Day 03

BASIC CONCEPTS OF JAVA 3

FUNDAMENTALS OF TELECOMMUNICATIONS LAB

Dr. Huy Nguyen

AGENDA

- Arrays & Array Lists
- Interfaces & Polymorphism
- Inheritance
- Input / Output & Exception Handling

AGENDA

- Arrays & Array Lists
- Interfaces & Polymorphism
- Inheritance
- Input / Output & Exception Handling

BASIC CONCEPTS OF JAVA 3

ARRAYS & ARRAY LISTS

Chapter Goals

- To become familiar with using arrays and array lists
- To learn about wrapper classes, auto-boxing and the generalized for loop
- To study common array algorithms
- To learn how to use two-dimensional arrays
- To understand when to choose array lists and arrays in your programs
- To implement partially filled arrays
- To understand the concept of regression testing

Arrays

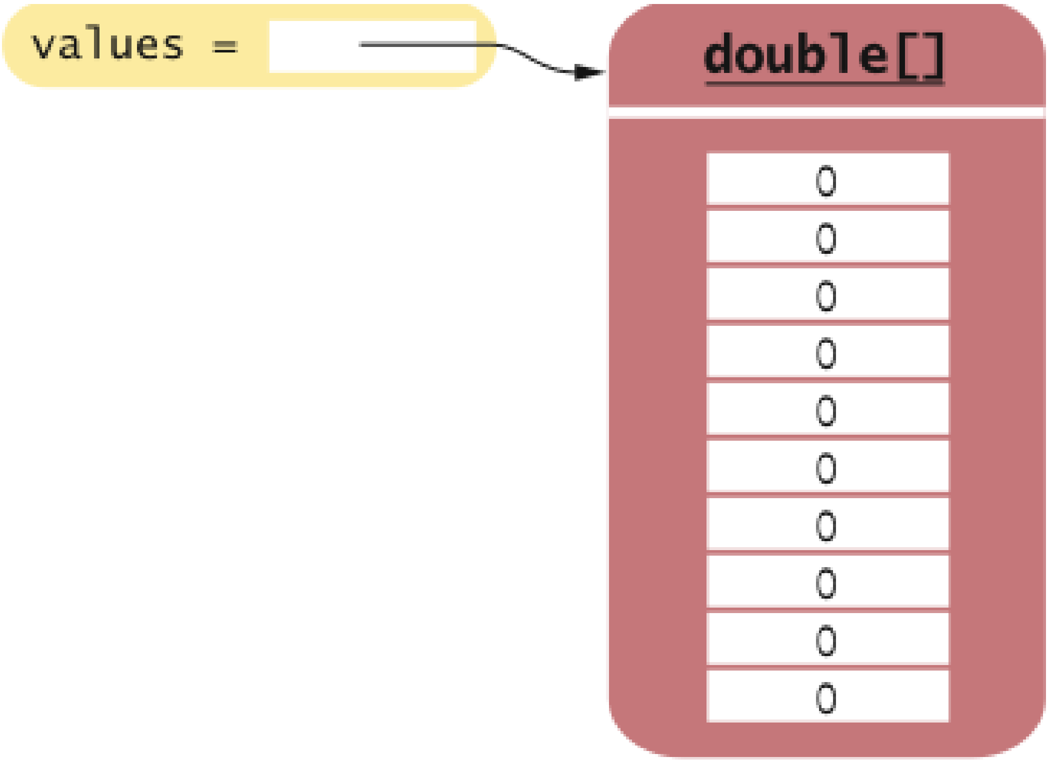
- Array: Sequence of values of the same type
- Construct array:

  `new double[10]`
- Store in variable of type `double[]`:

  `double[] data = new double[10];`
- When array is created, all values are initialized depending on array type:
  - *Numbers:* `0`
  - *Boolean :* `false`
  - *Object References:* `null`

Arrays

- Array reference:

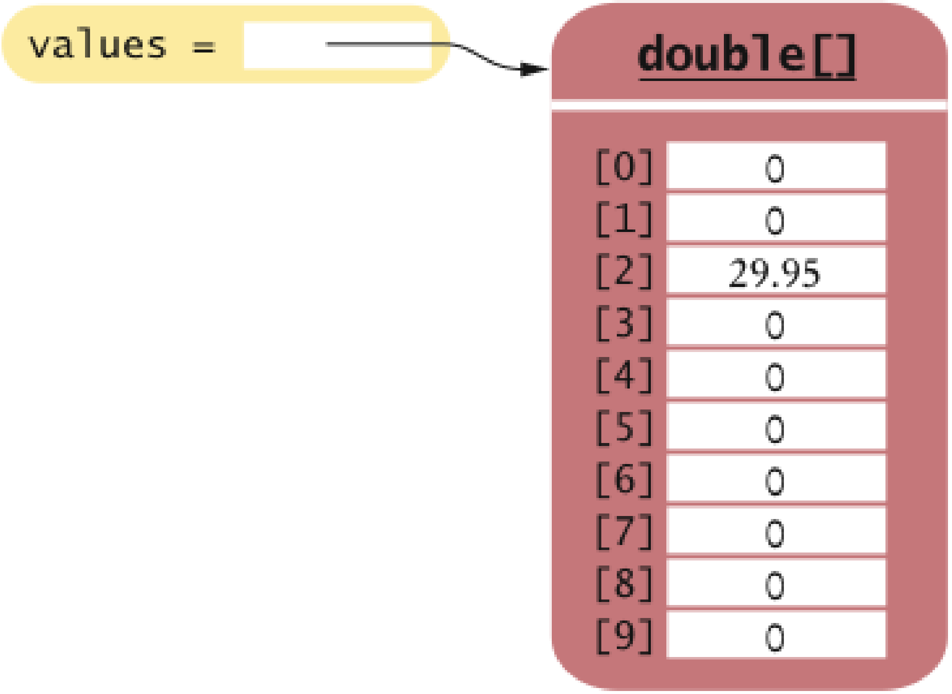


Arrays

- Use `[]` to access an element:

```
values[2] = 29.95;
```

- Modifying an array element



Arrays

- Using the value stored:

```
System.out.println("The value of this data item is "
   + values[2]);
```

- Get array length as `values.length` (Not a method!)
- Index values range from `0` to `length - 1`
- Accessing a nonexistent element results in a **bounds error**:

```
double[] values = new double[10];
values[10] = 29.95; // ERROR
```

- Limitation: Arrays have fixed length

Declaring Arrays

`int[] numbers = new int[10];`	An array of ten integers. All elements are initialized with zero.
`final int NUMBERS_LENGTH = 10;` `int[] numbers = new int[NUMBERS_LENGTH];`	It is a good idea to use a named constant instead of a "magic number".
`int valuesLength = in.nextInt();` `double[] values = new double[valuesLength];`	The length need not be a constant.
`int[] squares = { 0, 1, 4, 9, 16 };`	An array of five integers, with initial values.
`String[] names = new String[3];`	An array of three string references, all initially `null`.
`String[] friends = { "Emily", "Bob", "Cindy" };`	Another array of three strings.
`double[] values = new int[10]`	**Error:** You cannot initialize a `double[]` variable with an array of type `int[]`.

Syntax *Arrays*

Syntax To construct an array: new *typeName*[*length*]

To access an element: *arrayReference*[*index*]

Example

Name of array variable

Type of array variable

Element type

Length

```
double[] values = new double[10];
```

Initialized with zero

```
double[] moreValues = { 32, 54, 67.5, 29, 35 };
```

Initialized with these elements

Use brackets to access an element.

```
values[i] = 29.95;
```

The index must be ≥ 0 and < the length of the array.

Self Check

What elements does the data array contain after the following statements?

```
double[] values = new double[10];
for (int i = 0; i < values.length; i++)
   values[i] = i * i;
```

Answer: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, but not 100

Self Check

What do the following program segments print? Or, if there is an error, describe the error and specify whether it is detected at compile-time or at run-time.

```
a) double[] a = new double[10];
   System.out.println(a[0]);
b) double[] b = new double[10];
   System.out.println(b[10]);
c) double[] c;
   System.out.println(c[0]);
```

Answer:

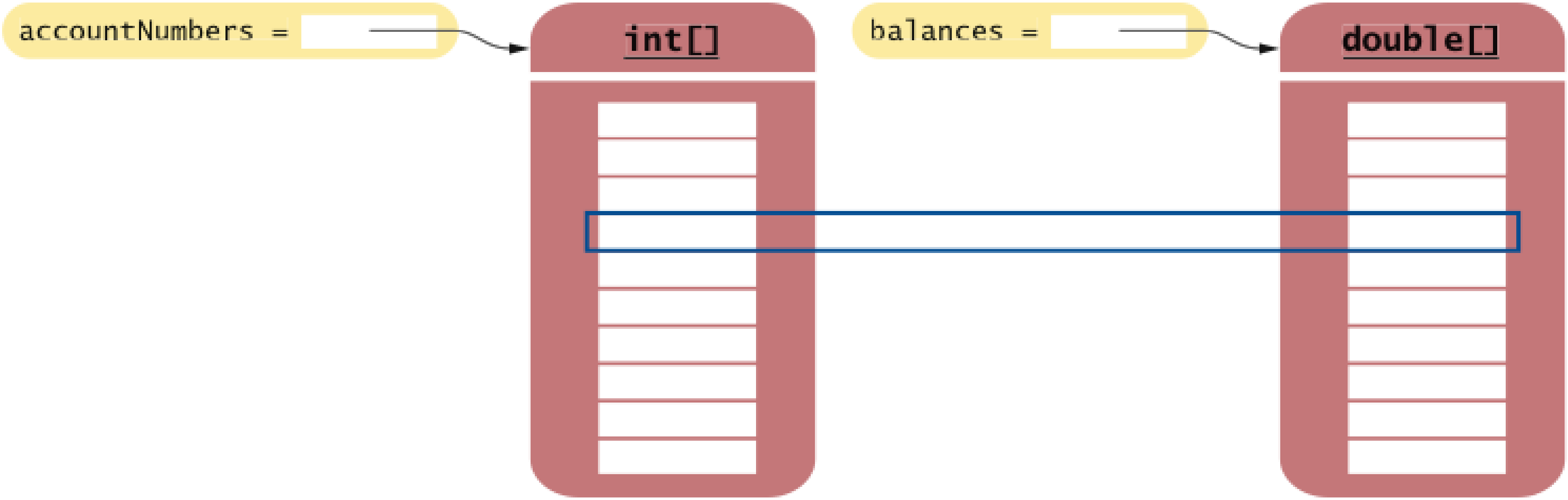
a) 0

b) a run-time error: array index out of bounds

c) a compile-time error: c is not initialized

Make Parallel Arrays into Arrays of Objects

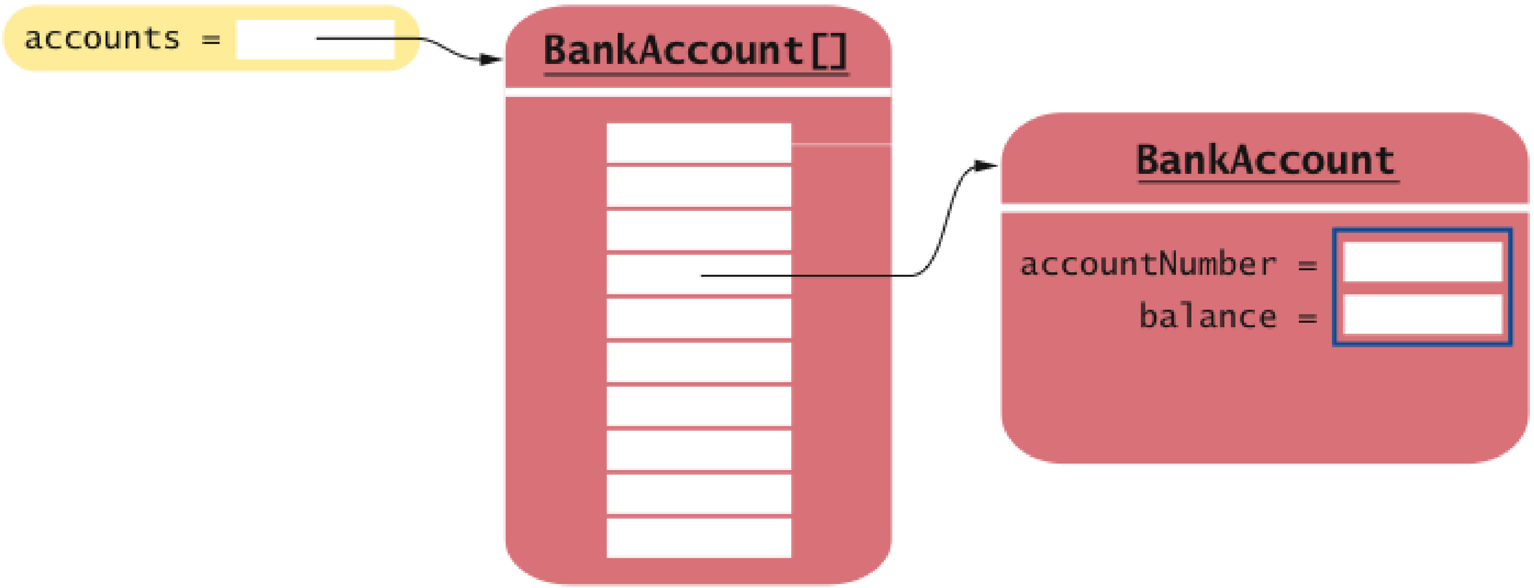
```
// Don't do this
int[] accountNumbers;
double[] balances;
```



