Stevens Institute of Technology
The Innovation University®

CS 570: Data Structures

Java & Object-Oriented Design (Chapter 1)

Instructor: Iraklis Tsekourakis

Email: itsekour@stevens.edu

Six Stages of Debugging

1. That can't happen.
2. That doesn't happen on my machine.
3. That shouldn't happen.
4. Why does that happen?
5. Oh, I see.
6. How did that ever work?

Textbook Companion Site

- http://bcs.wiley.com/he-bcs/Books?action=index&itemId=1119186528&bcsId=10110
- Google: Koffman and Wolfgang student companion site
  - Source code
  - Solutions to self check problems

Chapter Objectives

- Interfaces
- Inheritance and code reuse
- How Java determines which method to execute when there are multiple methods
- Abstract classes
- Abstract data types and interfaces
- `Object` class and overriding `Object` class methods
- `Exception` hierarchy
- Packages and visibility
- Class hierarchy for shapes

Week 2

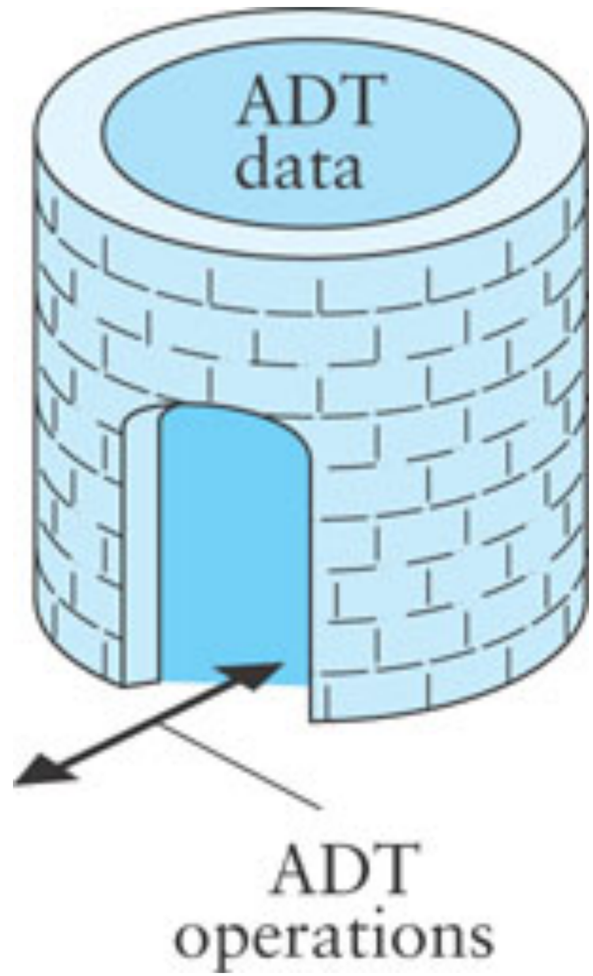
- Reading Assignment: Koffman and Wolfgang, Sections 1.1-1.5

ADTs, Interfaces, and the Java API

Section 1.1

ADTs

- Abstract Data Type (ADT)
- An encapsulation of data and methods
- Allows for reusable code
- The user need not know about the implementation of the ADT
- A user interacts with the ADT using only public methods



ADTs (cont.)

- ADTs facilitate storage, organization, and processing of information
- Such ADTs often are called *data structures*
- The Java Collections Framework provides implementations of common data structures

Interfaces

- An 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 (cont.)

- 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 (cont.)

Interface

- An automated teller machine (ATM) enables a user to perform certain banking operations from a remote location. It must support the following operations:
  - 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

Code

```
public interface ATM {
}
```

The keyword `interface` in the header indicates that an interface is being declared

Example: ATM Interface (cont.)

Interface

- An automated teller machine (ATM) enables a user to perform certain banking operations from a remote location. It must support the following operations:
  - 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

Code

```
public interface ATM {

  /** Verifies a user's PIN.
      @param pin The user's PIN
  */
  boolean verifyPIN(String pin);
}
```

Example: ATM Interface (cont.)

Interface

- An automated teller machine (ATM) enables a user to perform certain banking operations from a remote location. It must support the following operations:
  - 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

Code

```
public interface ATM {

  /** Verifies a user's PIN.
      @param pin The user's PIN
  */
  boolean verifyPIN(String pin);

  /** Allows the user to select an
      account.
      @return a String representing
              the account selected
  */
  String selectAccount();

}
```

Example: ATM Interface (cont.)

Interface

- An automated teller machine (ATM) enables a user to perform certain banking operations from a remote location. It must support the following operations:
  - 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

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 (cont.)

Interface

- An automated teller machine (ATM) enables a user to perform certain banking operations from a remote location. It must support the following operations:
  - 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

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 (cont.)

Interface

- An automated teller machine (ATM) enables a user to perform certain banking operations from a remote location. It must support the following operations:
  - 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

Code

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

Interfaces (cont.)

