

# CHAPTER 1

Object-Oriented Programming and Class Hierarchies

# Chapter Objectives

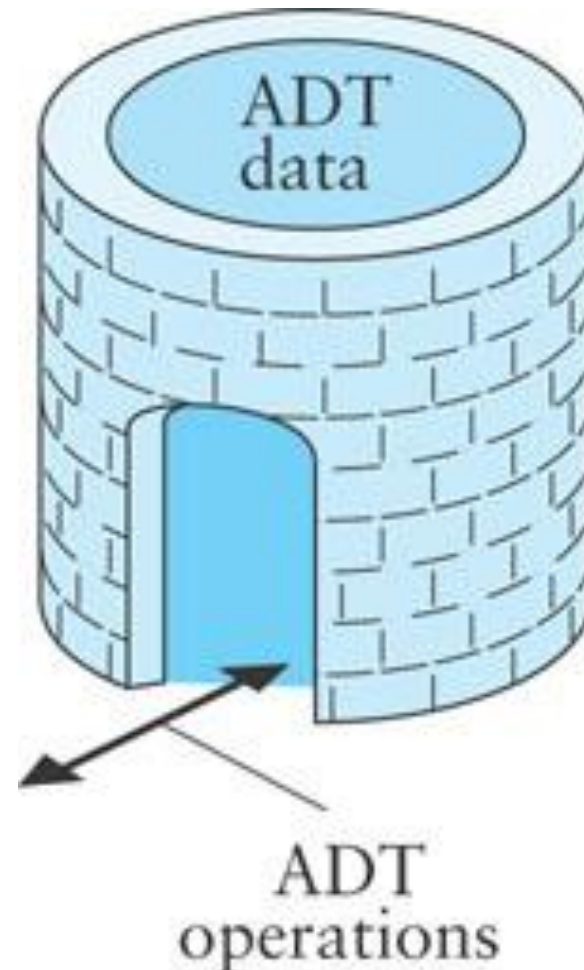
- ❑ Interfaces
- ❑ Inheritance and code reuse
- ❑ How Java determines which method to execute when there are multiple methods
- ❑ Abstract classes
- ❑ Abstract data types and interfaces
- ❑ `Object` class and overriding `Object` class methods
- ❑ `Exception` hierarchy
- ❑ Checked and unchecked exceptions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

# 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 implementation
  - ▣ methods not in the implementation
  - ▣ 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 {  
}
```

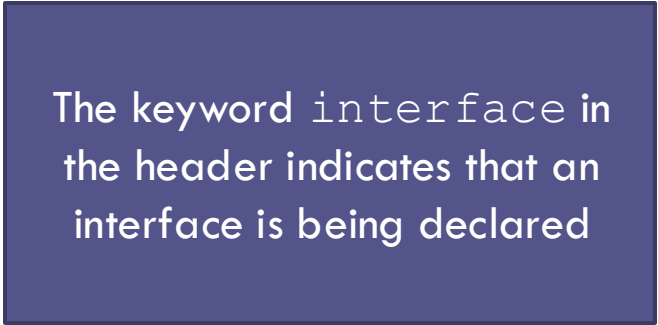
# Example: ATM Interface (cont.)

## 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 are 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

# Interface Definition (cont.)

## FORM:

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

## EXAMPLE:

```
public interface Payable {  
    double calcSalary();  
    boolean salaried();  
    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

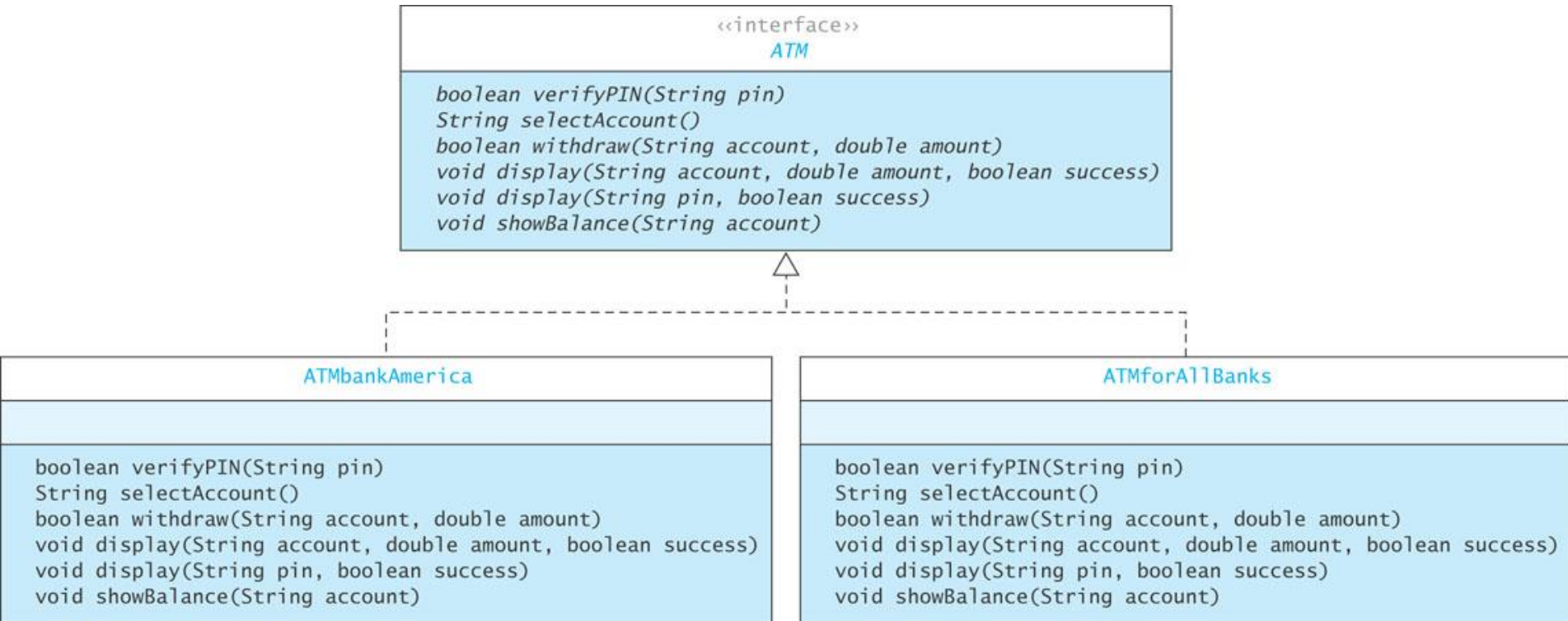
# The `implements` Clause

- 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 to 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