Make Parallel Arrays into Arrays of Objects

Avoid parallel arrays by changing them into arrays of objects:

```
BankAccount[] accounts;
```



Array Lists

- `ArrayList` class manages a sequence of objects
- Can grow and shrink as needed
- `ArrayList` class supplies methods for many common tasks, such as inserting and removing elements
- `ArrayList` is a **generic class**:

  ```
  ArrayList<T>
  ```

  collects objects of **type parameter** `T`:

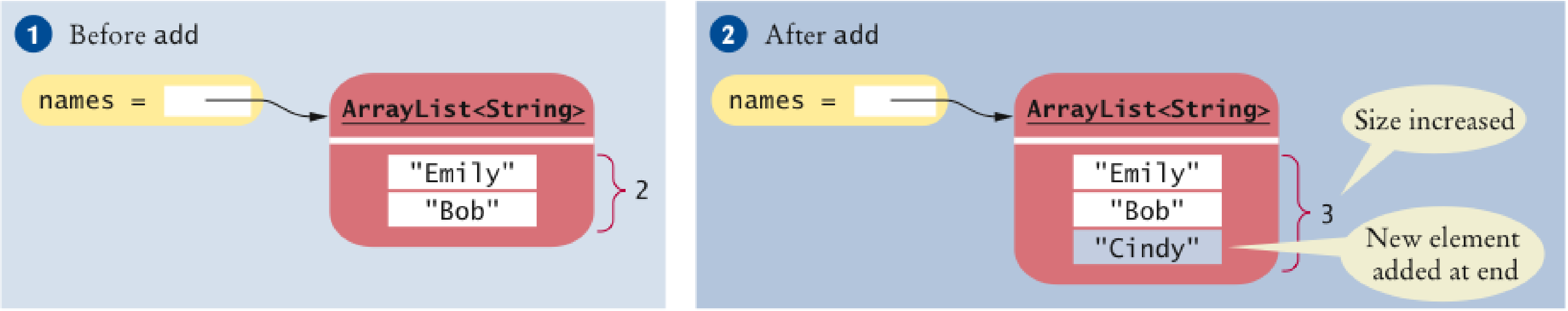
  ```
  ArrayList<String> names = new ArrayList<String>();
  names.add("Emily");
  names.add("Bob");
  names.add("Cindy");
  ```

- `size` method yields number of elements

Adding Elements

- To add an object to the end of the array list, use the add method:

```
names.add("Emily");
names.add("Bob");
names.add("Cindy");
```



Retrieving Array List Elements

- To obtain the value an element at an index, use the `get` method
- Index starts at 0

```
String name = names.get(2);
// gets the third element of the array list
```

- Bounds error if index is out of range
- Most common bounds error:

```
int i = names.size();
name = names.get(i); // Error
// legal index values are 0 ... i-1
```

Setting & Removing Elements

- To set an element to a new value, use the `set` method:

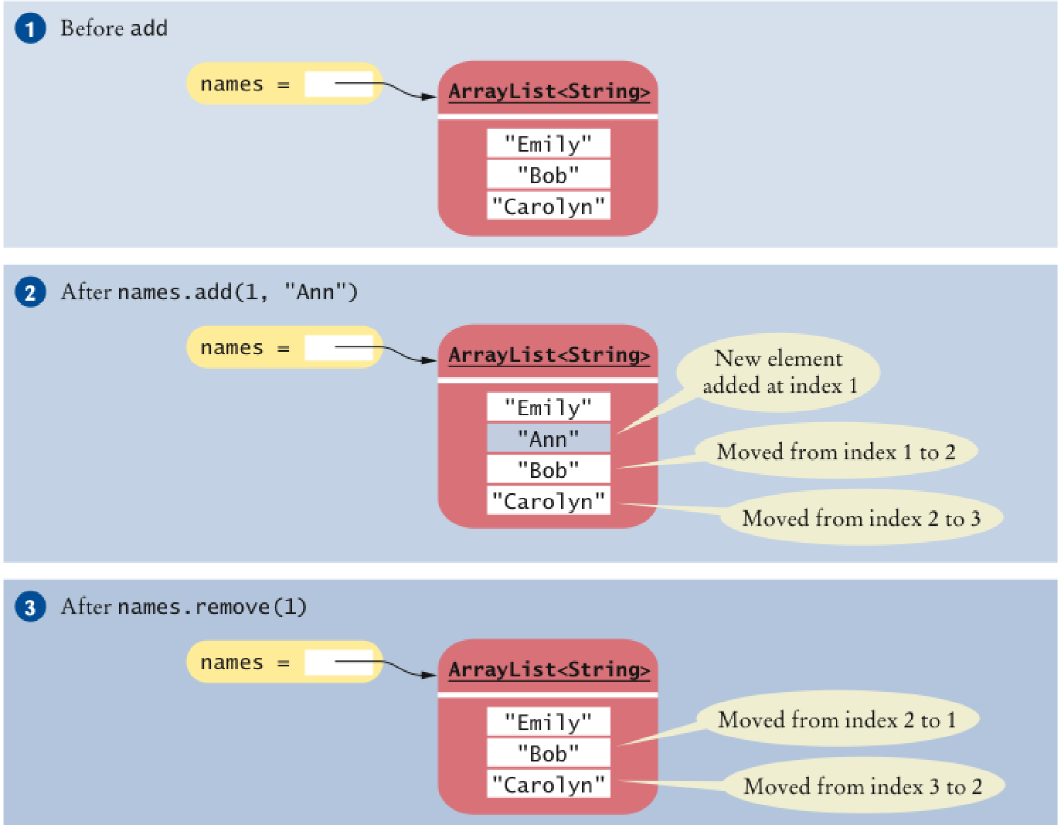
```
names.set(2, "Carolyn");
```

- To remove an element at an index, use the `remove` method:

```
names.remove(1);
```

Adding and Removing Elements

```
names.add("Emily");
names.add("Bob");
names.add("Cindy");
names.set(2, "Carolyn");
names.add(1, "Ann");
names.remove(1);
```



Working with Array Lists

ArrayList<String> names = new ArrayList<String>();	Constructs an empty array list that can hold strings.
names.add("Ann"); names.add("Cindy");	Adds elements to the end.
System.out.println(names);	Prints [Ann, Cindy].
names.add(1, "Bob");	Inserts an element at index 1. names is now [Ann, Bob, Cindy].
names.remove(0);	Removes the element at index 0. names is now [Bob, Cindy].
names.set(0, "Bill");	Replaces an element with a different value. names is now [Bill, Cindy].
String name = names.get(i);	Gets an element.
String last = names.get(names.size() - 1);	Gets the last element.

ArrayList<Integer> squares =
   new ArrayList<Integer>();
for (int i = 0; i < 10; i++)
{
   squares.add(i * i);
}
	Constructs an array list holding the first ten squares.

Syntax *Array Lists*

Syntax

To construct an array list: `new ArrayList<typeName>()`

To access an element:

```
arraylistReference.get(index)
arraylistReference.set(index, value)
```

Example

Variable type

Variable name

An array list object of size 0

```
ArrayList<String> friends = new ArrayList<String>();
```

```
friends.add("Cindy");
String name = friends.get(i);
friends.set(i, "Harry");
```

Use the `get` and `set` methods to access an element.

The `add` method appends an element to the array list, increasing its size.

The index must be ≥ 0 and < `friends.size()`.

ArrayListTester.java

```java
import java.util.ArrayList;

/**
     This program tests the ArrayList class.
*/
public class ArrayListTester
{
   public static void main(String[] args)
   {
      ArrayList<BankAccount> accounts = new ArrayList<BankAccount>();
      accounts.add(new BankAccount(1001));
      accounts.add(new BankAccount(1015));
      accounts.add(new BankAccount(1729));
      accounts.add(1, new BankAccount(1008));
      accounts.remove(0);

      System.out.println("Size: " + accounts.size());
      System.out.println("Expected: 3");
      BankAccount first = accounts.get(0);
      System.out.println("First account number: "
            + first.getAccountNumber());
      System.out.println("Expected: 1008");
      BankAccount last = accounts.get(accounts.size() - 1);
      System.out.println("Last account number: "
            + last.getAccountNumber());
      System.out.println("Expected: 1729");
   }
}
```

BankAccount.java

```
/**
    A bank account has a balance that can be changed by
    deposits and withdrawals.
*/
public class BankAccount
{
   private int accountNumber;
   private double balance;

   /**
      Constructs a bank account with a zero balance.
      @param anAccountNumber the account number for this account
   */
   public BankAccount(int anAccountNumber)
   {
      accountNumber = anAccountNumber;
      balance = 0;
   }

```

BankAccount.java (cont.)

```
    /**
        Constructs a bank account with a given balance
        @param anAccountNumber the account number for this account
        @param initialBalance the initial balance
    */
    public BankAccount(int anAccountNumber, double initialBalance)
    {
        accountNumber = anAccountNumber;
        balance = initialBalance;
    }

    /**
        Gets the account number of this bank account.
        @return the account number
    */
    public int getAccountNumber()
    {
        return accountNumber;
    }

```

BankAccount.java (cont.)

```
    /**
       Deposits money into the bank account.
       @param amount the amount to deposit
    */
    public void deposit(double amount)
    {
       double newBalance = balance + amount;
       balance = newBalance;
    }

    /**
       Withdraws money from the bank account.
       @param amount the amount to withdraw
    */
    public void withdraw(double amount)
    {
       double newBalance = balance - amount;
       balance = newBalance;
    }

```

BankAccount.java (cont.)

```
    /**
       Gets the current balance of the bank account.
       @return the current balance
    */
    public double getBalance()
    {
       return balance;
    }
}
```

Program Run:

```
Size: 3
Expected: 3
First account number: 1008
Expected: 1008
Last account number: 1729
Expected: 1729
```

Self Check

How do you construct an array of 10 strings? An array list of strings?

Answer:

```
new String[10];
new ArrayList<String>();
```

Self Check

What is the content of `names` after the following statements?

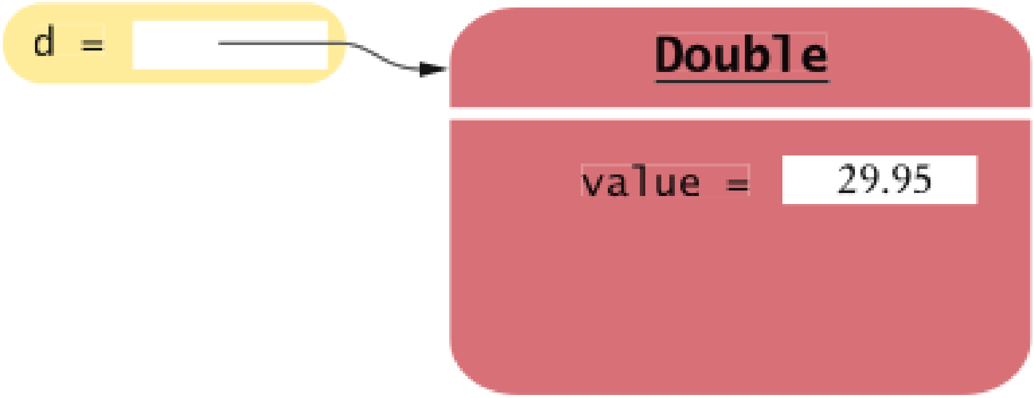
```
ArrayList<String> names = new ArrayList<String>();
names.add("A");
names.add(0, "B");
names.add("C");
names.remove(1);
```

Answer: `names` contains the strings `"B"` and `"C"` at positions 0 and 1

Wrapper Classes

- For each primitive type there is a **wrapper class** for storing values of that type:

```
Double d = new Double(29.95);
```



- Wrapper objects can be used anywhere that objects are required instead of primitive type values:

```
ArrayList<Double> values = new ArrayList<Double>();
values.add(29.95);
double x = values.get(0);
```

Wrappers

There are wrapper classes for all eight primitive types:

Primitive Type	Wrapper Class
byte	Byte
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

Auto-boxing

- **Auto-boxing:** Automatic conversion between primitive types and the corresponding wrapper classes:

```
Double d = 29.95; // auto-boxing; same as
                  // Double d = new Double(29.95);
double x = d; // auto-unboxing; same as
              // double x = d.doubleValue();
```

- Auto-boxing even works inside arithmetic expressions:

```
d = d + 1;
```

  Means:
  - *auto-unbox `d` into a `double`*
  - *add `1`*
  - *auto-box the result into a new `Double`*
  - *store a reference to the newly created wrapper object in `d`*

Auto-boxing and Array Lists

- To collect numbers in an array list, use the wrapper type as the type parameter, and then rely on auto-boxing:

```
ArrayList<Double> values = new ArrayList<Double>();
values.add(29.95);
double x = values.get(0);
```

- Storing wrapped numbers is quite inefficient
  - *Acceptable if you only collect a few numbers*
  - *Use arrays for long sequences of numbers or characters*

Self Check

What is the difference between the types `double` and `Double`?