- The interface definition shows only headings for its methods
- Because only headings are shown, they are considered *abstract methods*
- Each abstract method must be defined in a class that implements the interface

Interface Definition

FORM:

```
public interface interfaceName {
    abstract method headings
    constant declarations
}
```

EXAMPLE:

```
public interface Payable {
    public abstract double calcSalary();
    public abstract boolean salaried();
    public static final
           double DEDUCTIONS = 25.5;
}
```

- Constants are defined in the interface
- `DEDUCTIONS` is accessible in classes that implement the interface

Interface Definition (cont.)

FORM:

```
public interface interfaceName {
    abstract method headings
    constant declarations
}
```

EXAMPLE:

```
public interface Payable {
    public abstract double calcSalary();
    public abstract boolean salaried();
    public static final
           double DEDUCTIONS = 25.5;
}
```

- The keywords `public` and `abstract` are implicit in each *abstract method* definition
- And keywords `public static final` are implicit in each *constant* declaration
- As such, they may be omitted

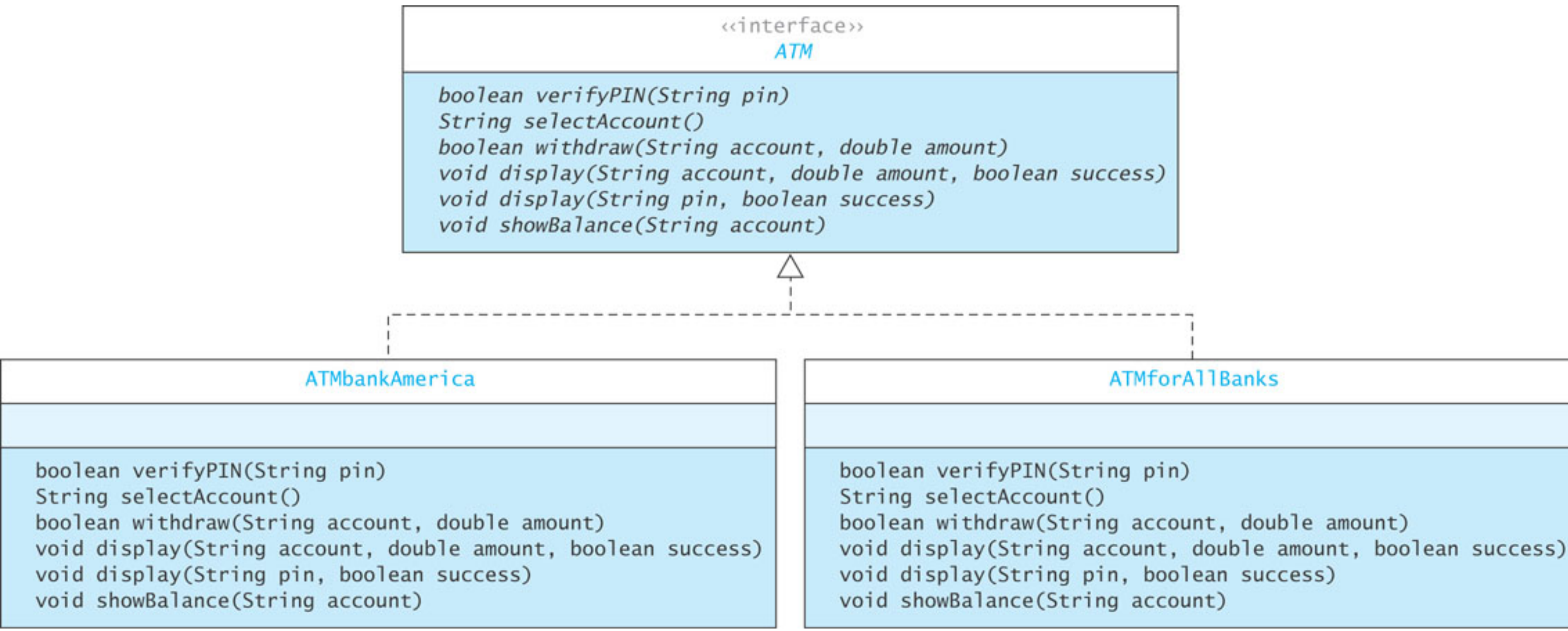
The `implements` Clause

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

```
public class ATMbankAmerica implements ATM
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:

```
Class ATMforAllBanks should be declared abstract; it does not
define method verifyPIN(String) in interface ATM
```

- If a class contains an undefined abstract method, the compiler will require that the class be declared an abstract class

The `implements` Clause: Pitfalls (cont.)

- You cannot instantiate an interface:

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

- Doing so will cause a syntax error:

```
interface ATM is abstract; cannot be instantiated
```

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 ATMforAllBanks();
```

