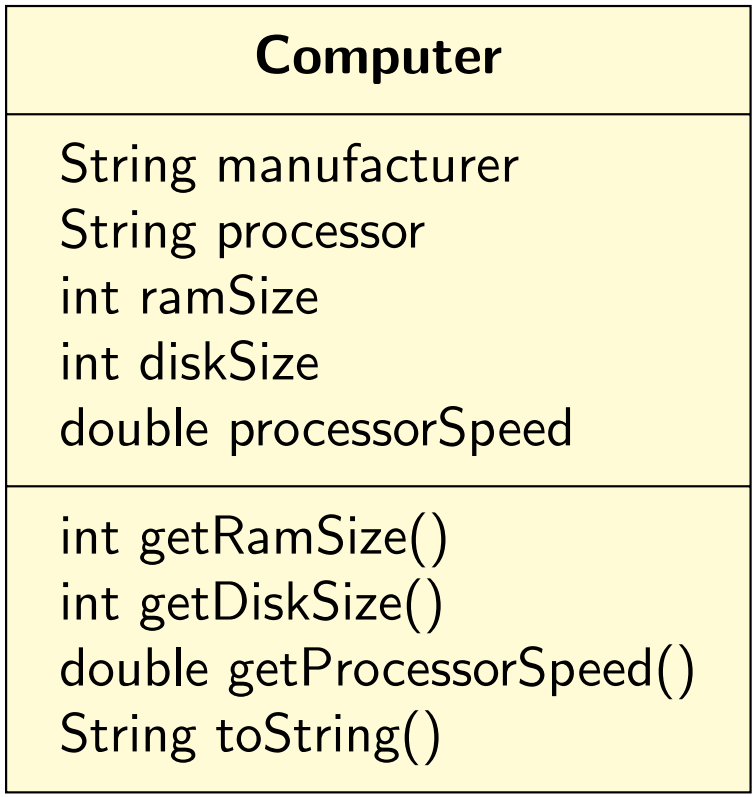


Inheritence

- Analogous to inheritance in human
  - We inherit knowledge and experience from parents
  - Our experience does not affect parents' experience
  - We develop our own knowledge and experience
- Inheritance allows to capture the idea that one thing is a *refinement* or *extension* of another
  - Allow programmers to reuse and extend previouly-defined code
- An example: Computer vs. Laptop
  - Suppose you are a Java developer who works for Dell, how to manage Dell's computer inventory?
  - Laptop, desktop

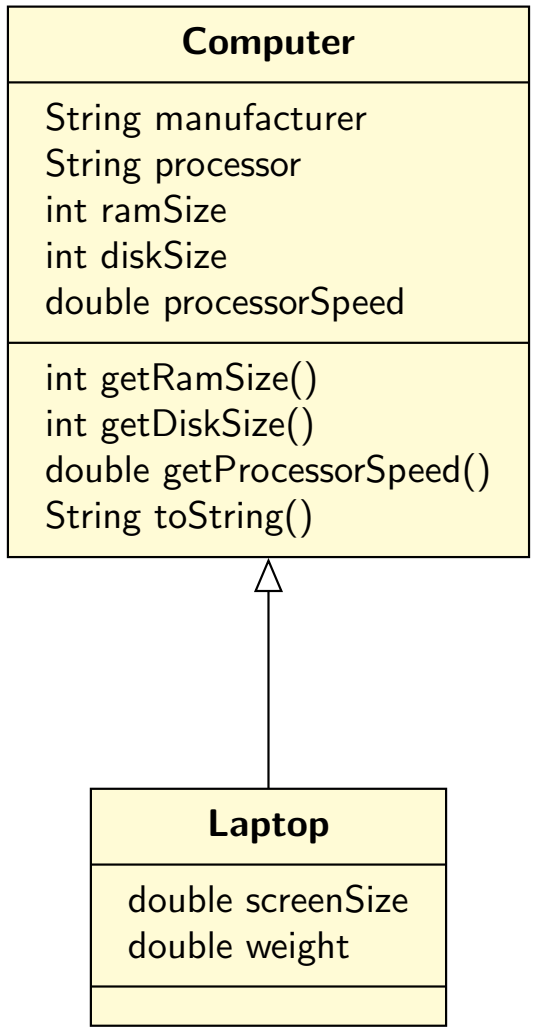
Inheritance Example: Laptop vs. Computer