Self Check

Suppose `values` is an `ArrayList<Double>` of size > 0. How do you increment the element with index 0?

Answer:

```
values.set(0, values.get(0) + 1);
```

The Enhanced `for` *Loop*

- Traverses all elements of a collection:

```
double[] values = ...;
double sum = 0;
for (double element : values)
{
   sum = sum + element;
}
```

- Read the loop as "for each `element` in `values`"
- Traditional alternative:

```
double[] values = ...;
double sum = 0;
for (int i = 0; i < values.length; i++)
{
   double element = values[i];
   sum = sum + element;
}
```

The Enhanced `for` *Loop*

- Works for `ArrayLists` too:

```
ArrayList<BankAccount> accounts = ...;
double sum = 0;
for (BankAccount account : accounts)
{
   sum = sum + account.getBalance();
}
```

- Equivalent to the following ordinary `for` loop:

```
double sum = 0;
for (int i = 0; i < accounts.size(); i++)
{
   BankAccount account = accounts.get(i);
   sum = sum + account.getBalance();
}
```

The Enhanced `for` Loop

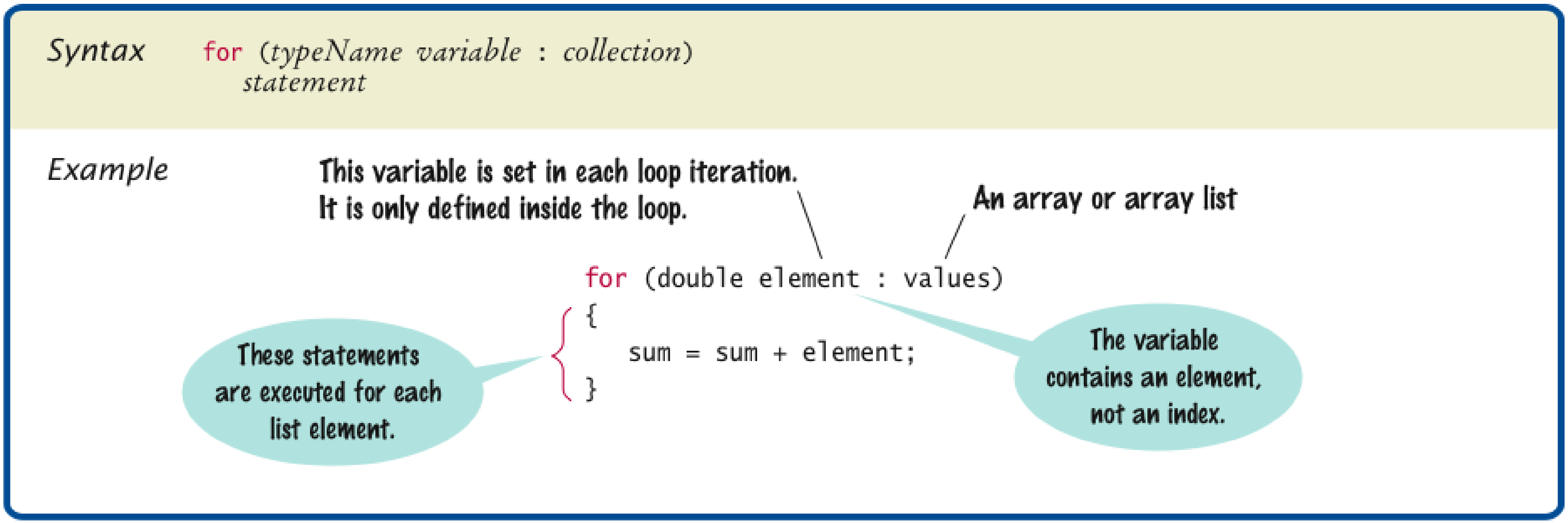
- The “for each loop” does not allow you to modify the contents of an array:

```
for (double element : values)
{
   element = 0;
   // ERROR-this assignment does not
   // modify array element
}
```

- Must use an ordinary `for` loop:

```
for (int i = 0; i < values.length; i++)
{
   values[i] = 0; // OK
}
```

Syntax *The "for each" Loop*



Self Check

Write a “for each” loop that prints all elements in the array `values`.

Answer:

```
for (double element : values)
   System.out.println(element);
```

Self Check

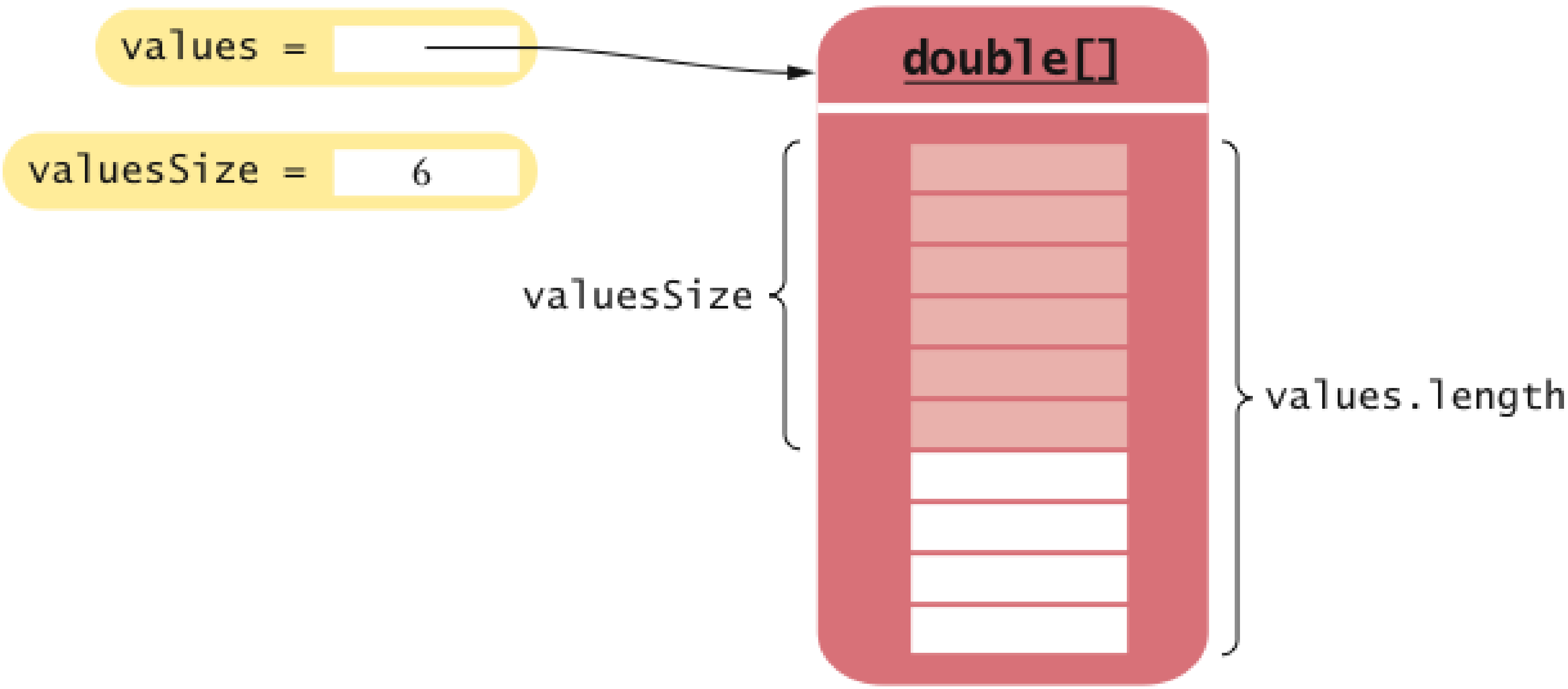
What does this “for each” loop do?

```
int counter = 0; for (BankAccount a: accounts)
{
   if (a.getBalance() == 0) { counter++; }
}
```

Partially Filled Arrays

- Array length = maximum number of elements in array
- Usually, array is partially filled
- Need companion variable to keep track of current size
    - *Uniform naming convention:*
```
final int VALUES_LENGTH = 100;
double[] values = new double[VALUES_LENGTH];
int valuesSize = 0;
```
    - *Update* `valuesSize` *as array is filled:*
```
values[valuesSize] = x;
valuesSize++;
```

Partially Filled Arrays



Partially Filled Arrays

- Example: Read numbers into a partially filled array:

```
int valuesSize = 0;
Scanner in = new Scanner(System.in);
while (in.hasNextDouble())
{
   if (valuesSize < values.length)
   {
      values[valuesSize] = in.nextDouble();
      valuesSize++;
   }
}
```

- To process the gathered array elements, use the companion variable, not the array length:

```
for (int i = 0; i < valuesSize; i++)
{
   System.out.println(values[i]);
}
```

Self Check

Write a loop to print the elements of the partially filled array `values` in reverse order, starting with the last element.

Answer:

```
for (int i = valuesSize - 1; i >= 0; i--)
   System.out.println(values[i]);
```

Self Check

How do you remove the last element of the partially filled array `values`?

Answer:

```
valuesSize--;
```

Self Check

Why would a programmer use a partially filled array of numbers instead of an array list?

Common Array Algorithms

- Filling
- Computing Sum and Average
- Counting Matches
- Finding the Maximum or Minimum
- Searching for a Value
- Locating the Position of an Element
- Removing an Element
- Inserting an Element
- Copying an Array
- Printing Element Separators

Bank.java

- Bank class stores an array list of bank accounts
- Methods of the Bank class use some of the previous algorithms:

```
import java.util.ArrayList;

/**
   This bank contains a collection of bank accounts.
*/
public class Bank
{
   private ArrayList<BankAccount> accounts;

   /**
      Constructs a bank with no bank accounts.
   */
   public Bank()
   {
      accounts = new ArrayList<BankAccount>();
   }

```

Bank.java (cont.)

```
    /**
       Adds an account to this bank.
       @param a the account to add
    */
    public void addAccount(BankAccount a)
    {
       accounts.add(a);
    }

    /**
       Gets the sum of the balances of all accounts in this bank.
       @return the sum of the balances
    */
    public double getTotalBalance()
    {
       double total = 0;
       for (BankAccount a : accounts)
       {
          total = total + a.getBalance();
       }
       return total;
    }
```

Bank.java (cont.)

```
    /**
       Counts the number of bank accounts whose balance is at
       least a given value.
       @param atLeast the balance required to count an account
       @return the number of accounts having least the given balance
    */
    public int countBalancesAtLeast(double atLeast)
    {
       int matches = 0;
       for (BankAccount a : accounts)
       {
          if (a.getBalance() >= atLeast) matches++; // Found a match
       }
       return matches;
    }

    /**
       Finds a bank account with a given number.
       @param accountNumber the number to find
       @return the account with the given number, or null if there
       is no such account
    */
    public BankAccount find(int accountNumber)
    {
       for (BankAccount a : accounts)
       {
          if (a.getAccountNumber() == accountNumber) // Found a match
             return a;
       }
       return null; // No match in the entire array list
    }

```

Bank.java (cont.)

```
    /**
       Gets the bank account with the largest balance.
       @return the account with the largest balance, or null if the
       bank has no accounts
    */
    public BankAccount getMaximum()
    {
       if (accounts.size() == 0) return null;
       BankAccount largestYet = accounts.get(0);
       for (int i = 1; i < accounts.size(); i++)
       {
          BankAccount a = accounts.get(i);
          if (a.getBalance() > largestYet.getBalance())
             largestYet = a;
       }
       return largestYet;
    }
}
```

BankTester.java

```
/**
   This program tests the Bank class.
*/
public class BankTester
{
   public static void main(String[] args)
   {
      Bank firstBankOfJava = new Bank();
      firstBankOfJava.addAccount(new BankAccount(1001, 20000));
      firstBankOfJava.addAccount(new BankAccount(1015, 10000));
      firstBankOfJava.addAccount(new BankAccount(1729, 15000));

      double threshold = 15000;
      int count = firstBankOfJava.countBalancesAtLeast(threshold);
      System.out.println("Count: " + count);
      System.out.println("Expected: 2");

      int accountNumber = 1015;
      BankAccount account = firstBankOfJava.find(accountNumber);
      if (account == null)
         System.out.println("No matching account");
      else
         System.out.println("Balance of matching account: "
            + account.getBalance());
      System.out.println("Expected: 10000");

      BankAccount max = firstBankOfJava.getMaximum();
      System.out.println("Account with largest balance: "
            + max.getAccountNumber());
      System.out.println("Expected: 1001");
   }
}
```

Program Run:

```
Count: 2
Expected: 2
Balance of matching account: 10000.0
Expected: 10000
Account with largest balance: 1001
Expected: 1001
```

Self Check

What does the `find` method do if there are two bank accounts with a matching account number?

Self Check

Would it be possible to use a “for each” loop in the `getMaximum` method?

Self Check

When printing separators, we skipped the separator before the initial element. Rewrite the loop so that the separator is printed *after* each element, except for the last element.