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

Introduction to Object-Oriented Programming

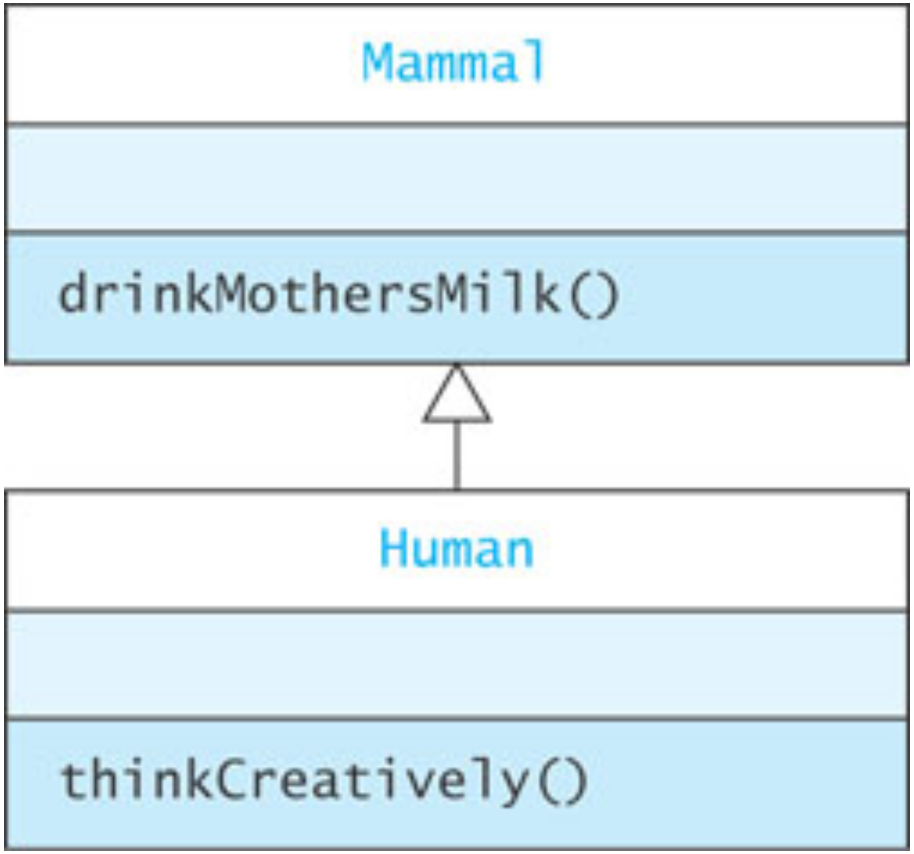
Section 1.2

Object-Oriented Programming

- Object-oriented programming (OOP) is popular because:
  - it enables *reuse* of previous code saved as *classes*
  - saves time because previously written code has been tested and debugged already
- If a new class is similar to an existing class, the existing class can be extended
- This extension of an existing class is called *inheritance*

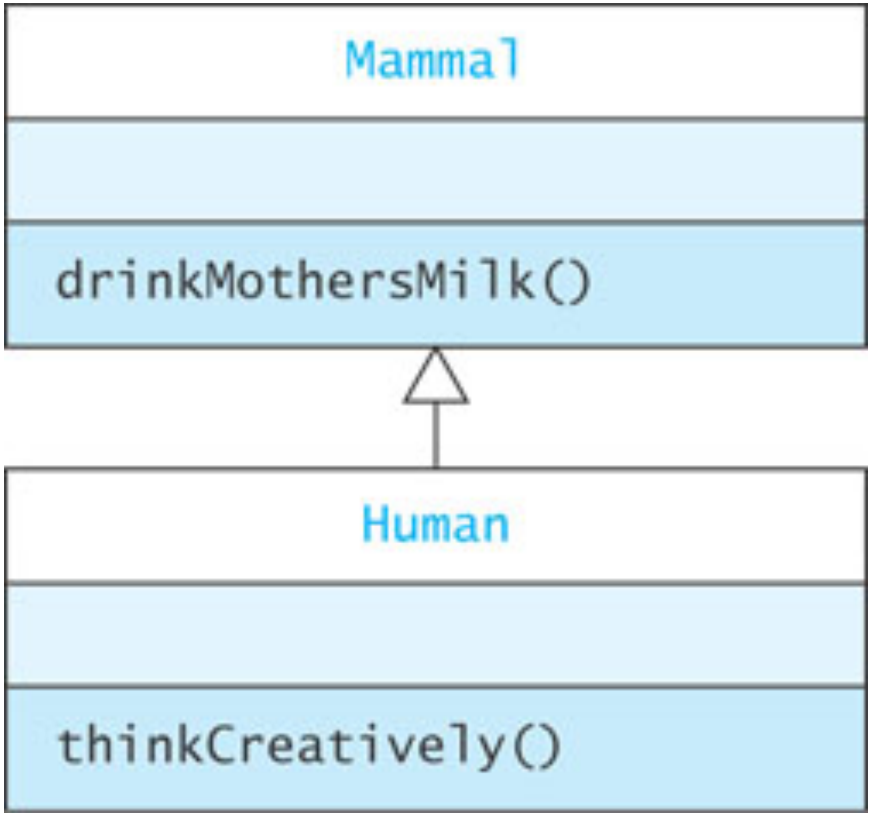
Inheritance

- A Human *is a* Mammal
- Human has all the data fields and methods defined by Mammal
- Mammal is the *superclass* of Human
- Human is a *subclass* of Mammal
- Human may define other variables and methods that are not contained in Mammal



Inheritance (cont.)