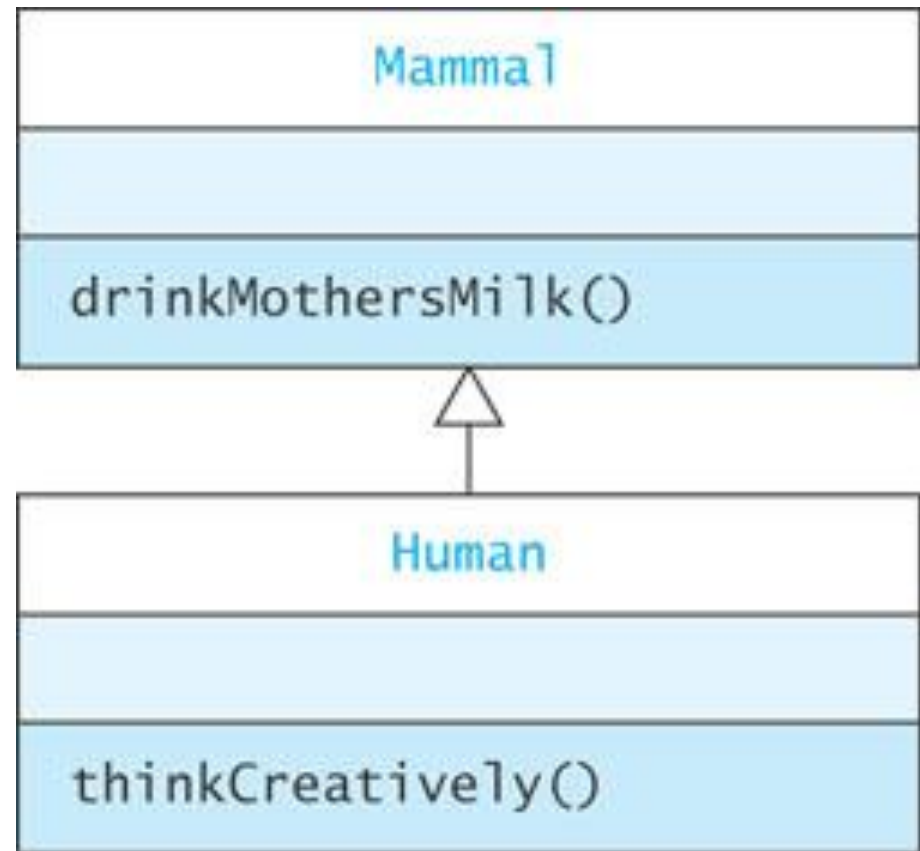
## Section 1.2

# Object-Oriented Programming

- Object-oriented programming (OOP) is popular because:
  - ▣ it enables *reuse* of previous code saved as *classes*
  - ▣ which saves times 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*

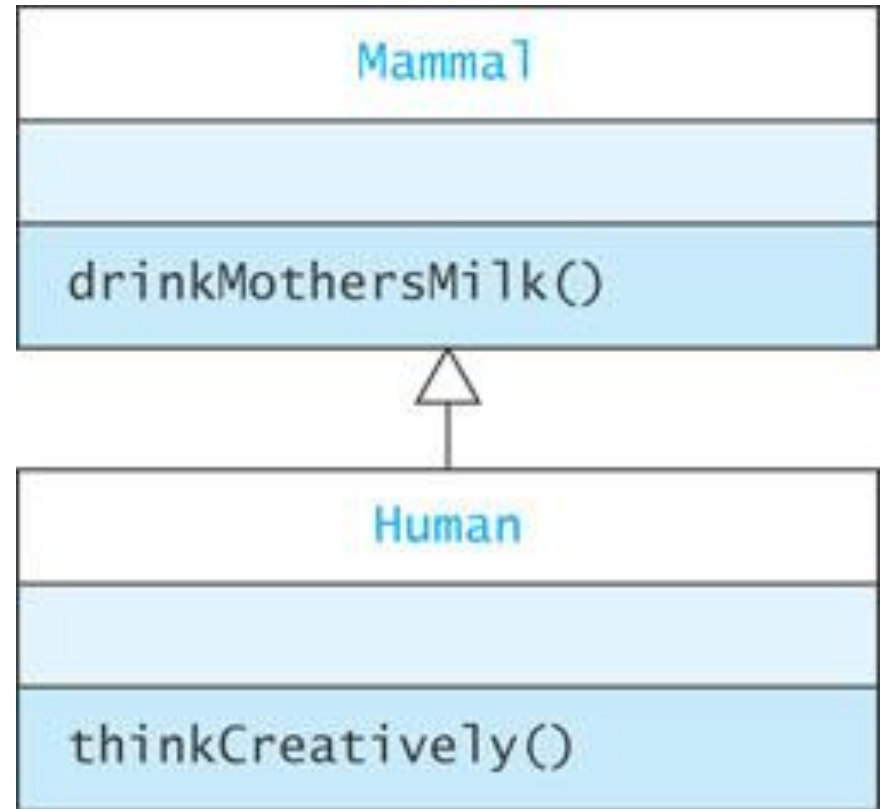
# Inheritance

- 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 a
  - ▣ manufacturer
  - ▣ processor
  - ▣ RAM
  - ▣ disk



# A Superclass and Subclass Example

## (cont.)

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

Computer
<pre>String manufacturer String processor int ramSize int diskSize double processorSpeed</pre>



# 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

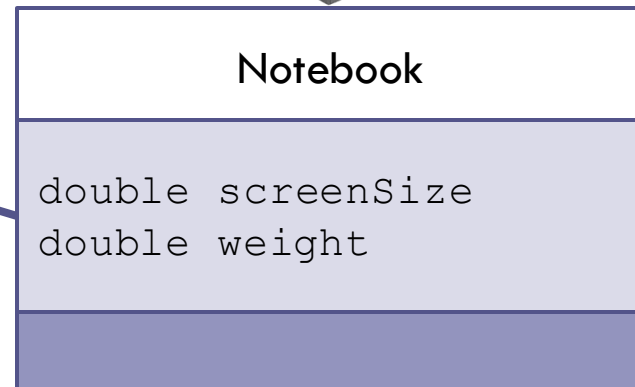
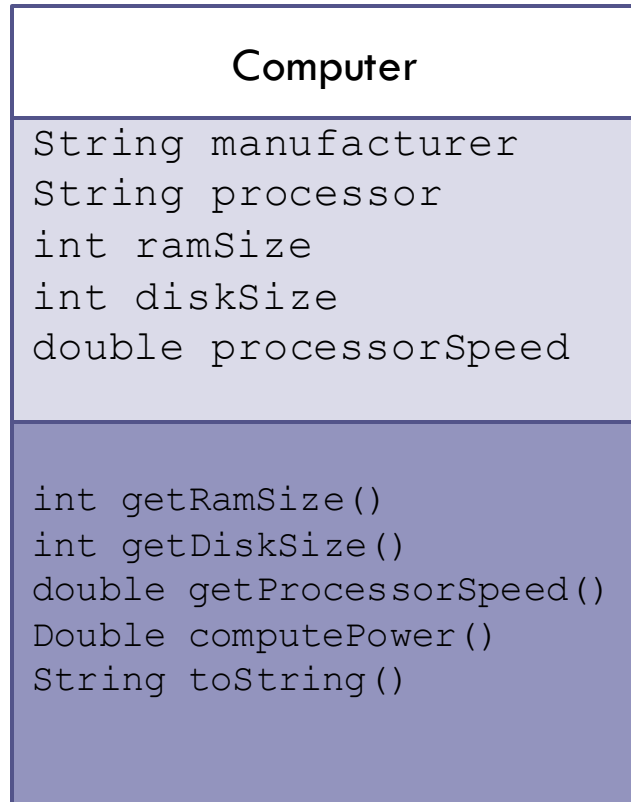
## (cont.)