Answer:

```
for (int i = 0; i < values.size(); i++)
{
   System.out.print(values.get(i));
   if (i < values.size() - 1)
   {
      System.out.print(" | ");
   }
}
```

Now you know why we set up the loop the other way.

Self Check

The following replacement has been suggested for the algorithm that prints element separators:

```
System.out.print(names.get(0));
for (int i = 1; i < names.size(); i++)
   System.out.print(" | " + names.get(i));
```

What is problematic about this suggestion?

Regression Testing

- **Test suite:** a set of tests for repeated testing
- **Cycling:** bug that is fixed but reappears in later versions
- **Regression testing:** repeating previous tests to ensure that known failures of prior versions do not appear in new versions

BankTester.java

```
import java.util.Scanner;

/**
    This program tests the Bank class.
*/
public class BankTester
{
   public static void main(String[] args)
   {
      Bank firstBankOfJava = new Bank();
      firstBankOfJava.addAccount(new BankAccount(1001, 20000));
      firstBankOfJava.addAccount(new BankAccount(1015, 10000));
      firstBankOfJava.addAccount(new BankAccount(1729, 15000));

      Scanner in = new Scanner(System.in);

      double threshold = in.nextDouble();
      int c = firstBankOfJava.count(threshold);
      System.out.println("Count: " + c);
      int expectedCount = in.nextInt();
      System.out.println("Expected: " + expectedCount);

      int accountNumber = in.nextInt();
      BankAccount a = firstBankOfJava.find(accountNumber);
      if (a == null)
         System.out.println("No matching account");
      else
      {
         System.out.println("Balance of matching account: " + a.getBalance());
         int matchingBalance = in.nextInt();
         System.out.println("Expected: " + matchingBalance);
      }
   }
}
```

Regression Testing: Input Redirection

- Store the inputs in a file
- input1.txt:

```
15000
2
1015
10000
```

- Type the following command into a shell window:

```
java BankTester < input1.txt
```

- Program Run:

```
Count: 2
Expected: 2
Balance of matching account: 10000
Expected: 10000
```

Regression Testing: Output Redirection

- Output redirection:

```
java BankTester < input1.txt > output1.txt
```

Self Check

Suppose you modified the code for a method. Why do you want to repeat tests that already passed with the previous version of the code?

Self Check

Suppose a customer of your program finds an error. What action should you take beyond fixing the error?

Self Check

Why doesn't the `BankTester` program contain prompts for the inputs?

AGENDA

- Arrays & Array Lists
- Interfaces & Polymorphism
- Inheritance
- Input / Output & Exception Handling

AGENDA

- Arrays & Array Lists
- Interfaces & Polymorphism
- Inheritance
- Input / Output & Exception Handling

BASIC CONCEPTS OF JAVA 3

INTERFACE & POLYMORPHISM

Chapter Goals

- To be able to declare and use interface types
- To understand the concept of polymorphism
- To appreciate how interfaces can be used to decouple classes
- To learn how to implement helper classes as inner classes
- To implement event listeners in graphical applications

Using Interfaces for Algorithm Reuse

- Use *interface types* to make code more reusable
- Example: We create a `DataSet` to find the average and maximum of a set of *numbers*

Using Interfaces for Algorithm Reuse

```
public class DataSet {
   private double sum;
   private double maximum;
   private int count;
   ...
   public void add(double x)
   {
      sum = sum + x;
      if (count == 0 || maximum < x
         maximum = x;
      count++;
   }

   public double getMaximum()
   {
      return maximum;
   }
}
```

Using Interfaces for Algorithm Reuse

- What if we want to find the average and maximum of a set of `BankAccount` values?

Using Interfaces for Algorithm Reuse

```
public class DataSet // Modified for BankAccount objects
{
   private double sum;
   private BankAccount maximum;
   private int count;
   ...
   public void add(BankAccount x)
   {
      sum = sum + x.getBalance();
      if (count == 0 || maximum.getBalance() < x.getBalance())
         maximum = x;
      count++;
   }

   public BankAccount getMaximum()
   {
      return maximum;
   }
}
```

Using Interfaces for Algorithm Reuse

- What if we want to find the average and maximum of a set of `Coin` values?

Using Interfaces for Algorithm Reuse

```
public class DataSet // Modified for Coin objects
{
   private double sum;
   private Coin maximum;
   private int count;
   ...
   public void add(Coin x)
   {
      sum = sum + x.getValue();
      if (count == 0 || maximum.getValue() < x.getValue())
                       maximum = x;
      count++;
   }
   public Coin getMaximum()
   {
      return maximum;
   }
}
```

Using Interfaces for Algorithm Reuse

- The algorithm for the data analysis service is the same in all cases; details of measurement differ
- Classes could agree on a method that obtains the measure to be used in the analysis .
- Suppose the method is called **`getMeasure`**
- We can implement a single reusable `DataSet` class whose `add` method looks like this:

```
sum = sum + x.getMeasure();
if (count == 0 || maximum.getMeasure() < x.getMeasure())
   maximum = x;
count++;
```

Using Interfaces for Algorithm Reuse

- What is the type of the variable `x`?
    - *`x` should refer to any class that has a `getMeasure` method*
- In Java, an **interface type** is used to specify required operations:

```
public interface Measurable
{
   double getMeasure();
}
```

- Interface declaration lists all methods that the interface type requires

Syntax Declaring an Interface

Syntax

```
public interface InterfaceName
{
    method signatures
}
```

Example

```
public interface Measurable
{
   double getMeasure();
}
```

The methods of an interface are automatically public.

No implementation is provided.

Interfaces vs. Classes

An interface type is similar to a class, but there are several important differences:

- *All methods in an interface type are* ***abstract****; they don't have an implementation*
- *All methods in an interface type are automatically public*
- *An interface type does not have instance fields*
- *Interface cannot be instantiated*

Generic **DataSet** *for Measurable Objects*

```
public class DataSet
{
   private double sum;
   private Measurable maximum;
   private int count;
   ...
   public void add(Measurable x)
   {
      sum = sum + x.getMeasure();
      if (count == 0 || maximum.getMeasure() < x.getMeasure())
         maximum = x;
      count++;
   }

   public Measurable getMaximum()
   {
      return maximum;
   }
}
```

Implementing an Interface Type

- Use `implements` reserved word to indicate that a class implements an interface type:

```
public class BankAccount implements Measurable
{
  public double getMeasure()
  {
     ...
     return balance;
  }
}
```

- A class can implement more than one interface type
  - *Class must declare all the methods that are required by all the interfaces it implements*

Implementing an Interface Type

- Another example:

```
public class Coin implements Measurable
{
   public double getMeasure()
   {
      return value;
   }
   ...
}
```

Code Reuse

- A service type such as DataSet specifies an interface for participating in the service
- Use interface types to make code more reusable

Syntax Implementing an Interface

Syntax

```
public class ClassName implements InterfaceName, InterfaceName, . . .
{
   instance variables
   methods
}
```

Example

```
public class BankAccount implements Measurable
{
   . . .
   public double getMeasure()
   {
      return balance;
   }
   . . .
}
```

List all interface types that this class implements.

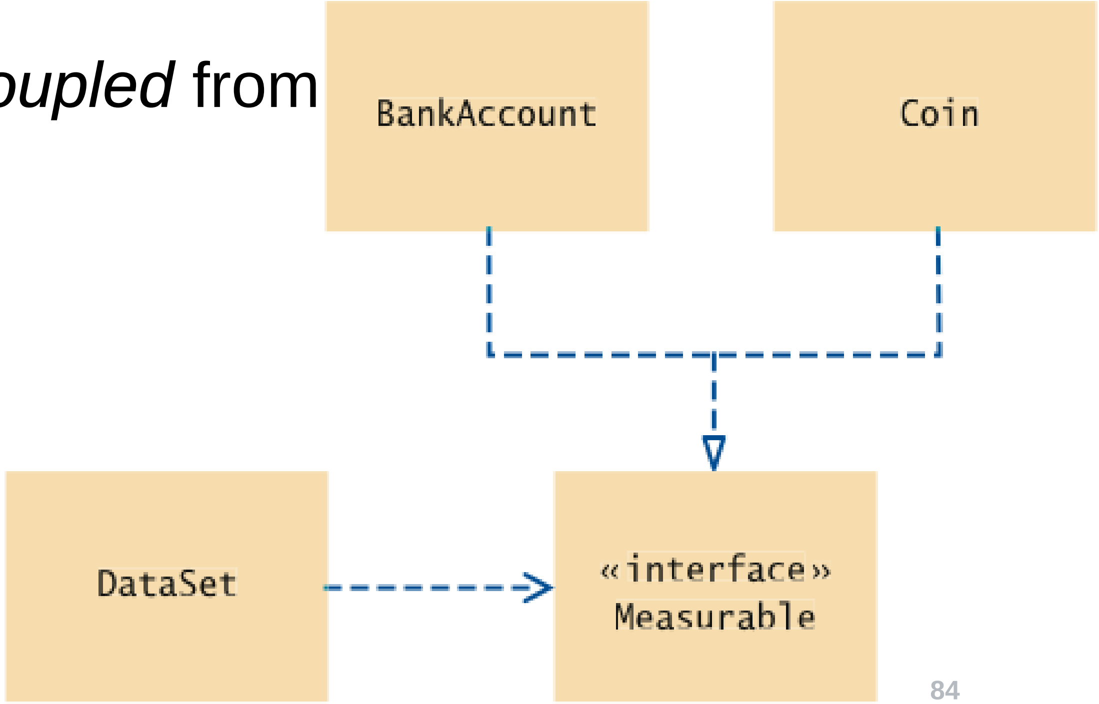
BankAccount instance variables

This method provides the implementation for the method declared in the interface.

Other BankAccount methods

UML Diagram of `DataSet` *and Related Classes*

- Interfaces can reduce the coupling between classes
- UML notation:
  - *Interfaces are tagged with a "stereotype" indicator «interface»*
  - *A dotted arrow with a triangular tip denotes the "is-a" relationship between a class and an interface*
  - *A dotted line with an open v-shaped arrow tip denotes the "uses" relationship or dependency*
- Note that `DataSet` is *decoupled* from `BankAccount` and `Coin`



DataSetTester.java

```java
/**
    This program tests the DataSet class.
*/
public class DataSetTester
{
   public static void main(String[] args)
   {
      DataSet bankData = new DataSet();

      bankData.add(new BankAccount(0));
      bankData.add(new BankAccount(10000));
      bankData.add(new BankAccount(2000));

      System.out.println("Average balance: " + bankData.getAverage());
      System.out.println("Expected: 4000");
      Measurable max = bankData.getMaximum();
      System.out.println("Highest balance: " + max.getMeasure());
      System.out.println("Expected: 10000");

      DataSet coinData = new DataSet();

      coinData.add(new Coin(0.25, "quarter"));
      coinData.add(new Coin(0.1, "dime"));
      coinData.add(new Coin(0.05, "nickel"));

      System.out.println("Average coin value: " + coinData.getAverage());
      System.out.println("Expected: 0.133");
      max = coinData.getMaximum();
      System.out.println("Highest coin value: " + max.getMeasure());
      System.out.println("Expected: 0.25");
   }
}
```

Program Run:

```
Average balance: 4000.0
Expected: 4000
Highest balance: 10000.0
Expected: 10000
Average coin value: 0.13333333333333333
Expected: 0.133
Highest coin value: 0.25
Expected: 0.25
```

Self Check

Suppose you want to use the `DataSet` class to find the `Country` object with the largest population. What condition must the `Country` class fulfill?

Self Check

Why can't the `add` method of the `DataSet` class have a parameter of type `Object`?