- Mammal has only method `drinkMothersMilk()`
- Human has method `drinkMothersMilk()` and `thinkCreatively()`
- Objects lower in the hierarchy are generally more powerful than their superclasses because of additional attributes



A Superclass and Subclass Example

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

Computer

`String manufacturer`
`String processor`
`int ramSize`
`int diskSize`
`double processorSpeed`

A Superclass and Subclass Example (cont.)

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

A Superclass and Subclass Example (cont.)

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

A Superclass and Subclass Example (cont.)

Computer

```
String manufacturer
String processor
int ramSize
int diskSize
double processorSpeed
```

```
int getRamSize()
int getDiskSize()
double getProcessorSpeed()
Double computePower()
String toString()
```

Notebook

```
double screenSize
double weight
```

A Superclass and Subclass Example (cont.)

- The constructor of a subclass begins by initializing the data fields inherited from the superclass(es)

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

which invokes the superclass constructor with the signature

```
Computer(String man, String processor, double ram,
        int disk, double procSpeed)
```

A Superclass and Subclass Example (cont.)


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

A Superclass and Subclass 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;
}
```

A Superclass and Subclass Example (cont.)


```
// Methods
/** Initializes a Computer object with a
  @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 proce
                double procSpeed) {
  manufacturer = man;
  this.processor = processor;
  ramSize = ram;
  diskSize = disk;
  processorSpeed = procSpeed;
}
```

Use of `this`

If you wrote this line as

```
processor = processor;
```

you would simply copy the variable processor to itself. To access the field, you need to prefix this:

```
this.processor = processor;
```

A Superclass and Subclass 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;
  }
}
```

A Superclass and Subclass Example (cont.)


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

  . . .
}
```

A Superclass and Subclass Example (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, double screen, double wei) {
    super(man, proc, 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 Visibility for Superclass Data Fields

- Variables with *private visibility* cannot be accessed by a subclass
- 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 (cont.)


```
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`**

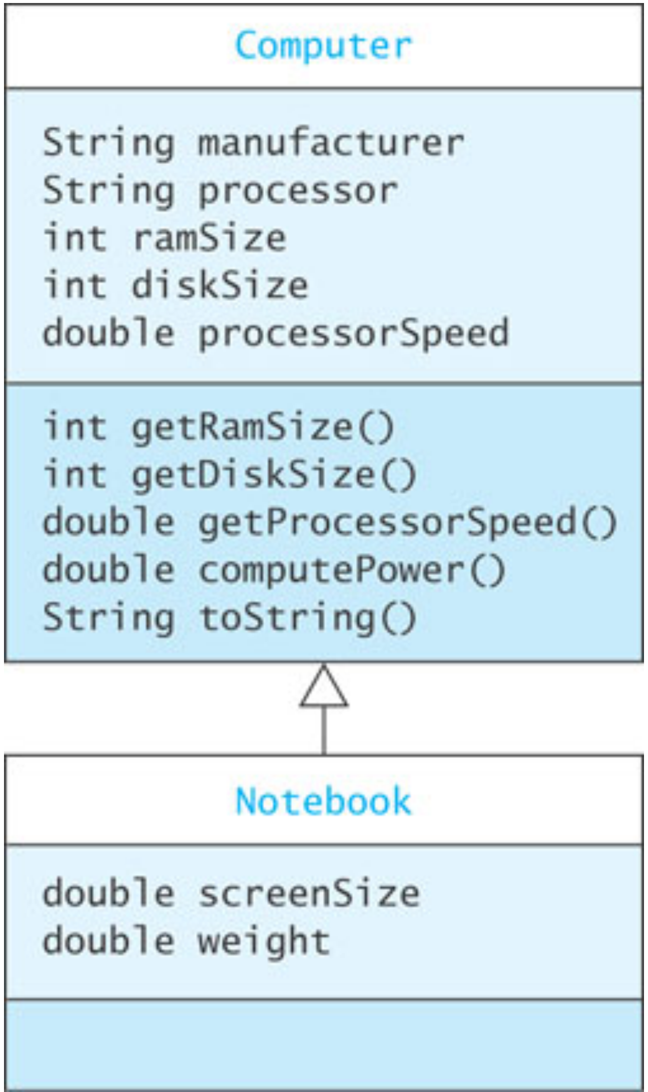
Method Overriding, Method Overloading, and Polymorphism