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



Inheritance by Example: Laptop vs. Computer

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



Inheritance Example: Laptop vs. Computer

```
/** 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;

  public Computer(String man, String processor, double
  ram, int disk) {...}

  public double getRamSize() {...}
  public int getDiskSize() {...}
  public double getProcessorSpeed() {...}
  public String toString() {...}
 }
```

Inheritance by Example: Laptop vs. Computer

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

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

Constructors in a Subclass

- They begin by initializing the data fields inherited from the super class using `super`
- This invokes the superclass constructor with the signature
- They also need to initialize the data specific to their class

```
/* Initializes a Laptop object with all properties
specified. */
 public Laptop(String man, String processor, double
 ram, int disk, double screen, double weight)
 {
    super(man, proc, ram, disk);
    screenSize = screen;
    weight = weight;
 }
```

What Happens When `super` is Not Called?

- Java automatically invokes the no-parameter constructor for the superclass, e.g., `Computer()`
- Requirements on the super class's constructors
  - If no constructors are defined, the no-prameter constructor is called by default
  - However, if any constructors are defined, you must explicitly define a no-parameter constructor

Constructors in a Subclass (cont.)

- By calling `super`, we do not need to initialize inherited data fields from scratch
- Is the following code valid?

```
/* Initializes a Laptop object with all properties
specified. */
 public Laptop(String man, String processor, double
 ram, int disk, double screen, double weight)
 {
    manufacturer = man;
    processor = processor;
    ram = ram;
    disk = disk;
    screenSize = screen;
    weight = weight;
 }
```

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 `super` method
  - 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-a* Memory
- But a Computer is not a Memory (i.e. not an *is-a* relationship)
- If a Laptop extends Computer, then the Laptop *is-a* Computer