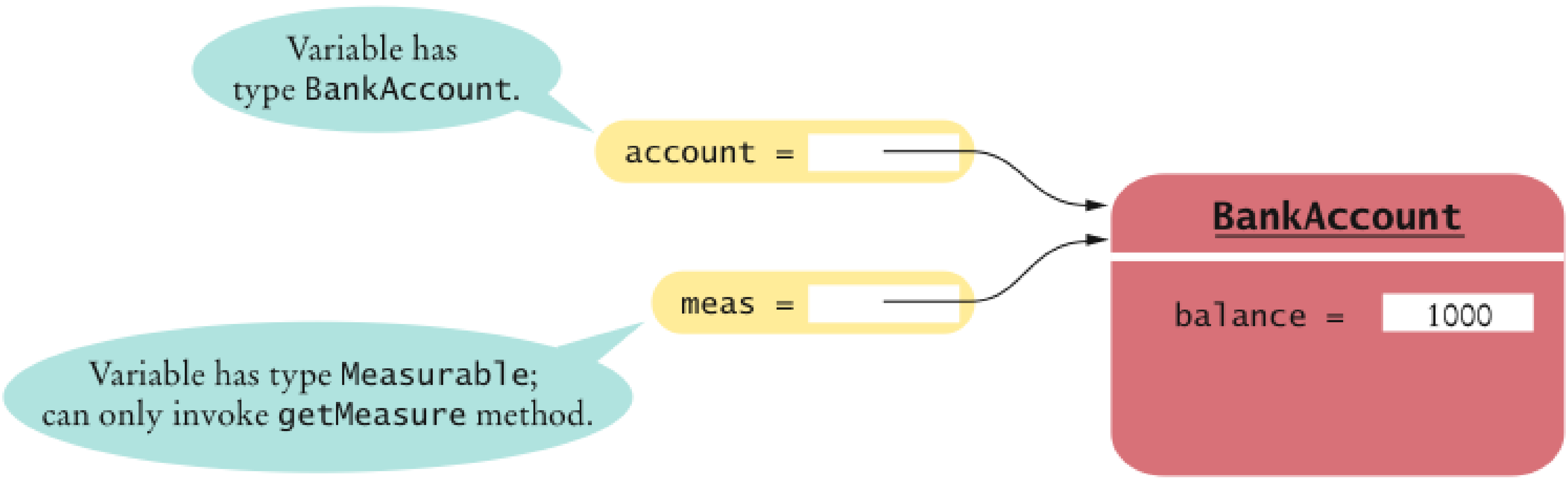
Converting Between Class and Interface Types

- You can convert from a class type to an interface type, provided the class implements the interface
- ```
  BankAccount account = new BankAccount(10000);
  Measurable x = account; // OK
  ```
- ```
  Coin dime = new Coin(0.1, "dime");
  Measurable x = dime; // Also OK
  ```
- Cannot convert between unrelated types:

  ```
  Measurable x = new Rectangle(5, 10, 20, 30); // ERROR
  ```

  Because `Rectangle` doesn't implement `Measurable`

Variables of Class and Interface Types



Casts

- Add `Coin` objects to `DataSet`:

```
DataSet coinData = new DataSet();
coinData.add(new Coin(0.25, "quarter"));
coinData.add(new Coin(0.1, "dime"));
coinData.add(new Coin(0.05, ”nickel"));
Measurable max = coinData.getMaximum(); // Get the largest coin
```

- What can you do with `max`? It’s not of type `Coin`:

```
String name = max.getName(); // ERROR
```

- You need a cast to convert from an interface type to a class type
- You know it’s a `Coin`, but the compiler doesn’t. Apply a cast:

```
Coin maxCoin = (Coin) max;
String name = maxCoin.getName();
```

Casts

- If you are wrong and `max` isn't a coin, the program throws an exception and terminates
- Difference with casting numbers:
    - *When casting number types you agree to the information loss*
    - *When casting object types you agree to that risk of causing an exception*

Self Check

Can you use a cast `(BankAccount) x` to convert a `Measurable` variable `x` to a `BankAccount` reference?

Self Check

If both `BankAccount` and `Coin` implement the `Measurable` interface, can a `Coin` reference be converted to a `BankAccount` reference?

Polymorphism

- An interface variable holds a reference to object of a class that implements the interface:

```
Measurable meas;
meas = new BankAccount(10000);
meas = new Coin(0.1, "dime");
```

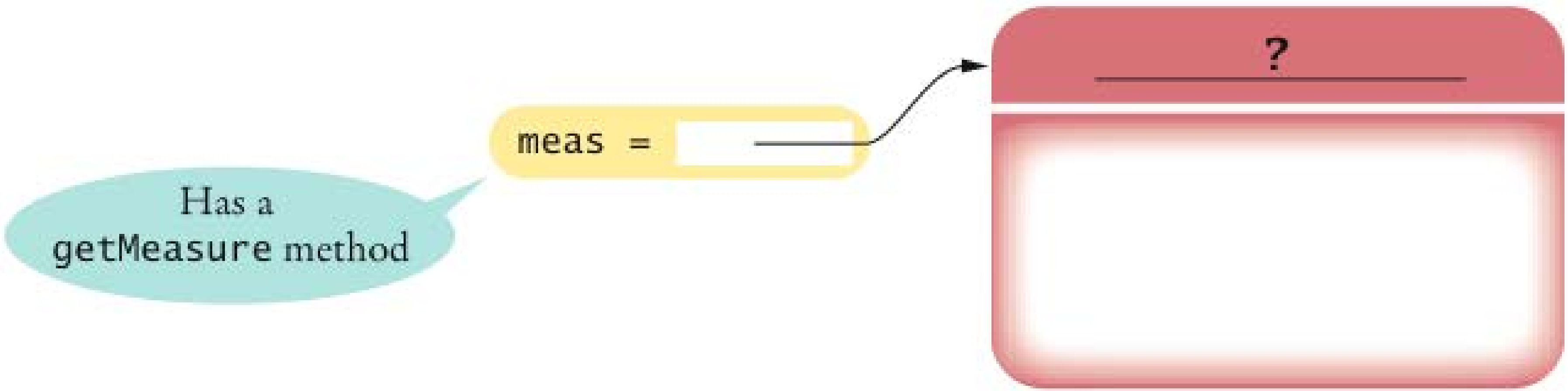
  Note that the object to which `meas` refers doesn't have type `Measurable`; the type of the object is some class that implements the `Measurable` interface
- You can call any of the interface methods:

```
double m = meas.getMeasure();
```

- Which method is called?

Interface Reference

- An interface reference can refer to an object of any class that implements the interface



Polymorphism

- When the virtual machine calls an instance method, it locates the method of the implicit parameter's class - called *dynamic method lookup*
- If `meas` refers to a `BankAccount` object, then `meas.getMeasure()` calls the `BankAccount.getMeasure` method
- If `meas` refers to a `Coin` object, then method `Coin.getMeasure` is called
- Polymorphism (many shapes) denotes the ability to treat objects with differences in behavior in a uniform way

Self Check

Why is it impossible to construct a `Measurable` object?

Self Check

Why can you nevertheless declare a variable whose type is `Measurable`?

Self Check

What does this code fragment print? Why is this an example of polymorphism?

```
DataSet data = new DataSet();
data.add(new BankAccount(1000));
data.add(new Coin(0.1, "dime"));
System.out.println(data.getAverage());
```

Answer: The code fragment prints 500.05. Each call to `add` results in a call `x.getMeasure()`.
In the first call, `x` is a `BankAccount`.
In the second call, `x` is a `Coin`.
A different `getMeasure` method is called in each case. The first call returns the account balance, the second one the coin value.

Using Interfaces for Callbacks

- Limitations of `Measurable` interface:
  - *Can add `Measurable` interface only to classes under your control*
  - *Can measure an object in only one way*
  - *E.g., cannot analyze a set of savings accounts both by bank balance and by interest rate*
- **Callback:** a mechanism for specifying code that is executed at a later time
- In previous `DataSet` implementation, responsibility of measuring lies with the added objects themselves

Using Interfaces for Callbacks

- Alternative: Hand the object to be measured to a method of an interface:

```
public interface Measurer
{
   double measure(Object anObject);
}
```

- `Object` is the “lowest common denominator” of all classes

Using Interfaces for Callbacks

- The code that makes the call to the callback receives an object of class that implements this interface:

```
public DataSet(Measurer aMeasurer)
{
   sum = 0;
   count = 0;
   maximum = null;
   measurer = aMeasurer; // Measurer instance variable
}
```

- The measurer instance variable carries out the measurements:

```
public void add(Object x)
{
   sum = sum + measurer.measure(x);
   if (count == 0 || measurer.measure(maximum) < measurer.measure(x))
      maximum = x;
   count++;
}
```

Using Interfaces for Callbacks

- A specific callback is obtained by implementing the Measurer interface:

```
public class RectangleMeasurer implements Measurer
{
   public double measure(Object anObject)
   {
      Rectangle aRectangle = (Rectangle) anObject;
      double area = aRectangle.getWidth() *
         aRectangle.getHeight();
      return area;
   }
}
```

- Must cast from `Object` to `Rectangle`:

```
Rectangle aRectangle = (Rectangle) anObject;
```

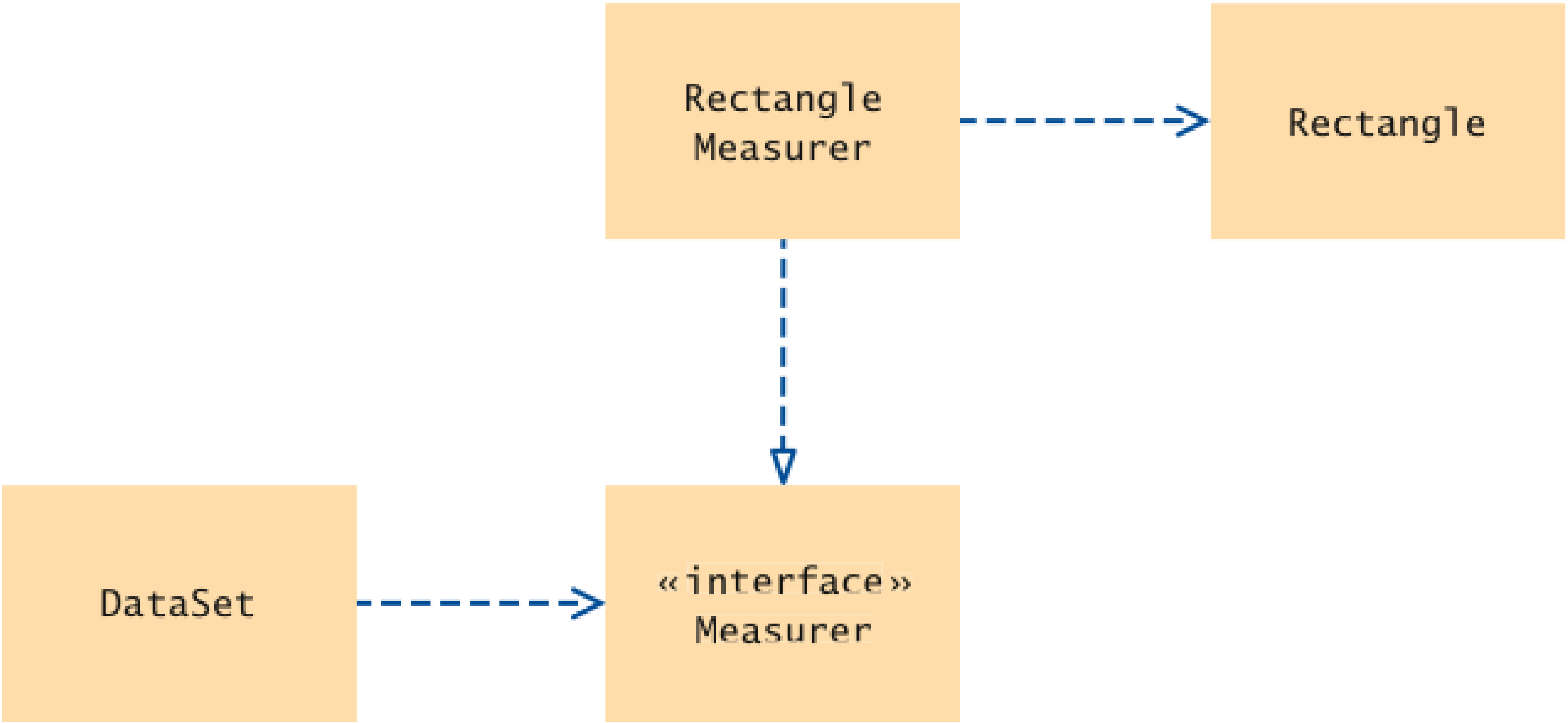
Using Interfaces for Callbacks

- Pass measurer to data set constructor:

```
Measurer m = new RectangleMeasurer();
DataSet data = new DataSet(m);
data.add(new Rectangle(5, 10, 20, 30));
data.add(new Rectangle(10, 20, 30, 40));
...
```

UML Diagram of `Measurer` *Interface and Related Classes*

Note that the `Rectangle` class is decoupled from the `Measurer` interface



Measurer.java

```
/**
    Describes any class whose objects can measure other objects.
*/
public interface Measurer
{
    /**
        Computes the measure of an object.
        @param anObject the object to be measured
        @return the measure
    */
    double measure(Object anObject);
}
```

RectangleMeasurer.java

```
import java.awt.Rectangle;

/**
   Objects of this class measure rectangles by area.
*/
public class RectangleMeasurer implements Measurer
{
   public double measure(Object anObject)
   {
      Rectangle aRectangle = (Rectangle) anObject;
      double area = aRectangle.getWidth() * aRectangle.getHeight();
      return area;
   }
}
```

DataSet.java

```
/**
    Computes the average of a set of data values.
*/
public class DataSet
{
   private double sum;
   private Object maximum;
   private int count;
   private Measurer measurer;

   /**
      Constructs an empty data set with a given measurer.
      @param aMeasurer the measurer that is used to measure data values
   */
   public DataSet(Measurer aMeasurer)
   {
      sum = 0;
      count = 0;
      maximum = null;
      measurer = aMeasurer;
   }

```

DataSet.java (cont.)

```
    /**
       Adds a data value to the data set.
       @param  x a data value
    */
    public void add(Object x)
    {
       sum = sum + measurer.measure(x);
       if (count == 0 || measurer.measure(maximum) < measurer.measure(x))
          maximum = x;
       count++;
    }

    /**
       Gets the average of the added data.
       @return the average or 0 if no data has been added
    */
    public double getAverage()
    {
       if (count == 0) return 0;
       else return sum / count;
    }

    /**
       Gets the largest of the added data.
       @return the maximum or 0 if no data has been added
    */
    public Object getMaximum()
    {
       return maximum;
    }
 }
```

DataSetTester2.java

```java
import java.awt.Rectangle;

/**
   This program demonstrates the use of a Measurer.
*/
public class DataSetTester2
{
   public static void main(String[] args)
   {
      Measurer m = new RectangleMeasurer();

      DataSet data = new DataSet(m);

      data.add(new Rectangle(5, 10, 20, 30));
      data.add(new Rectangle(10, 20, 30, 40));
      data.add(new Rectangle(20, 30, 5, 15));

      System.out.println("Average area: " + data.getAverage());
      System.out.println("Expected: 625");

      Rectangle max = (Rectangle) data.getMaximum();
      System.out.println("Maximum area rectangle: " + max);
      System.out.println("Expected: "
         + "java.awt.Rectangle[x=10,y=20,width=30,height=40]");
   }
}
```

Program Run:

```
Average area: 625
Expected: 625
Maximum area rectangle:java.awt.Rectangle[x=10,y=20,width=30,height=40]
Expected: java.awt.Rectangle[x=10,y=20,width=30,height=40]
```

Self Check

Suppose you want to use the `DataSet` class to find the longest `String` from a set of inputs. Why can't this work?