Section 1.3

Method Overriding

- Continuing the previous example, if we run:

```
Computer myComputer = new
   Computer("Acme", "Intel", 2, 160,
   2.4);
Notebook yourComputer = new
   Notebook("DellGate", "AMD", 4, 240,
   1.8, 15.0, 7.5);
System.out.println("My computer is:\n" +
   myComputer.toString());
System.out.println("Your computer is:\n"
   + yourComputer.toString());
```



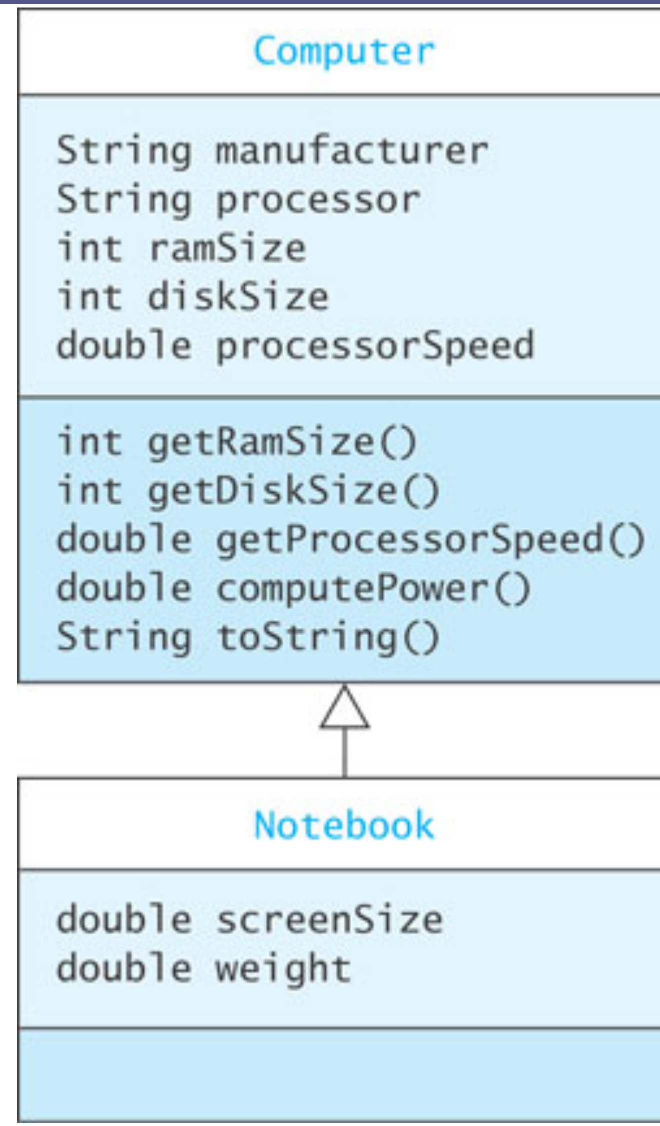
Method Overriding (cont.)

- the output would be:

```
My Computer is:
Manufacturer: Acme
CPU: Intel
RAM: 2.0 gigabytes
Disk: 160 gigabytes
Speed: 2.4 gigahertz

Your Computer is:
Manufacturer: DellGate
CPU: AMD
RAM: 4.0 gigabytes
Disk: 240 gigabytes
Speed: 1.8 gigahertz
```

- The `screensize` and `weight` variables are not printed because `Notebook` has not defined a `toString()` method



Method Overriding (cont.)

- To define a `toString()` for `Notebook`:

```
public String toString() {
  String result = super.toString() +
           "\nScreen size: " +
           screenSize + " inches" +
           "\nWeight: " + weight +
           " pounds";
  return result;

}
```

- Now `Notebook`'s `toString()` method will *override* `Computer`'s inherited `toString()` method and will be called for all `Notebook` objects

Method Overriding (cont.)

- To define a `toString()` for `Notebook`:

```
public String toString() {
  String result = super.toString() +
           "\nScreen size: " +
           screenSize + " inches" +
           "\nWeight: " + weight +
           " pounds";
  return result;
}
```

`super`.*`methodName`*`()`

Using the prefix **`super`** in a call to a method *methodName* calls the method with that name in the superclass of the current class

- No … *erride*
  Co … will
  be called for all `Notebook` objects

Method Overloading (cont.)

- Methods in the class hierarchy which have the same name, return type, and parameters *override* corresponding inherited methods
- Methods with the same name but different parameters are *overloaded*

Method Overloading (cont.)