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





# A Superclass and Subclass Example (cont.)



# 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 computers */
public class Computer {
    // Data fields
    private String manufacturer;
    private String processor;
    private double ramSize;
    private int diskSize;
    private double processorSpeed;

    // 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) {
        manufactuer = man;
        this.processor = processor;
        ramSize = ram;
        diskSize = disk;
        processorSpeed = procSpeed;
    }
}
```

# A Superclass and Subclass 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;

    // 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) {
        manufactuer = 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 {  
    // Date 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;
}
```

# 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;
}
```

**super ()**  
**super (argumentList)**

The `super ()` call in a class constructor invokes the superclass's constructor that has the corresponding *argumentList*.

The `super ()` call must be the first statement in a constructor.



# 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

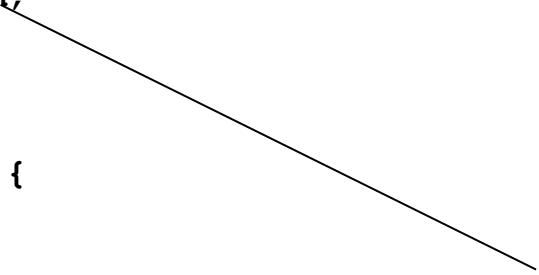
- ❑ Variables with *private visibility* cannot be accessed by a subclass
- ❑ Variables with *protected visibility* (defined by the keyword `protected`) are accessible by any subclass or any class in the same package
- ❑ In general, it is better to use private visibility and to restrict access to variables 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

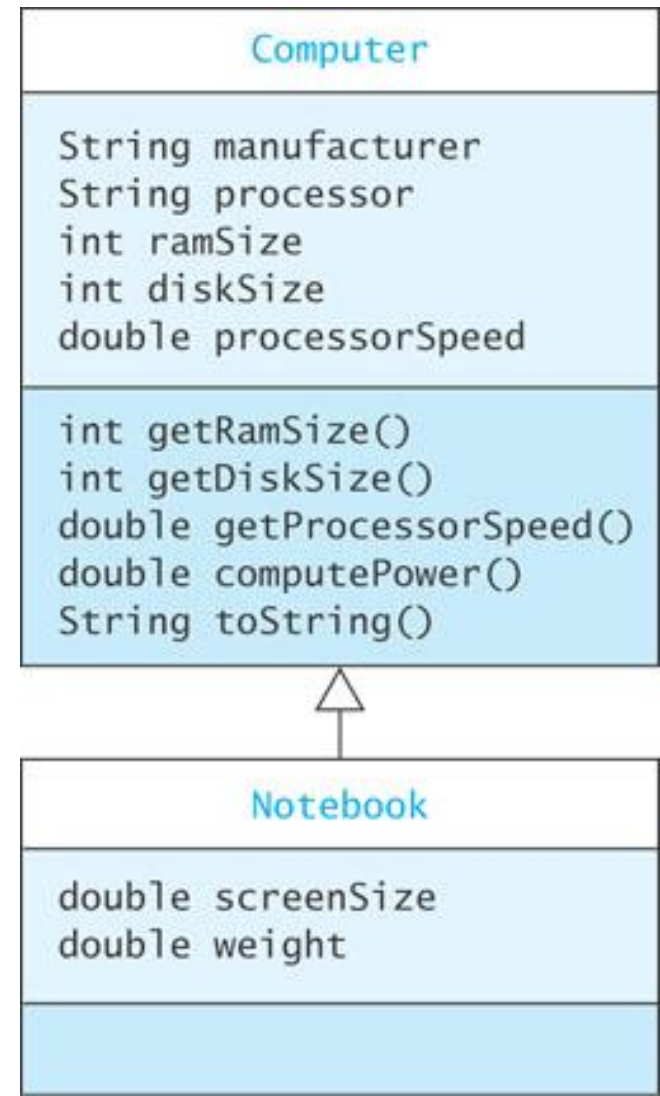
- Continuing the previous example, if we declare and then run:

```
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

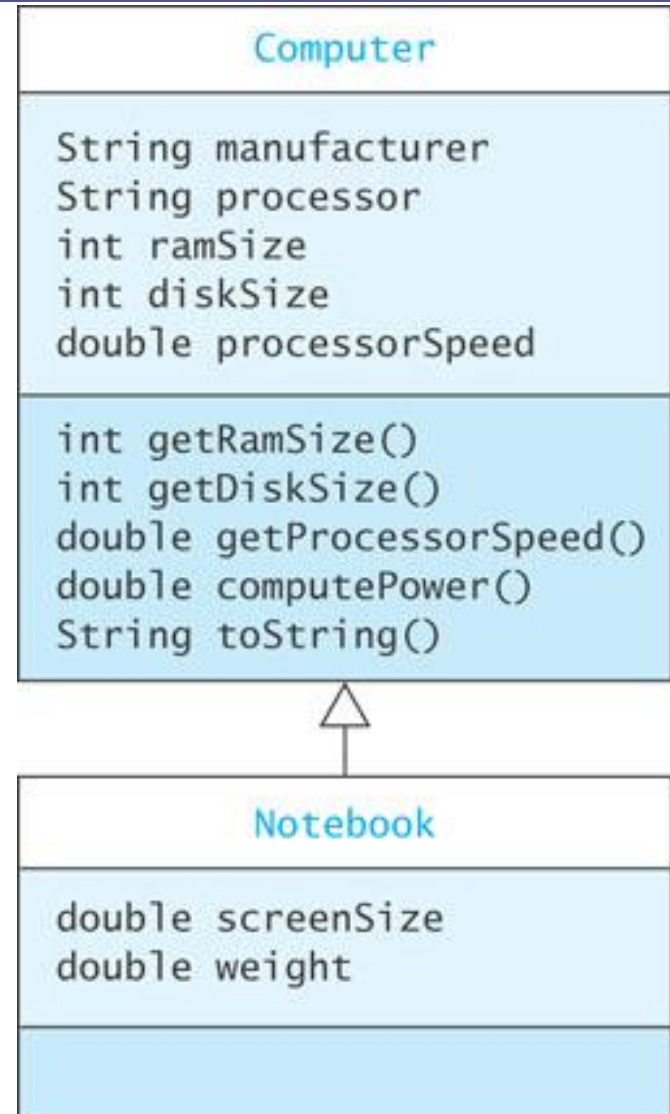
CPU: AMD

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



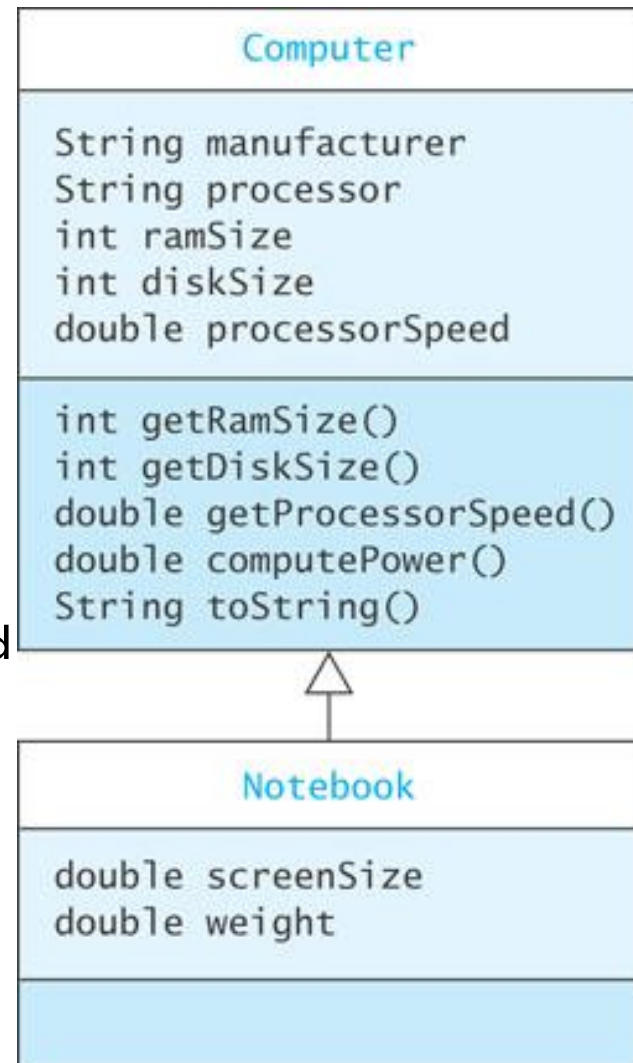
# Method Overriding (cont.)

- 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.)

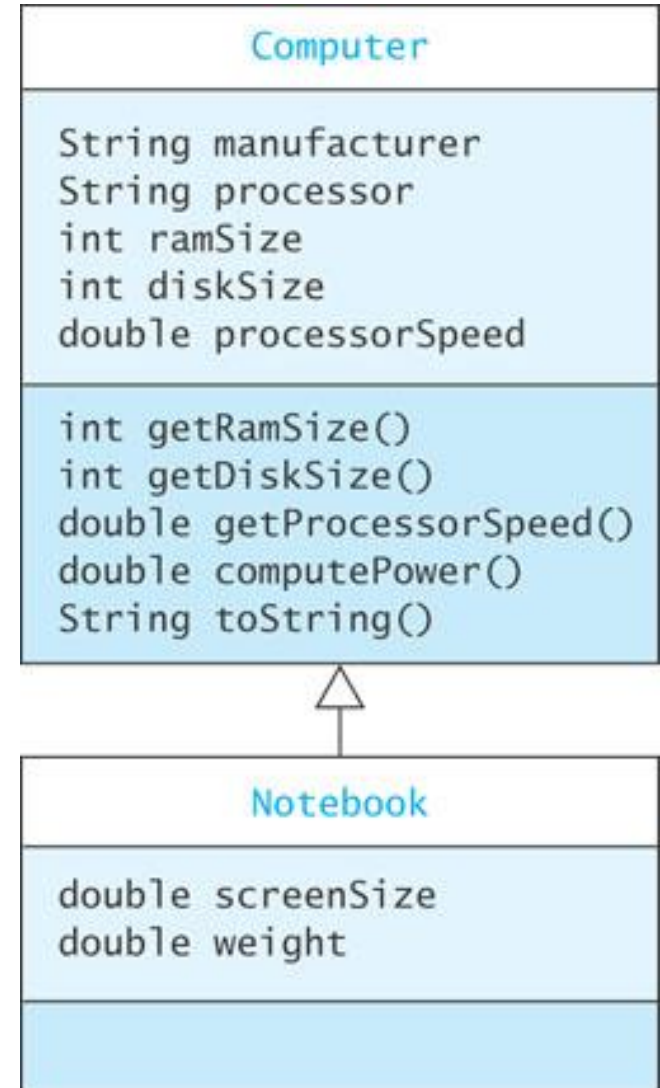
- To define a `toString()` for

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



# 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 (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 create a Notebook, we can use the constructor with six parameters

This call invokes the seven-parameter constructor for a Notebook, passing on the six parameters in this constructor, plus the default manufacturer constant DEFAULT\_NB\_MAN

```
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 compiler 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.)

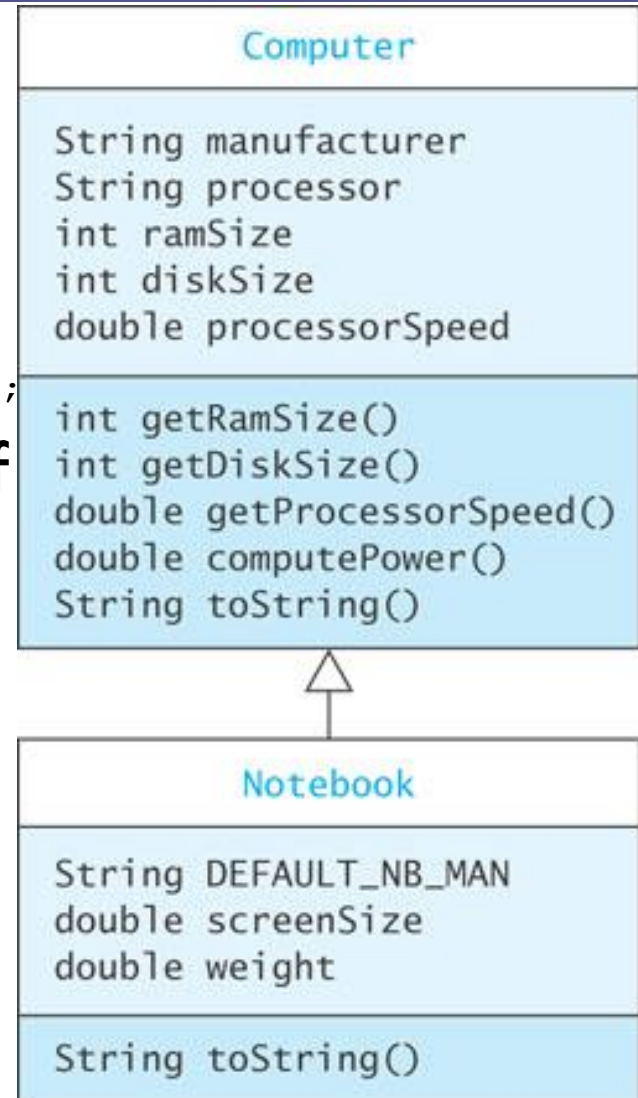
- For example, if you write a program to reference computers, you may want a variable to reference a `Computer` **or** a `Notebook`
- If you declare the reference variable as `Computer theComputer;` it can reference either a `Computer` **or** a `Notebook`—because a `Notebook` *is-a* `Computer`

# Polymorphism (cont.)

- Suppose the following statements are executed:

```
theComputer = new Notebook("Bravo",  
    "Intel", 4, 240, 2/4, 15.07.5);  
System.out.println(theComputer.toString());
```

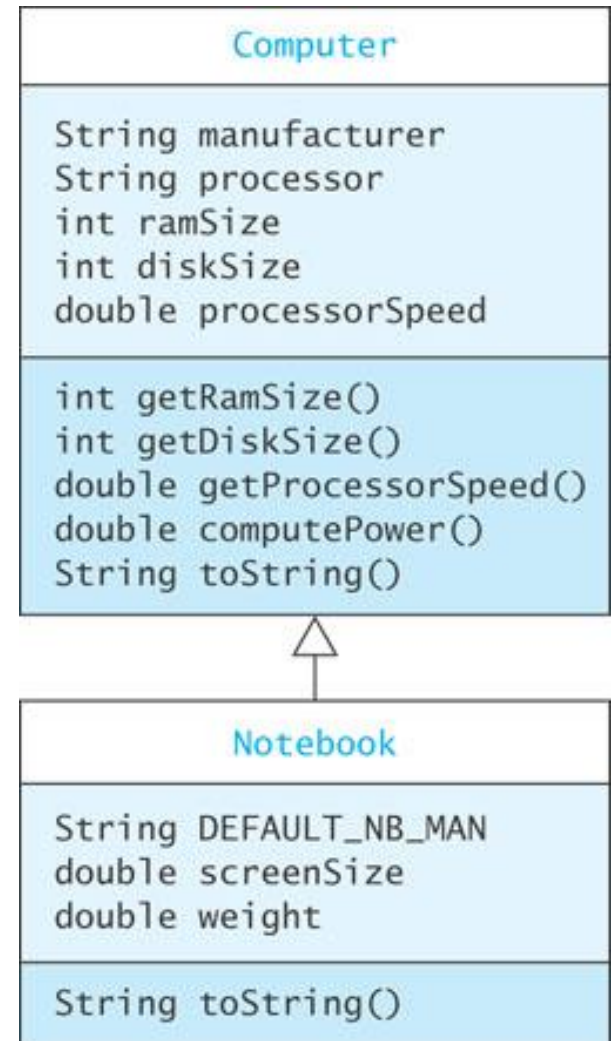
- The variable `theComputer` is of type `Computer`,
- Which `toString()` method will be called, `Computer`'s or `Notebook`'s?





# 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*



# 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);*

- An actual class that is a subclass of an abstract class must provide an implementation for each abstract method

# Abstract Classes (cont.)

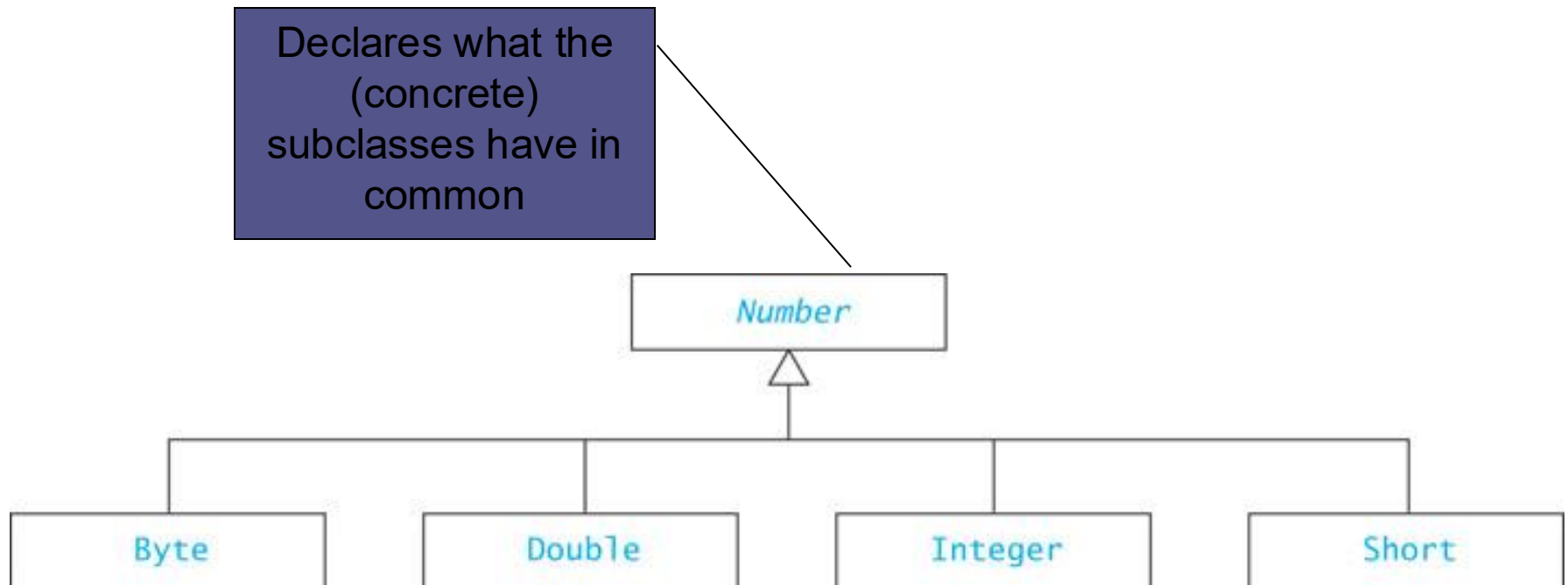
- Use an abstract class in a class hierarchy when you need a base class for two or more subclasses that share some attributes
- You can declare some or all of the attributes and define some or all of the methods that are common to these subclasses
- You can also require that the actual subclasses implement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





# 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`

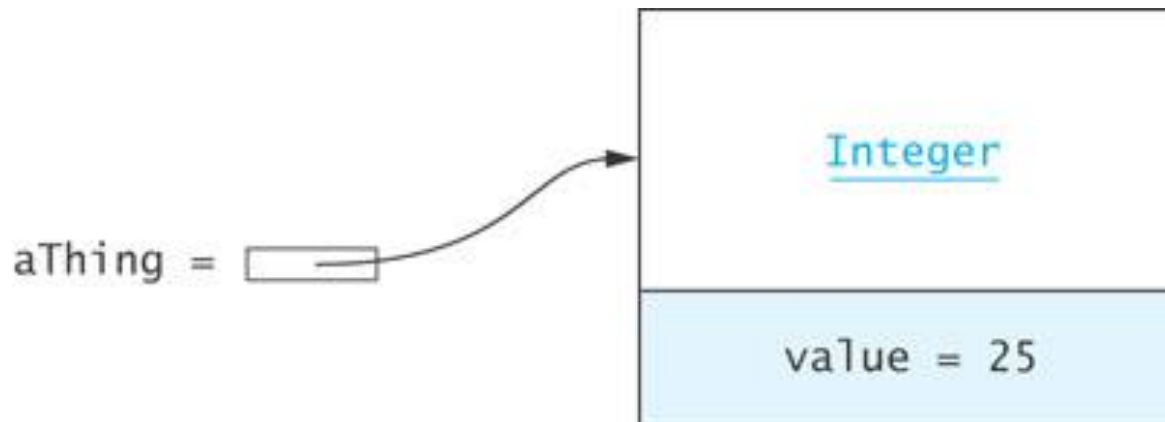
- You should always override `toString` method if you want to print object state
- If you do *not* override it:
  - ▣ `Object.toString` will return a `String`
  - ▣ Just not the `String` you want!

**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 = new Integer(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 = new Integer(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(new Integer("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;
```

- *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**

# 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 {
    ...
}
```

# Polymorphism Eliminates Nested if Statements

```
Number[] stuff = new Number[10];  
// each element of stuff must reference actual  
// object which is a subclass of Number  
...  
  