Self Check

How can you use the `DataSet` class of this section to find the longest `String` from a set of inputs?

Self Check

Why does the `measure` method of the `Measurer` interface have one more parameter than the `getMeasure` method of the `Measurable` interface?

Inner Classes

- Trivial class can be declared inside a method:

```
public class DataSetTester3
{
   public static void main(String[] args)
   {
      class RectangleMeasurer implements Measurer
      {
         ...
      }
      Measurer m = new RectangleMeasurer();
      DataSet data = new DataSet(m);
      ...
   }
}
```

Inner Classes

- If inner class is declared inside an enclosing class, but outside its methods, it is available to all methods of enclosing class:

```
public class DataSetTester3
{
   class RectangleMeasurer implements Measurer
   {
      . . .
   }

   public static void main(String[] args)
   {
      Measurer m = new RectangleMeasurer();
      DataSet data = new DataSet(m);
      . . .
   }
}
```

- Compiler turns an inner class into a regular class file:

```
DataSetTester$1$RectangleMeasurer.class
```

DataSetTester3.java

```
import java.awt.Rectangle;

/**
   This program demonstrates the use of an inner class.
*/
public class DataSetTester3
{
   public static void main(String[] args)
   {
      class RectangleMeasurer implements Measurer
      {
         public double measure(Object anObject)
         {
            Rectangle aRectangle = (Rectangle) anObject;
            double area
                  = aRectangle.getWidth() * aRectangle.getHeight();
            return area;
         }
      }

      Measurer m = new RectangleMeasurer();

      DataSet data = new DataSet(m);

      data.add(new Rectangle(5, 10, 20, 30));
      data.add(new Rectangle(10, 20, 30, 40));
      data.add(new Rectangle(20, 30, 5, 15));

      System.out.println("Average area: " + data.getAverage());
      System.out.println("Expected: 625");

      Rectangle max = (Rectangle) data.getMaximum();
      System.out.println("Maximum area rectangle: " + max);
      System.out.println("Expected: "
         + "java.awt.Rectangle[x=10,y=20,width=30,height=40]");
   }
}
```

Self Check

Why would you use an inner class instead of a regular class?

Self Check

How many class files are produced when you compile the `DataSetTester3` program?

Mock Objects

- Want to test a class before the entire program has been completed
- A **mock object** provides the same services as another object, but in a simplified manner
- **Example:** a grade book application, `GradingProgram`, manages quiz scores using class `GradeBook` with methods:

```
public void addScore(int studentId, double score)
public double getAverageScore(int studentId)
public void save(String filename)
```

- Want to test `GradingProgram` without having a fully functional `GradeBook` class

Mock Objects

- Declare an interface type with the same methods that the `GradeBook` class provides
  - *Convention: use the letter* `I` *as a prefix for the interface name:*

```
public interface IGradeBook
{
   void addScore(int studentId, double score);
   double getAverageScore(int studentId);
   void save(String filename);
   . . .
}
```

- The `GradingProgram` class should *only* use this interface, never the `GradeBook` class which implements this interface

Mock Objects

- Meanwhile, provide a simplified mock implementation, restricted to the case of one student and without saving functionality:

```
public class MockGradeBook implements IGradeBook
{
   private ArrayList<Double> scores;
   public void addScore(int studentId, double score)
   {
      // Ignore studentId
      scores.add(score);
   }
   double getAverageScore(int studentId)
   {
      double total = 0;
      for (double x : scores) { total = total + x; }
      return total / scores.size();
   }
   void save(String filename)
   {
      // Do nothing
   }
   . . .
}
```

Mock Objects

- Now construct an instance of `MockGradeBook` and use it immediately to test the `GradingProgram` class
- When you are ready to test the actual class, simply use a `GradeBook` instance instead
- Don't erase the mock class - it will still come in handy for regression testing

Self Check

Why is it necessary that the real class and the mock class implement the same interface type?

Self Check

Why is the technique of mock objects particularly effective when the `GradeBook` and `GradingProgram` class are developed by two programmers?

Events

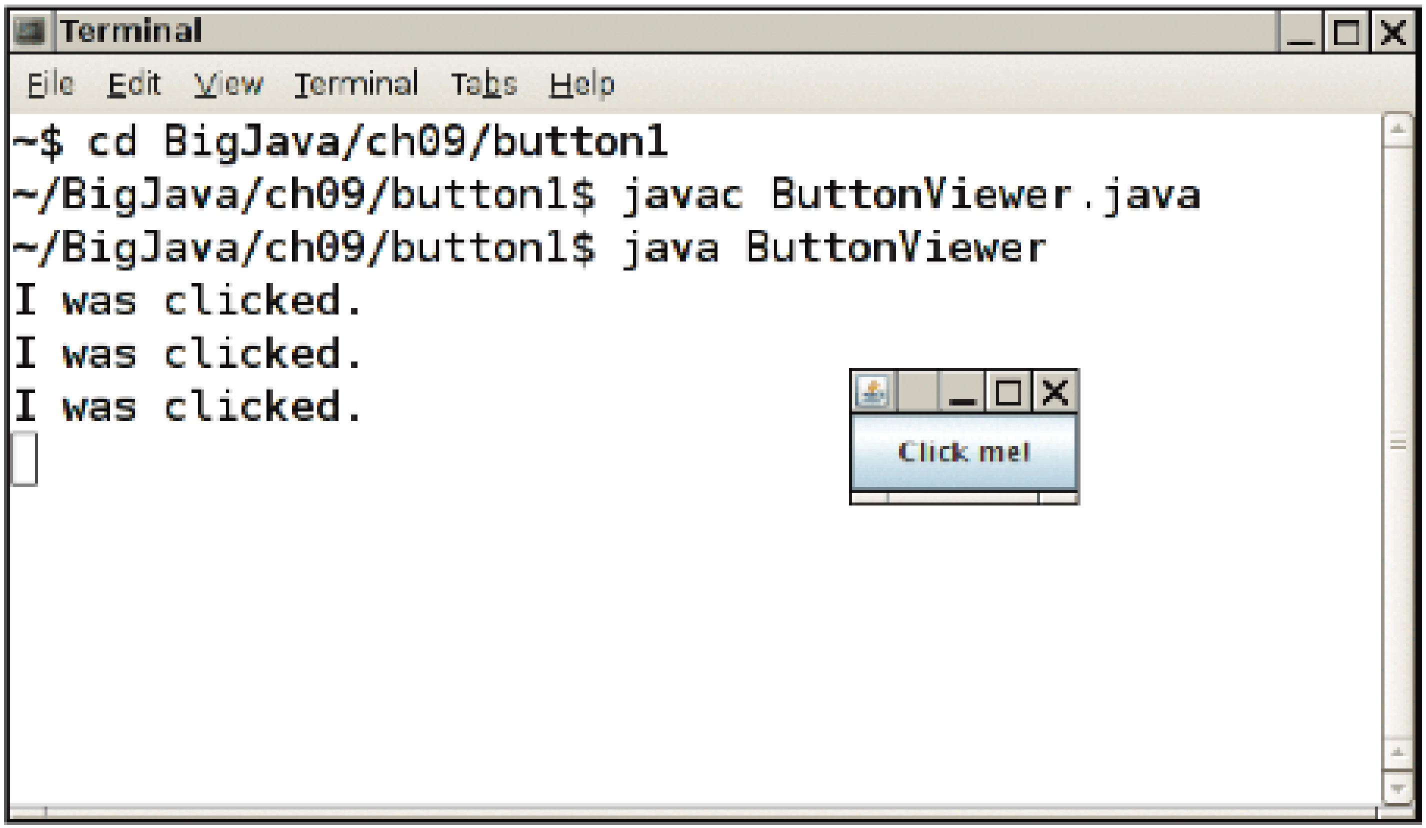
- User interface *events* include key presses, mouse moves, button clicks, and so on
- Most programs don’t want to be flooded by boring events
- A program can indicate that it only cares about certain specific events

Event Sources and Event Listeners

- **Event listener:**
  - *Notified when event happens*
  - *Belongs to a class that is provided by the application programmer*
  - *Its methods describe the actions to be taken when an event occurs*
  - *A program indicates which events it needs to receive by installing event listener objects*
- **Event source:**
  - *User interface component that generates a particular event*
  - *Add an event listener object to the appropriate event source*
  - *When an event occurs, the event source notifies all event listeners*

Events, Event Sources, and Event Listeners

- Example: Implementing an action listener - A program that prints a message whenever a button is clicked:



Events, Event Sources, and Event Listeners

- Use `JButton` components for buttons; attach an `ActionListener` to each button
- `ActionListener` interface:

```
public interface ActionListener
{
   void actionPerformed(ActionEvent event);
}
```

- Need to supply a class whose `actionPerformed` method contains instructions to be executed when button is clicked
- `event` parameter contains details about the event, such as the time at which it occurred
- Construct an object of the listener and add it to the button:

```
ActionListener listener = new ClickListener();
button.addActionListener(listener);
```

ClickListener.java

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
   An action listener that prints a message.
*/
public class ClickListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      System.out.println("I was clicked.");
   }
}
```

ButtonViewer.java

```
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;

/**
    This program demonstrates how to install an action listener.
*/
public class ButtonViewer
{
   private static final int FRAME_WIDTH = 100;
   private static final int FRAME_HEIGHT = 60;

   public static void main(String[] args)
   {
      JFrame frame = new JFrame();
      JButton button = new JButton("Click me!");
      frame.add(button);

      ActionListener listener = new ClickListener();
      button.addActionListener(listener);

      frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

Self Check

Which objects are the event source and the event listener in the `ButtonViewer` program?

Self Check

Why is it legal to assign a `ClickListener` object to a variable of type `ActionListener`?

Using Inner Classes for Listeners

- Implement simple listener classes as inner classes like this:

```
 JButton button = new JButton("...");
// This inner class is declared in the same method as the
// button variable
 class MyListener implements ActionListener
{
    ...
};
ActionListener listener = new MyListener();
button.addActionListener(listener);
```

- This places the trivial listener class exactly where it is needed, without cluttering up the remainder of the project
- Methods of an inner class can access the variables from the enclosing scope
  - *Local variables that are accessed by an inner class method must be declared as final*

Using Inner Classes for Listeners

Example: Add interest to a bank account whenever a button is clicked:

```
JButton button = new JButton("Add Interest");
final BankAccount account = new BankAccount(INITIAL_BALANCE);
// This inner class is declared in the same method as
// the account and button variables.
class AddInterestListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
   {
      // The listener method accesses the account
      // variable from the surrounding block
      double interest = account.getBalance() * INTEREST_RATE / 100;
      account.deposit(interest);
   }
};
ActionListener listener = new AddInterestListener();
button.addActionListener(listener);
```

InvestmentViewer1.java

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;

/**
   This program demonstrates how an action listener can access
   a variable from a surrounding block.
*/
public class InvestmentViewer1
{
   private static final int FRAME_WIDTH = 120;
   private static final int FRAME_HEIGHT = 60;

   private static final double INTEREST_RATE = 10;
   private static final double INITIAL_BALANCE = 1000;

   public static void main(String[] args)
   {
      JFrame frame = new JFrame();

```

InvestmentViewer1.java (cont.)