- Take, for example, our Notebook constructor:

```
public Notebook(String man, String processor, double ram, int
disk, double procSpeed, double screen, double wei) {
     . . .
 }
```

- If we want to have a default manufacturer for a Notebook, we can create a constructor with six parameters instead of seven

```
public Notebook(String processor, double ram, int disk,
                 double procSpeed, double screen, double wei) {
     this(DEFAULT_NB_MAN, double ram, int disk, double procSpeed,
            double screen, double wei);
```

Method Overloading: Pitfall

- When *overriding* a method, the method must have the same name and the same number and types of parameters in the same order
- If not, the method will *overload*
- This error is common; the annotation `@Override` preceding an overridden method will signal the complier to issue an error if it does not find a corresponding method to override

```
@Override
public String toString() {
  . . .
}
```

- It is good programming practice to use the `@Override` annotation in your code

Polymorphism

- Polymorphism means *having many shapes*
- Polymorphism is a central feature of OOP
- It enables the JVM to determine at run time which of the classes in a hierarchy is referenced by a superclass variable or parameter

Polymorphism (cont.)

- For example, if you write a program to reference computers, you may want a variable to reference a `Computer` or a `Notebook`
- If you declare the reference variable as

  `Computer theComputer;` it can reference either a `Computer` or a `Notebook`—because a `Notebook` *is-a* `Computer`

Polymorphism (cont.)

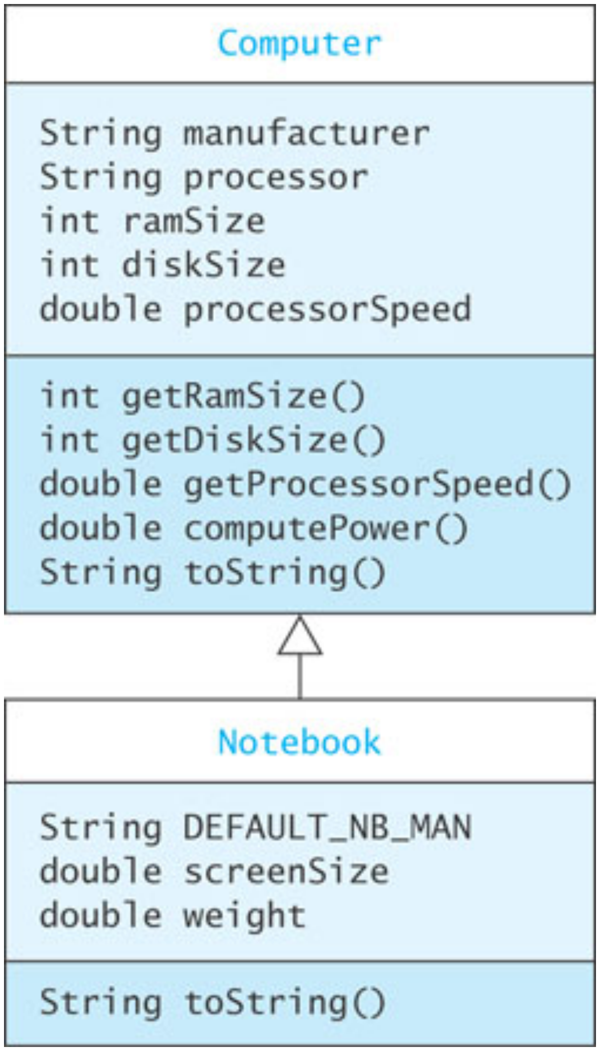
- Suppose the following statements are executed:

```
theComputer = new Notebook("Bravo", "Intel",
  4, 240, 2.4, 15, 7.5);
System.out.println(theComputer.toString());
```

- The variable `theComputer` is of type `Computer`,
- Which `toString()` method will be called, `Computer`'s or `Notebook`'s?

Polymorphism (cont.)

- The JVM correctly identifies the type of `theComputer` as `Notebook` and calls the `toString()` method associated with `Notebook`
- This is an example of *polymorphism*
- The type cannot be determined at *compile time*, but it can be determined at *run time*



Polymorphism (cont.)

- `Computer [] labComputers = new Computer[10];`
- `labComputers[i]` can reference either a `Computer` or a `Notebook` because `Notebook` is a subclass of `Computer`
- For `labComputers[i].toString()` polymorphism ensures that the correct `toString` method will be executed

Methods with Class Parameters

- Polymorphism simplifies programming when writing methods with class parameters
- If we want to compare the power of two computers (either `Computers` or `Notebooks`) we do not need to overload methods with parameters for two `Computers`, or two `Notebooks`, or a `Computer` and a `Notebook`
- We simply write one method with two parameters of type `Computer` and allow the JVM, using polymorphism, to call the correct method

Methods with Class Parameters (cont.)


```
/** Compares power of this computer and its argument computer
    @param aComputer The computer being compared to this computer
    @return -1 if this computer has less power,
            0 if the same, and
            +1 if this computer has more power.
*/
public int comparePower(Computer aComputer) {
    if (this.computePower() < aComputer.computePower())
            return -1;
    else if (this.computePower() == aComputer.computePower())
            return 0;
    else return 1;    }
```

Abstract Classes

Section 1.4

Abstract Classes

- An abstract class is denoted by using the word `abstract` in its heading:

```
visibility abstract class className
```

- An abstract class differs from an actual class (sometimes called a concrete class) in two respects:
  - An abstract class cannot be instantiated
  - An abstract class may declare abstract methods
- Just as in an interface, an abstract method is declared through a method heading:

```
visibility abstract resultType methodName (parameterList);
```

- A concrete class that is a subclass of an abstract class must provide an implementation for each abstract method

Abstract Classes (cont.)