// Non OO style:  
if (stuff[i] instanceof Integer)  
    sum += ((Integer) stuff[i]).doubleValue();  
else if (stuff[i] instanceof Double)  
    sum += ((Double) stuff[i]).doubleValue();  
...  
  
// OO style:  
sum += stuff[i].doubleValue();
```

# Polymorphism Eliminates Nested `if` Statements (cont.)

- ❑ Polymorphic code style is more *extensible*; it works *automatically* with new subclasses
- ❑ Polymorphic code is more *efficient*; the system does one indirect branch versus many tests
- ❑ So ... uses of `instanceof` may suggest poor coding style

# 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`

# A Java Inheritance Example—The Exception Class Hierarchy

## Section 1.6

# Run-time Errors or Exceptions

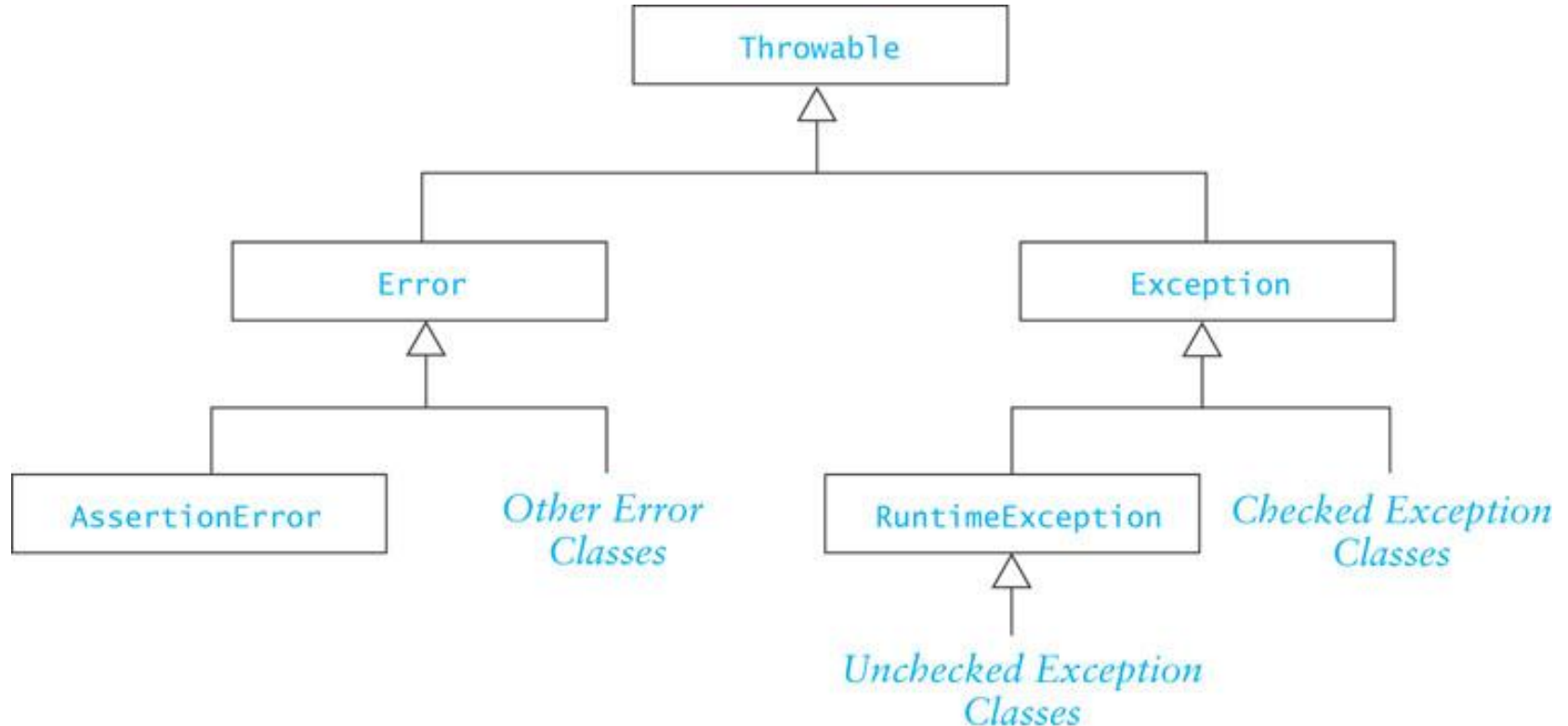
- Run-time errors
  - ▣ occur during program execution (i.e. at run-time)
  - ▣ occur when the JVM detects an operation that it knows to be incorrect
  - ▣ cause the JVM to throw an exception
- Examples of run-time errors include
  - ▣ division by zero
  - ▣ array index out of bounds
  - ▣ number format error
  - ▣ null pointer exception

# Run-time Errors or Exceptions (cont.)

Class	Cause/Consequence
ArithmeticException	An attempt to perform an integer division by zero.
ArrayIndexOutOfBoundsException	An attempt to access an array element using an index (subscript) less than zero or greater than or equal to the array's length.
NumberFormatException	An attempt to convert a string that is not numeric to a number.
NullPointerException	An attempt to use a null reference value to access an object.
NoSuchElementException	An attempt to get a next element after all elements were accessed.
InputMismatchException	The token returned by a Scanner <code>next . . .</code> method does not match the pattern for the expected data type.

# Class Throwable

- Throwable is the superclass of all exceptions
- All exception classes inherit its methods



# Class Throwable (cont.)

- Throwable has several useful methods that are available to all of its subclasses

Method	Behavior
String getMessage()	Returns the detail message.
void printStackTrace()	Prints the stack trace to System.err.
String toString()	Returns the name of the exception followed by the detail message.

# Checked and Unchecked Exceptions

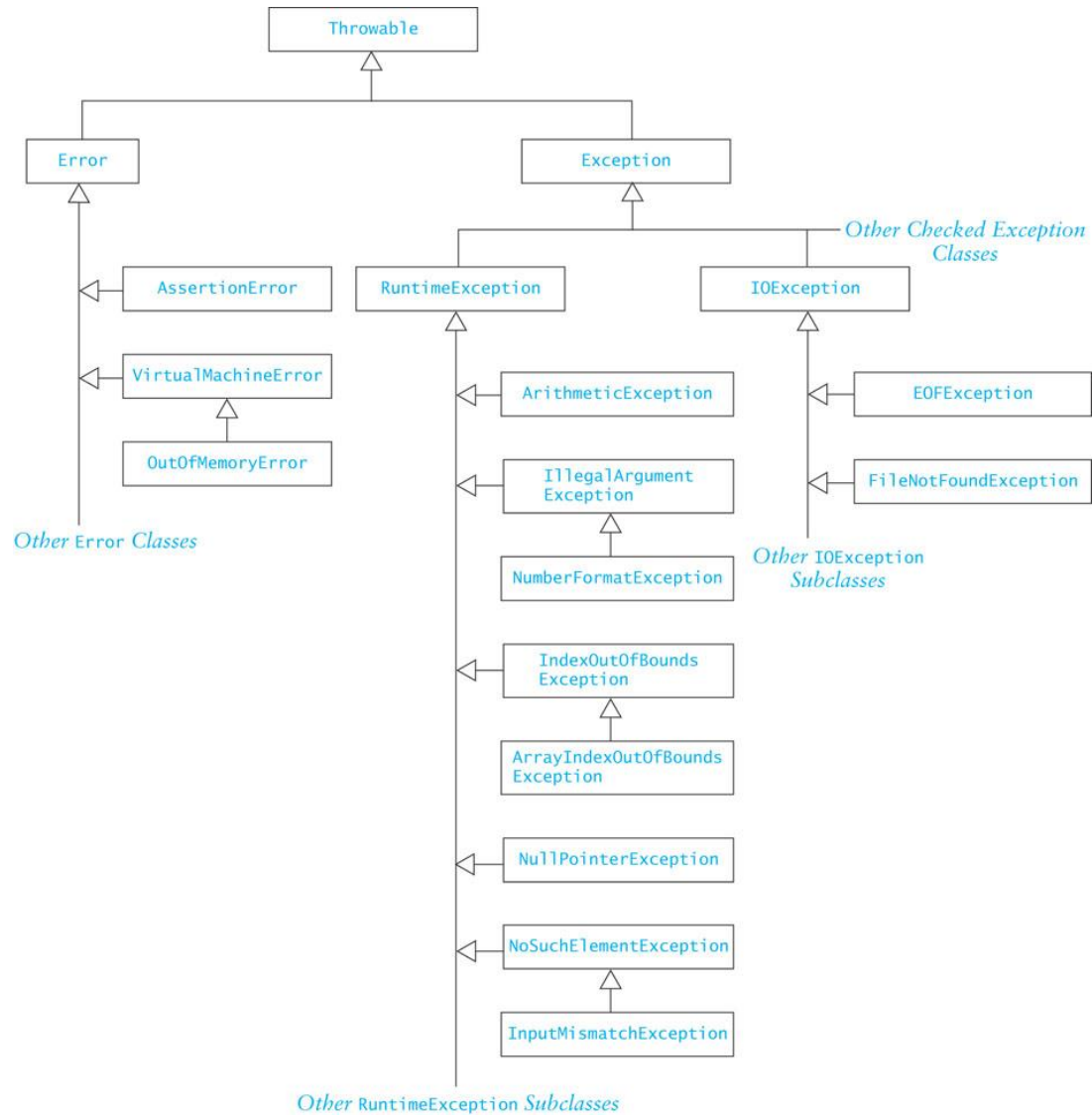
- *Checked* exceptions
  - ▣ normally not due to programmer error
  - ▣ generally beyond the control of the programmer
  - ▣ all input/output errors are checked exceptions
  - ▣ **Examples:** `IOException`, `FileNotFoundException`
- *Unchecked* exceptions result from
  - ▣ programmer error (try to prevent them with defensive programming).
  - ▣ a serious external condition that is unrecoverable
  - ▣ **Examples:** `NullPointerException`,  
`ArrayIndexOutOfBoundsException`

# Checked and Unchecked Exceptions (cont.)

- The class `Error` and its subclasses represent errors due to serious external conditions; they are unchecked
  - ▣ Example: `OutOfMemoryError`
  - ▣ You cannot foresee or guard against them
  - ▣ While you can attempt to handle them, it is generally not a good idea as you will probably be unsuccessful
  
- The class `Exception` and its subclasses can be handled by a program
  - ▣ `RuntimeException` and its subclasses are unchecked
  - ▣ *All others must be either:*
    - explicitly *caught* or
    - explicitly *mentioned as thrown* by the method



# Checked and Unchecked Exceptions (cont.)



# Some Common Unchecked Exceptions

- ❑ `ArithmeticException`: **division by zero, etc.**
- ❑ `ArrayIndexOutOfBoundsException`
- ❑ `NumberFormatException`: **converting a “bad” string to a number**
- ❑ `NullPointerException`
- ❑ `NoSuchElementException`: **no more tokens available**

# Handling Exceptions

- When an exception is thrown, the normal sequence of execution is interrupted
- Default behavior (no handler)
  - ▣ Program stops
  - ▣ JVM displays an error message
- The programmer may provide a *handle*
  - ▣ Enclose statements in a `try` block
  - ▣ Process the exception in a `catch` block

# The try-catch Sequence

- The try-catch sequence resembles an if-then-else statement