```
        // The button to trigger the calculation
        JButton button = new JButton("Add Interest");
        frame.add(button);

        // The application adds interest to this bank account
        final BankAccount account = new BankAccount(INITIAL_BALANCE);

        class AddInterestListener implements ActionListener
        {
           public void actionPerformed(ActionEvent event)
           {
              // The listener method accesses the account variable
              // from the surrounding block
              double interest = account.getBalance() * INTEREST_RATE / 100;
              account.deposit(interest);
              System.out.println("balance: " + account.getBalance());
           }
        }

        ActionListener listener = new AddInterestListener();
        button.addActionListener(listener);

        frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
     }
  }
```

Program Run:

```
balance: 1100.0
balance: 1210.0
balance: 1331.0
balance: 1464.1
```

Self Check

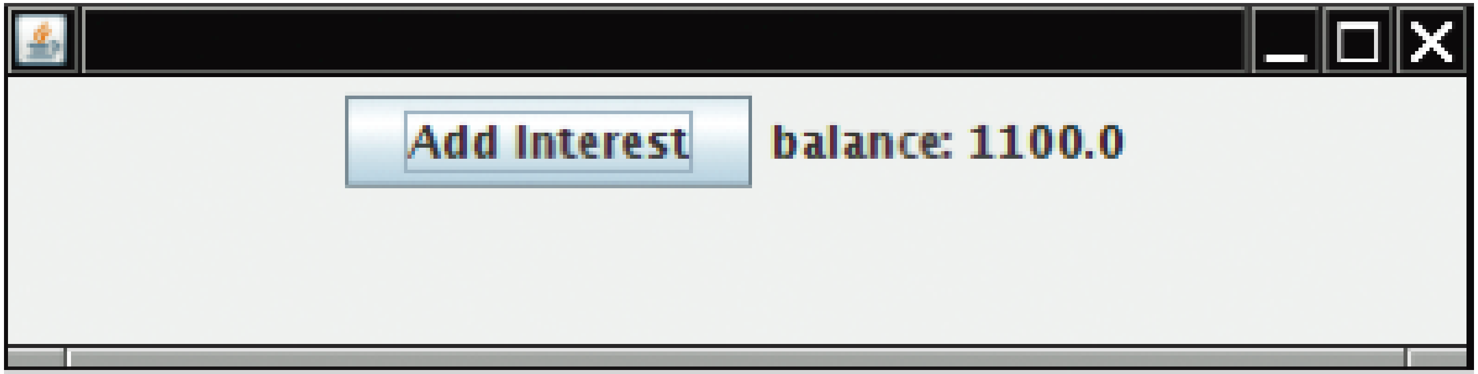
Why would an inner class method want to access a variable from a surrounding scope?

Self Check

Why would an inner class method want to access a variable from a surrounding If an inner class accesses a local variable from a surrounding scope, what special rule applies?

Building Applications with Buttons

- Example: Investment viewer program; whenever button is clicked, interest is added, and new balance is displayed:



Building Applications with Buttons

- Construct an object of the `JButton` class:

```
JButton button = new JButton("Add Interest");
```

- We need a user interface component that displays a message:

```
JLabel label = new JLabel("balance: ”
   + account.getBalance());
```

- Use a `JPanel` container to group multiple user interface components together:

```
JPanel panel = new JPanel();
panel.add(button);
panel.add(label);
frame.add(panel);
```

Building Applications with Buttons

- Listener class adds interest and displays the new balance:

```
class AddInterestListener implements ActionListener
 {
    public void actionPerformed(ActionEvent event)
    {
       double interest = account.getBalance() *
          INTEREST_RATE / 100;
       account.deposit(interest);
       label.setText("balance=" + account.getBalance());
    }
 }
```

- Add `AddInterestListener` as inner class so it can have access to surrounding `final` variables (`account` and `label`)

InvestmentViewer2.java

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

/**
   This program displays the growth of an investment.
*/
public class InvestmentViewer2
{
   private static final int FRAME_WIDTH = 400;
   private static final int FRAME_HEIGHT = 100;

   private static final double INTEREST_RATE = 10;
   private static final double INITIAL_BALANCE = 1000;

   public static void main(String[] args)
   {
      JFrame frame = new JFrame();

      // The button to trigger the calculation
      JButton button = new JButton("Add Interest");

      // The application adds interest to this bank account
      final BankAccount account = new BankAccount(INITIAL_BALANCE);

```

InvestmentViewer2.java (cont.)

```
      // The label for displaying the results
      final JLabel label = new JLabel("balance: " + account.getBalance());

      // The panel that holds the user interface components
      JPanel panel = new JPanel();
      panel.add(button);
      panel.add(label);
      frame.add(panel);

      class AddInterestListener implements ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {
            double interest = account.getBalance() * INTEREST_RATE / 100;
            account.deposit(interest);
            label.setText("balance: " + account.getBalance());
         }
      }

      ActionListener listener = new AddInterestListener();
      button.addActionListener(listener);

      frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

Self Check

How do you place the `"balance: ..."` message to the left of the `"Add Interest"` button?

Self Check

Why was it not necessary to declare the `button` variable as `final`?

Processing Timer Events

- `javax.swing.Timer` generates equally spaced timer events, sending events to installed action listeners
- Useful whenever you want to have an object updated in regular intervals
- Declare a class that implements the `ActionListener` interface:

```
class MyListener implements ActionListener
{
   void actionPerformed(ActionEvent event)
   {
      Listener action (executed at each timer event)
   }
}
```

- Add listener to timer and start timer:

```
MyListener listener = new MyListener();
Timer t = new Timer(interval, listener);
t.start();
```

RectangleComponent.java

Displays a rectangle that can be moved

The `repaint` method causes a component to repaint itself. Call this method whenever you modify the shapes that the `paintComponent` method draws

```
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import javax.swing.JComponent;

/**
   This component displays a rectangle that can be moved.
*/
public class RectangleComponent extends JComponent
{
   private static final int BOX_X = 100;
   private static final int BOX_Y = 100;
   private static final int BOX_WIDTH = 20;
   private static final int BOX_HEIGHT = 30;

```

RectangleComponent.java (cont.)

```
   private Rectangle box;

   public RectangleComponent()
   {
      // The rectangle that the paintComponent method draws
      box = new Rectangle(BOX_X, BOX_Y, BOX_WIDTH, BOX_HEIGHT);
   }

   public void paintComponent(Graphics g)
   {
      Graphics2D g2 = (Graphics2D) g;

      g2.draw(box);
   }

   /**
      Moves the rectangle by a given amount.
      @param x the amount to move in the x-direction
      @param y the amount to move in the y-direction
   */
   public void moveBy(int dx, int dy)
   {
      box.translate(dx, dy);
      repaint();
   }
}
```

RectangleMover.java

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.Timer;

/**
  This program moves the rectangle.
*/
public class RectangleMover
{
   private static final int FRAME_WIDTH = 300;
   private static final int FRAME_HEIGHT = 400;

   public static void main(String[] args)
   {
      JFrame frame = new JFrame();

      frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
      frame.setTitle("An animated rectangle");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

```

RectangleMover.java (cont.)

```
        final RectangleComponent component = new RectangleComponent();
        frame.add(component);

        frame.setVisible(true);

        class TimerListener implements ActionListener
        {
           public void actionPerformed(ActionEvent event)
           {
              component.moveBy(1, 1);
           }
        }

        ActionListener listener = new TimerListener();

        final int DELAY = 100; // Milliseconds between timer ticks
        Timer t = new Timer(DELAY, listener);
        t.start();
     }
  }
```

Self Check

Why does a timer require a listener object?

Self Check

What would happen if you omitted the call to `repaint` in the `moveBy` method?

Mouse Events

- Use a mouse listener to capture mouse events
- Implement the `MouseListener` interface:

```
public interface MouseListener
{
   void mousePressed(MouseEvent event);
   // Called when a mouse button has been pressed on a
   // component
   void mouseReleased(MouseEvent event);
   // Called when a mouse button has been released on a
   // component
   void mouseClicked(MouseEvent event);
   // Called when the mouse has been clicked on a component
   void mouseEntered(MouseEvent event);
   // Called when the mouse enters a component
   void mouseExited(MouseEvent event);
   // Called when the mouse exits a component
}
```

Mouse Events

- `mousePressed`, `mouseReleased`: Called when a mouse button is pressed or released
- `mouseClicked`: If button is pressed and released in quick succession, and mouse hasn’t moved
- `mouseEntered`, `mouseExited`: Mouse has entered or exited the component’s area
- Add a mouse listener to a component by calling the `addMouseListener` method:

```
public class MyMouseListener implements MouseListener
{
   // Implements five methods
}
MouseListener listener = new MyMouseListener();
component.addMouseListener(listener);
```

- Sample program: enhance `RectangleComponent` - when user clicks on rectangle component, move the rectangle

RectangleComponent.java

```
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import javax.swing.JComponent;

/**
   This component displays a rectangle that can be moved.
*/
public class RectangleComponent extends JComponent
{
   private static final int BOX_X = 100;
   private static final int BOX_Y = 100;
   private static final int BOX_WIDTH = 20;
   private static final int BOX_HEIGHT = 30;

   private Rectangle box;

   public RectangleComponent()
   {
      // The rectangle that the paintComponent method draws
      box = new Rectangle(BOX_X, BOX_Y, BOX_WIDTH, BOX_HEIGHT);
   }

```

RectangleComponent.java (cont.)

```
    public void paintComponent(Graphics g)
    {
       Graphics2D g2 = (Graphics2D) g;

       g2.draw(box);
    }

    /**
       Moves the rectangle to the given location.
       @param x the x-position of the new location
       @param y the y-position of the new location
    */
    public void moveTo(int x, int y)
    {
       box.setLocation(x, y);
       repaint();
    }
}
```

Mouse Events

- Call `repaint` when you modify the shapes that `paintComponent` draws:

```
box.setLocation(x, y);
repaint();
```

Mouse Events

- Mouse listener: if the mouse is pressed, `listener` moves the rectangle to the mouse location:

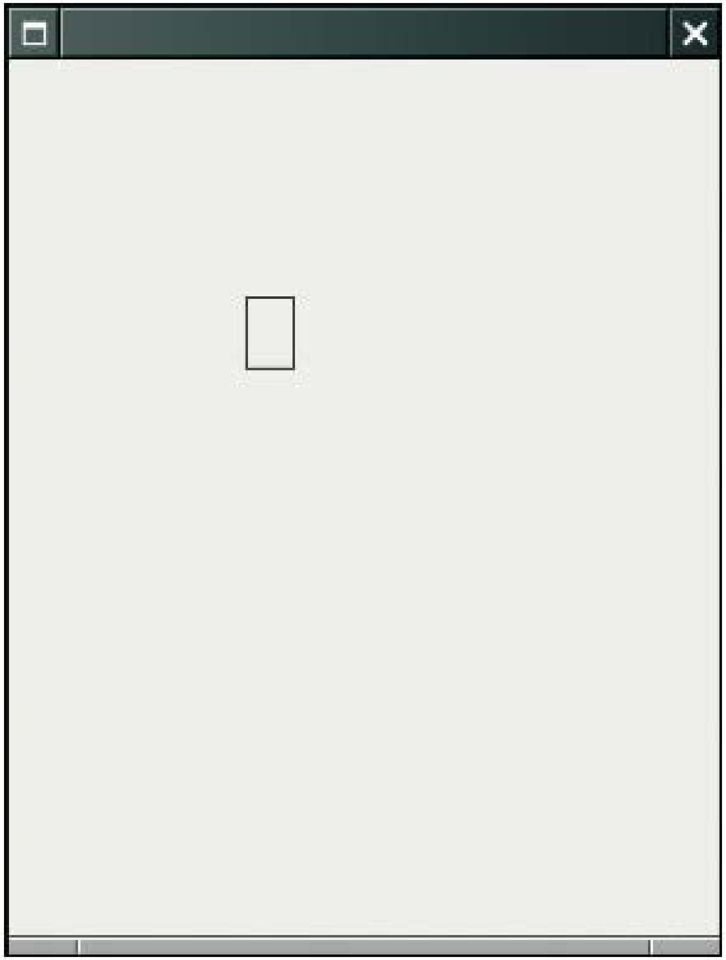
```
class MousePressListener implements MouseListener
{
   public void mousePressed(MouseEvent event)
   {
      int x = event.getX();
      int y = event.getY();
      component.moveTo(x, y);
   }
   // Do-nothing methods
   public void mouseReleased(MouseEvent event) {}
   public void mouseClicked(MouseEvent event) {}
   public void mouseEntered(MouseEvent event) {}
   public void mouseExited(MouseEvent event) {}
}
```

- All five methods of the interface must be implemented; unused methods can be empty

Mouse Events

`RectangleComponentViewer`

Program Run: *Clicking the mouse moves the rectangle*



RectangleComponentViewer.java

```
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;

/**
   This program displays a RectangleComponent.
*/
public class RectangleComponentViewer
{
   private static final int FRAME_WIDTH = 300;
   private static final int FRAME_HEIGHT = 400;

   public static void main(String[] args)
   {
      final RectangleComponent component = new RectangleComponent();