- Use an abstract class in a class hierarchy when you need a **base class** for two or more subclasses that share some attributes
- You can declare some or all of the attributes and define some or all of the methods that are common to these subclasses
- You can also require that the actual subclasses implement certain methods by declaring these methods abstract

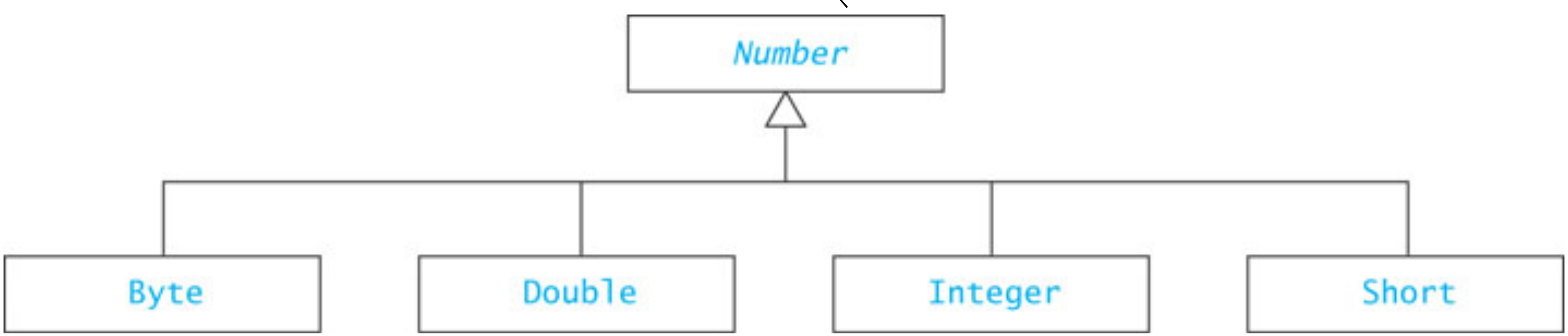
Example of an Abstract Class


```
public abstract class Food {
  public final String name;
  private double calories;
  // Actual methods
  public double getCalories () {
    return calories;
  }
  protected Food (String name, double calories) {
    this.name     = name;
    this.calories = calories;
  }
  // Abstract methods
  public abstract double percentProtein();
  public abstract double percentFat();
  public abstract double percentCarbs();
}
```

Java Wrapper Classes

- A *wrapper class* is used to store a primitive-type value in an object type

Declares what the (concrete) subclasses have in common



Interfaces, Abstract Classes, and Concrete Classes

- A Java *interface* can declare methods, but cannot implement them
- Methods of an interface are called abstract methods.
- An *abstract class* can have:
  - abstract methods (no body)
  - concrete methods (with a body)
  - data fields
- Unlike a concrete class, an *abstract class*
  - cannot be instantiated
  - can declare abstract methods which *must* be implemented in all *concrete* subclasses

Abstract Classes and Interfaces

- Abstract classes and interfaces cannot be instantiated
- An abstract class *can* have constructors!
  - *Purpose:* initialize data fields when a subclass object is created
  - The subclass uses `super(…)` to call the constructor
- An abstract class may *implement* an interface, but need not define all methods of the interface
  - Implementation is left to subclasses

Inheriting from Interfaces vs. Classes

- A class can *extend* 0 or 1 superclass
- An interface cannot extend a class
- A class or interface can *implement* 0 or more interfaces

Summary of Features of Actual Classes, Abstract Classes, and Interfaces

Property	Actual Class	Abstract Class	Interface
Instances (objects) of this can be created.	Yes	No	No
This can define instance variables and methods.	Yes	Yes	No
This can define constants.	Yes	Yes	Yes
The number of these a class can extend.	0 or 1	0 or 1	0
The number of these a class can implement.	0	0	Any number
This can extend another class.	Yes	Yes	No
This can declare abstract methods.	No	Yes	Yes
Variables of this type can be declared.	Yes	Yes	Yes

Class `Object` and Casting

Section 1.5

Class `Object`

- `Object` is the root of the class hierarchy
- Every *class* has `Object` as a superclass
- All classes inherit the methods of `Object` but may override them

Method	Behavior
`boolean equals(Object obj)`	Compares this object to its argument.
`int hashCode()`	Returns an integer hash code value for this object.
`String toString()`	Returns a string that textually represents the object.
`Class<?> getClass()`	Returns a unique object that identifies the class of this object.