Abstract Data Types

- A *conceptual* model for data structures that is specify a collection of data fields and a list of methods that can be performed on the data fields;
- ADT is *independent of* the programming language, e.g., ADT of queue, stack
- ADTs are standardized in Java, Python, etc.
- Some ADTs are specific case of other ADTs, e.g., Stack and Queue are lists
- 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:

```
/** 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

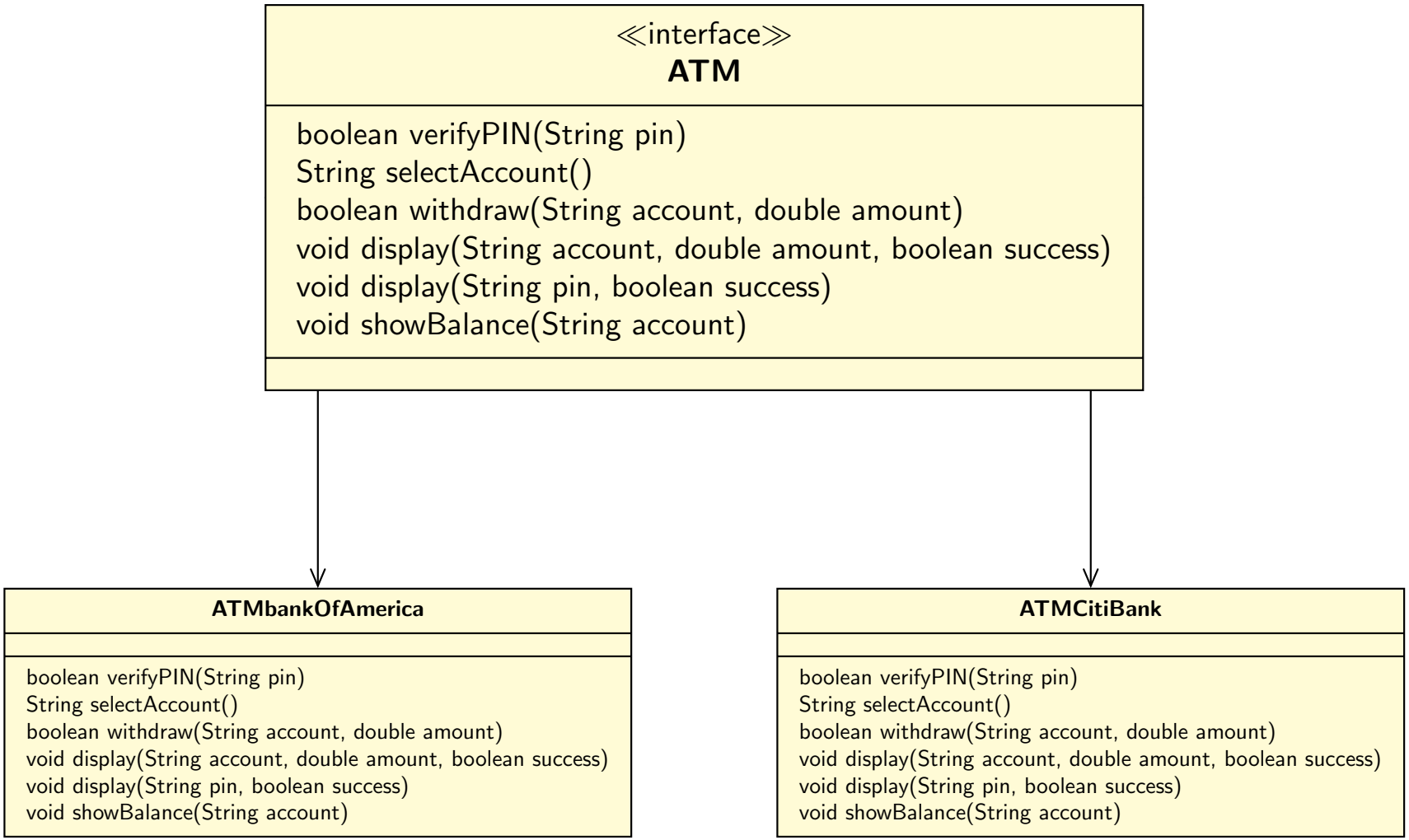
The `implements` clause

- 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

Review of Quiz 1

Suppose we have the following code:

```
public class Test{
  private int m;

  public static void func(){
     // some code...
  }
}
```

What is the correct way to replace line 2 so that variable m can be referenced within method func?

A. protected int m;

B. public int m;

C. static int m;

D. int m;

Review of Quiz 1

Define the following class:

- `BankAccount()`: Creates an account setting the balance to 0
- `BankAccount(double initialBalance)`
- `deposit(double amount)`
- `withdraw(double amount)`: Should print an error message if the balance is insufficient. This operation does not return any value.
- `getBalance()`
- `transfer(double amount, BankAccount destination)`. Should print an error message is there are insufficient funds in the origin account.

Review of Quiz 1

Error #1: What is wrong with the following code?

```
public class BankAccount{
    private double balance;

    public double deposit(amount){
        double new_balance = balance + amount;
        return new_balance;
}
}
```

Review of Quiz 1

Define the following class:

- `SavingsAccount(double rate)`: Creates a savings account with the given interest rate and 0 as balance. Eg. For a 1% interest rate the argument for this constructor would be 0.01.
- `addInterest()`: Deposits the interest w.r.t. the current balance.

Review of Quiz 1

Can SavingsAccount inherit BankAccount's balance?

```
public class SavingsAccount extends BankAccount {
    /*balance of bank account */
    private double rate = 0;

    public SavingsAccount(double rate) {
        super(0.0);
        this.rate = rate;
    }

    public void addInterests() {
        double current_balance = super.getBalance();
        super.deposit(current_balance * rate);
    }

    public static void main(String[] args) {
        SavingsAccount sa = new SavingsAccount(0.01);
    }
}
```