```
try {  
    // Execute the following statements until an  
    // exception is thrown  
    ...  
    // Skip the catch blocks if no exceptions were thrown  
  
} catch (ExceptionTypeA ex) {  
    // Execute this catch block if an exception of type  
    // ExceptionTypeA was thrown in the try block  
    ...  
  
} catch (ExceptionTypeB ex) {  
    // Execute this catch block if an exception of type  
    // ExceptionTypeB was thrown in the try block  
    ...  
  
}
```

# The try-catch Sequence (cont.)

- The try-catch sequence resembles an if

```
try {  
    // Execute the following statements until  
    // exception is thrown  
    ...  
    // Skip the catch blocks if no exception  
} catch (ExceptionTypeA ex) {  
    // Execute this catch block if an exception  
    // ExceptionTypeA was thrown in the try  
    ...  
} catch (ExceptionTypeB ex) {  
    // Execute this catch block if an exception  
    // ExceptionTypeB was thrown in the try block  
    ...  
}
```

## PITFALL!

### Unreachable catch block

`ExceptionTypeB` cannot be a subclass of `ExceptionTypeA`. If it was, its exceptions would be caught by the first catch clause and its catch clause would be unreachable.

# Using try-catch

- User input is a common source of exceptions

```
public static int getIntValue(Scanner scan) {
    int nextInt = 0;           // next int value
    boolean validInt = false; // flag for valid input
    while(!validInt) {
        try {
            System.out.println("Enter number of kids: ");
            nextInt = scan.nextInt();
            validInt = true;
        } catch (InputMismatchException ex) {
            scan.nextLine(); // clear buffer
            System.out.println("Bad data-enter an integer");
        }
    }
    return nextInt;
}
```

# Throwing an Exception When Recovery is Not Obvious

- In some cases, you may be able to write code that detects certain types of errors, but there may not be an obvious way to recover from them
- In these cases an the exception can be *thrown*
- The calling method receives the thrown exception and must handle it

# Throwing an Exception When Recovery is Not Obvious (cont.)

```
public static void processPositiveInteger(int n) {  
    if (n < 0) {  
        throw new IllegalArgumentException(  
            "Invalid negative argument");  
    } else {  
        // Process n as required  
        ...  
    }  
}
```



# Throwing an Exception When Recovery is Not Obvious (cont.)

```
public static void main(String[] args) {  
    Scanner scan = new Scanner(System.in);  
    try {  
        int num = getIntValue(scan);  
        processPositiveInteger(num);  
    } catch (IllegalArgumentException ex) {  
        System.err.println(ex.getMessage());  
        System.exit(1); // error indication  
    }  
    System.exit(0); // normal exit  
}
```

# Packages and Visibility

## Section 1.7

# Packages

- A Java *package* is a group of *cooperating classes*
- The Java API is organized as packages
- Indicate the package of a class at the top of the file:  
**`package classPackage;`**
- Classes in the *same package* should be in the *same directory* (folder)
- The folder must have the same name as the package
- Classes in the *same folder* must be  
in the *same package*

# Packages and Visibility

- Classes *not* part of a package can only access `public` members of classes in the package
- If a class is not part of the package, it must access the public classes by their complete name, which would be `packageName.className`
- For example,

```
x = Java.awt.Color.GREEN;
```

- If the package is imported, the `packageName` prefix is not required.