Method `toString`

- You should always override `toString` method if you want to print object state
- If you do *not* override it:
  - `Object.toString` will return a `String`
  - Just not the `String` you want!

    Example: `ArrayBasedPD@ef08879`

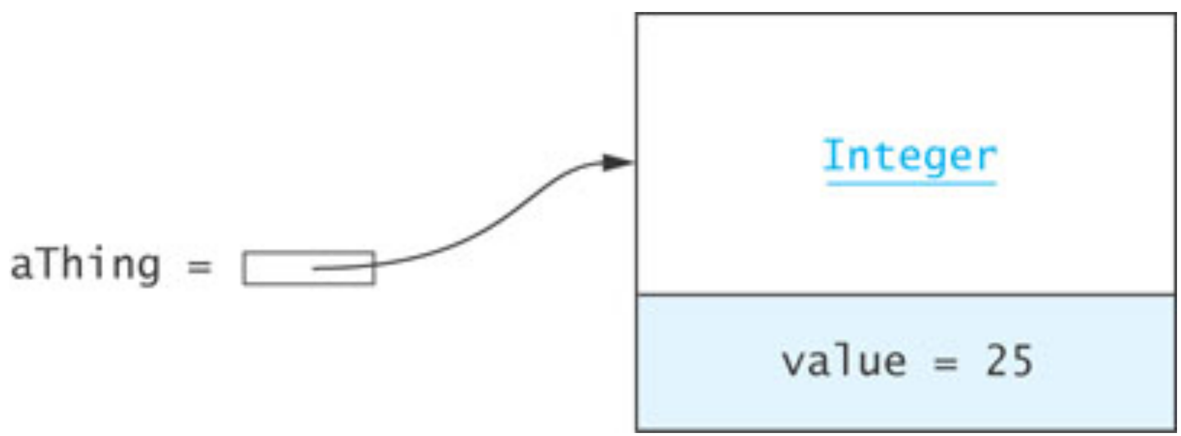
    The name of the class, @, instance's hash code

Operations Determined by Type of Reference Variable

- As shown previously with **`Computer`** and **`Notebook`**, a variable can refer to object whose type is a *subclass* of the variable's declared type
- Java is *strongly typed*

```
Object athing = Integer.valueOf(25);
```

  - The compiler always verifies that a variable's type includes the class of every expression assigned to the variable (*e.g., class* Object *must include class* Integer)



Operations Determined by Type of Reference Variable (cont.)

- The type of the *variable* determines what operations are legal

```
Object athing = Integer.valueOf(25);
```

- The following is legal:

```
athing.toString();
```

- But this is not legal:

```
athing.intValue();
```

- `Object` has a `toString()` method, but it does not have an `intValue()` method (even though **`Integer`** does, the reference is considered of type `Object`)

Operations Determined by Type of Reference Variable (cont.)

- The following method will compile,

```
athing.equals(Integer.valueOf("25"));
```

- `Object` has an **`equals`** method, and so does `Integer`
- Which one is called? Why?

- Why does the following generate a syntax error?

```
Integer aNum = aThing;
```

- Incompatible types!

Casting in a Class Hierarchy

- *Casting* obtains a reference of a different, but *matching,* type
- Casting *does not change* the object!
  - It creates an anonymous reference to the object

```
Integer aNum = (Integer) aThing;
```

- Does this work?

```
((Integer) aThing).intValue()
```

Casting in a Class Hierarchy (cont.)

- *Downcast:*
  - Cast *superclass* type to *subclass* type
  - Java checks *at run time* to make sure it's legal
  - If it's not legal, it throws **`ClassCastException`**
- *Upcast:*
  - Always valid but unnecessary

Using `instanceof` to Guard a Casting Operation

- **`instanceof`** can guard against a **`ClassCastException`**

```
Object obj = ...;
if (obj instanceof Integer) {
  Integer i = (Integer) obj;
  int val = i;
  ...;
} else {
  ...
}
```

Method `Object.equals`

- `Object.equals` method has a parameter of type `Object`

```
public boolean equals (Object other) {
... }
```

- Compares two objects to determine if they are equal
- A class must override `equals` in order to support comparison

Employee.equals()


```
/** Determines whether the current object matches its argument.
    @param obj The object to be compared to the current object
    @return true if the objects have the same name and address;
            otherwise, return false
*/
@Override
public boolean equals(Object obj) {
    if (obj == this) return true;
    if (obj == null) return false;
    if (this.getClass() == obj.getClass()) {
        Employee other = (Employee) obj;
        return name.equals(other.name) &&
               address.equals(other.address);
    } else {
        return false;
    }
}
```

`Class` **Class**

- Every class has a `Class` object that is created automatically when the class is loaded into an application
- Each `Class` object is unique for the class
- Method `getClass()` is a member of `Object` that returns a reference to this unique object
- In the previous example, if `this.getClass() == obj.getClass()` is true, then we know that `obj` and `this` are both of class `Employee`