```
import java.awt.Color;
```

```
...
```

```
x = Color.GREEN;
```

# The Default Package

- ❑ Files which do not specify a package are part of the default package
- ❑ If you do not declare packages, all of your classes belong to the default package
- ❑ The default package is intended for use during the early stages of implementation or for small prototypes
- ❑ When you develop an application, declare its classes to be in the same package

# Visibility

- You know about three visibility layers, `public`, `protected`, `private`
- A fourth layer, *package visibility*, lies between `private` and `protected`
- Classes, data fields, and methods with package visibility are accessible to all other methods of the same package, but are not accessible to methods outside the package
- Classes, data fields, and methods that are declared `protected` are visible within subclasses that are declared *outside* the package (in addition to being visible to all members *inside* the package)
- There is no keyword to indicate package visibility
- Package visibility is the default in a package if `public`, `protected`, `private` are not used

# Visibility Supports Encapsulation

- Visibility rules enforce encapsulation in Java
- `private`: for members that should be invisible even in subclasses
- `package`: shields classes and members from classes outside the package
- `protected`: provides visibility to extenders of classes in the package
- `public`: provides visibility to all

# Visibility Supports Encapsulation

## (cont.)

Visibility	Applied to Classes	Applied to Class Members
<b>private</b>	Applicable to inner classes. Accessible only to members of the class in which it is declared.	Visible only within this class.
Default or package	Visible to classes in this package.	Visible to classes in this package.
<b>protected</b>	Applicable to inner classes. Visible to classes in this package and to classes outside the package that extend the class in which it is declared.	Visible to classes in this package and to classes outside the package that extend this class.
<b>public</b>	Visible to all classes.	Visible to all classes. The class defining the member must also be public.



# Visibility Supports Encapsulation

## (cont.)

---

- ❑ Encapsulation insulates against change
- ❑ Greater visibility means less encapsulation
- ❑ So... use the most restrictive visibility possible to get the job done!

# A Shape Class Hierarchy

1.8

# Case Study: Processing Geometric Figures

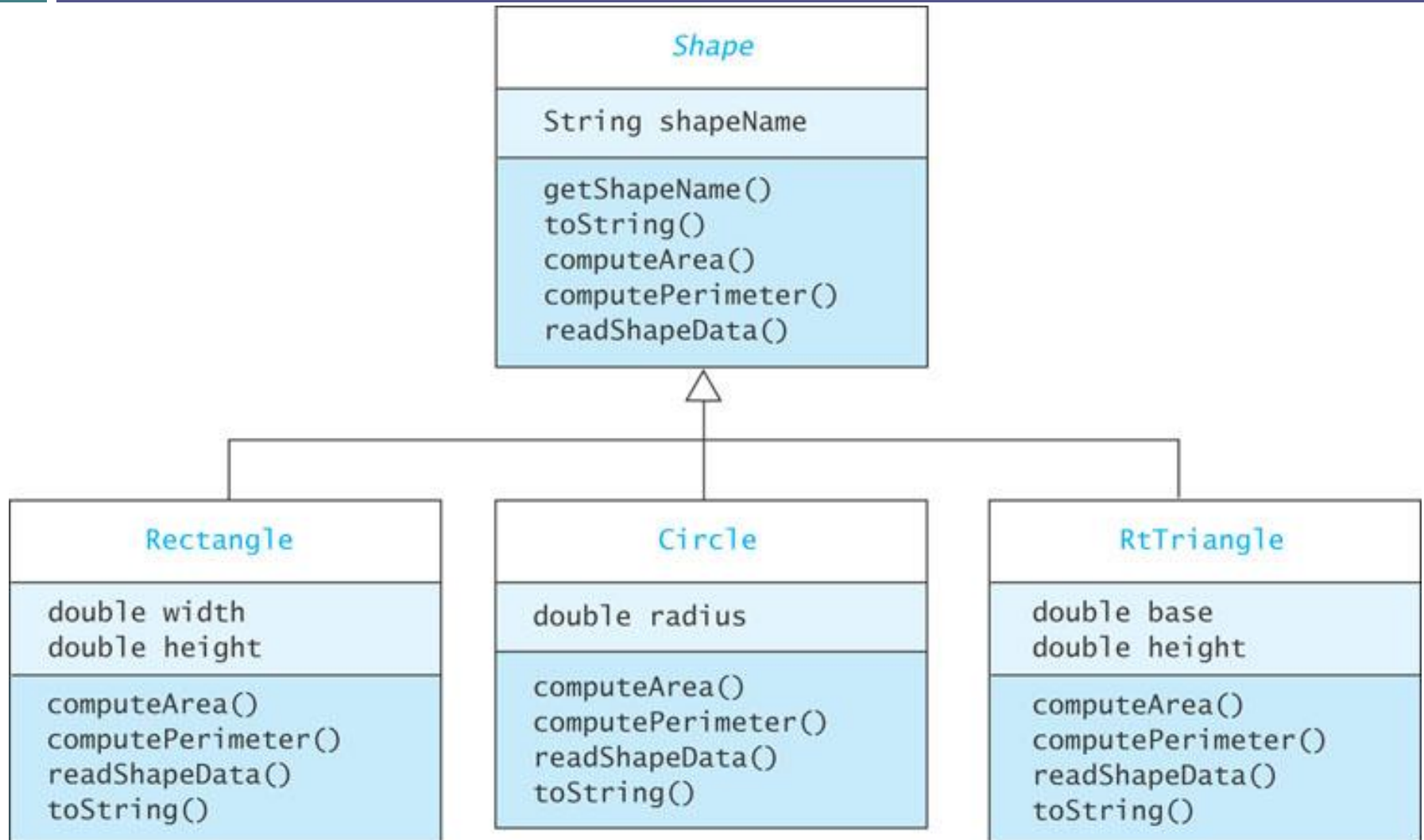
## Problem

We want to process some standard geometric shapes. Each figure object will be one of three standard shapes (rectangle, circle, right triangle). We want to do standard computations, such as finding the area and perimeter, for any of these shapes.

## Analysis

For each geometric shape, we need a class that represents the shape and knows how to perform the standard computations on it. These classes will be `Rectangle`, `Circle`, and `RtTriangle`. To ensure that these shape classes all define the required computational methods (finding area and perimeter), we will make them abstract methods in the base class (`Shape`) for the shape hierarchy. If a shape class does not have the required methods, we will get a syntax error when we attempt to compile it.

# Shape Class Hierarchy



# Class Rectangle

Data Field	Attribute
double width	Width of a rectangle
double height	Height of a rectangle
Method	Behavior
double computeArea()	Computes the rectangle area ( $\text{width} \times \text{height}$ )
double computePerimeter()	Computes the rectangle perimeter ( $2 \times \text{width} + 2 \times \text{height}$ )
void readShapeData()	Reads the width and height
String toString()	Returns a string representing the state