```

RectangleComponentViewer.java (cont.)

```
      // Add mouse press listener

      class MousePressListener implements MouseListener
      {
         public void mousePressed(MouseEvent event)
         {
            int x = event.getX();
            int y = event.getY();
            component.moveTo(x, y);
         }

         // Do-nothing methods
         public void mouseReleased(MouseEvent event) {}
         public void mouseClicked(MouseEvent event) {}
         public void mouseEntered(MouseEvent event) {}
         public void mouseExited(MouseEvent event) {}
      }

      MouseListener listener = new MousePressListener();
      component.addMouseListener(listener);

      JFrame frame = new JFrame();
      frame.add(component);

      frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

Self Check

Why was the `moveBy` method in the `RectangleComponent` replaced with a `moveTo` method?

Self Check

Why must the `MousePressListener` class supply five methods?

AGENDA

- Arrays & Array Lists
- Interfaces & Polymorphism
- Inheritance
- Input / Output & Exception Handling

AGENDA

- Arrays & Array Lists
- Interfaces & Polymorphism
- Inheritance
- Input / Output & Exception Handling

BASIC CONCEPTS OF JAVA 3

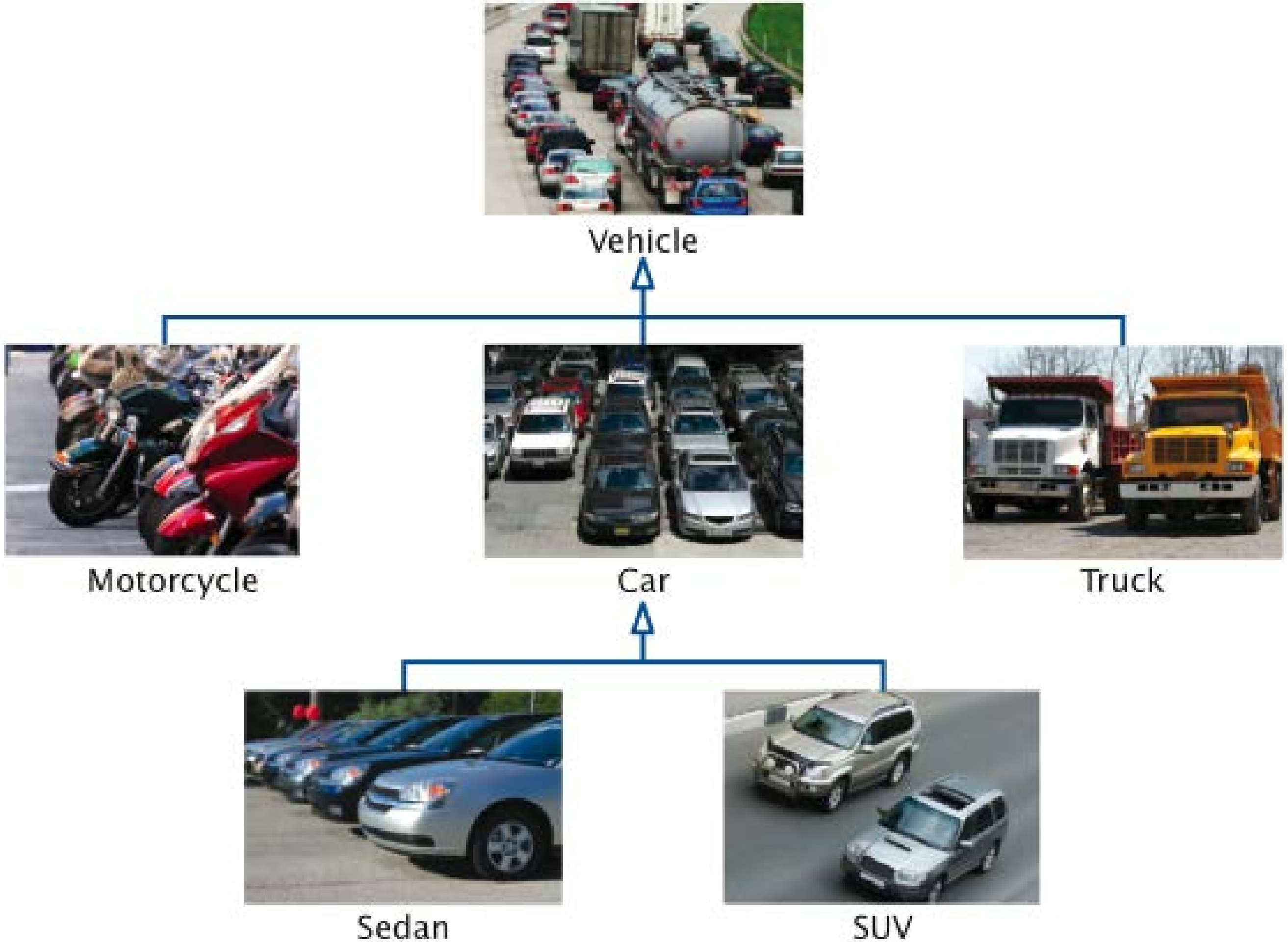
INHERITANCE

Chapter Goals

- To learn about inheritance
- To understand how to inherit and override superclass methods
- To be able to invoke superclass constructors
- To learn about `protected` and package access control
- To understand the common superclass `Object` and to override its `toString` and `equals` methods
- To use inheritance for customizing user interfaces

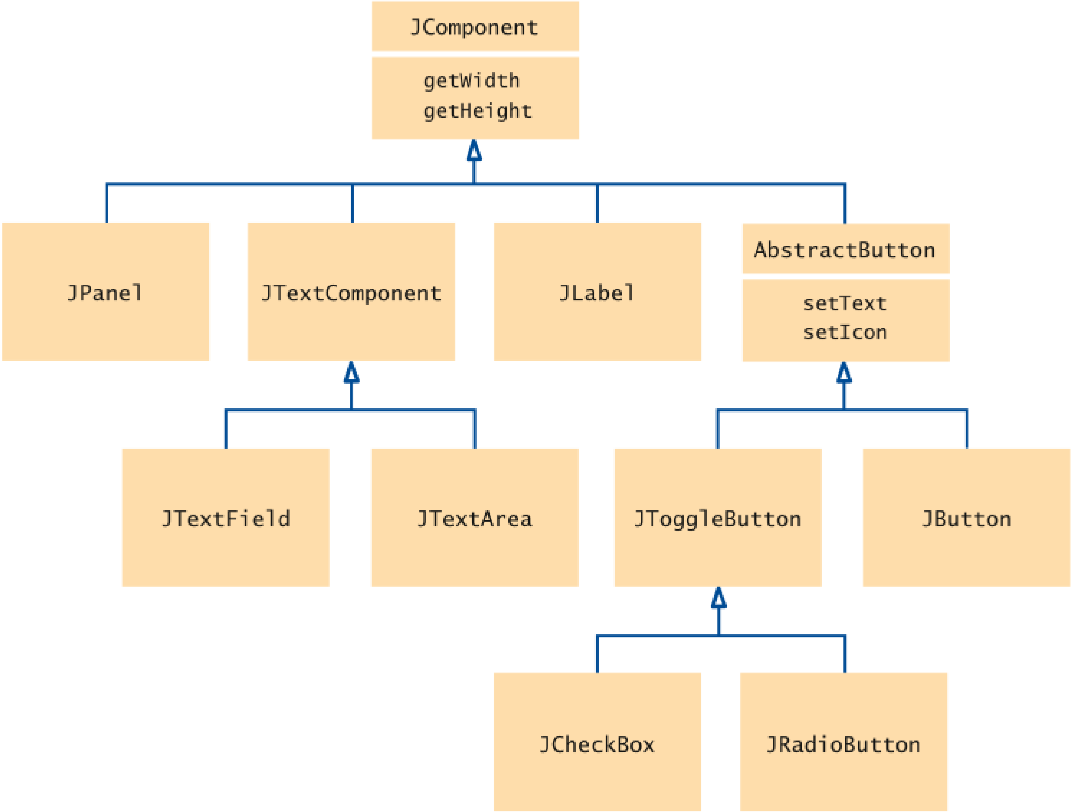
Inheritance Hierarchies

- Often categorize concepts into *hierarchies*:



Inheritance Hierarchies

- Set of classes can form an *inheritance hierarchy*
  - *Classes representing the most general concepts are near the root, more specialized classes towards the branches*
  - *Example: A part of the hierarchy of Swing User Interface components*



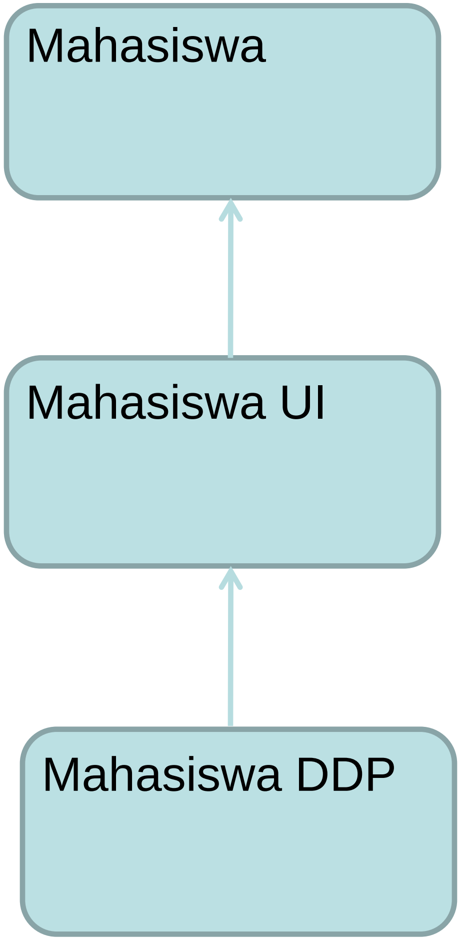
Inheritance Hierarchies

- **Superclass:** more general class
- **Subclass:** more specialized class that inherits from the superclass
  - *Example:*

  `JPanel` is a subclass of `Jcomponent`
  `JComponent is a superclass of JPanel`
  `JTextComponent` is superclass of `JTextArea` and subclass of `Jcomponent`
- Can inherit from only one superclass in Java
- Subclass inherits all public capabilities of its superclass
- Subclasse specializes its superclass: override, implement new specific methods

Inheritance

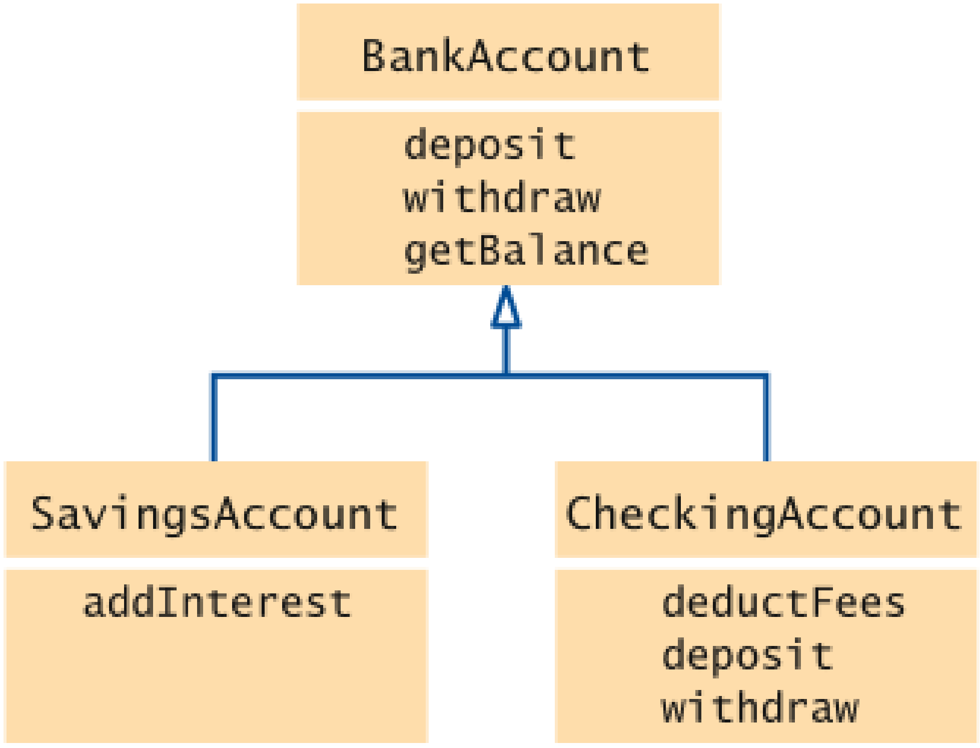
- **Mahasiswa has one capability:**
  - study(): going home, opening a book, read 1 chapter
- **Mahasiswa UI:**
  - study(): going to the library, opening a book, read 1 chapter
- **Mahasiswa DDP:**
  - codingLikeCrazy()



Inheritance Hierarchies

- **Example:** Different account types:
    1. *Checking account:*
        - *No interest*
        - *Small number of free transactions per month*
        - *Charges transaction fee for additional transactions*
    2. *Savings account:*
        - *Earns interest that compounds monthly*
- Superclass: `BankAccount`
- Subclasses: `CheckingAccount` & `SavingsAccount`
- Behavior of account classes:
    - *All support `getBalance` method*
    - *Also support `deposit` and `withdraw` methods, but implementation details differ*
    - *Checking account needs a method `deductFees` to deduct the monthly fees and to reset the transaction counter*
    - *Checking account must override `deposit` and `withdraw` methods to count the transactions*

Inheritance Hierarchies



Self Check

What is the purpose of the `JTextComponent` class?

Self Check

Why don’t we place the `addInterest` method in the `BankAccount` class?

Inheritance Hierarchies

- Inheritance is a mechanism for extending existing classes by adding instance variables and methods:

```
class SavingsAccount extends BankAccount
{
    added instance variables
    new methods
}
```

- A subclass inherits the methods of its superclass:

```
SavingsAccount collegeFund = new SavingsAccount(10);
// Savings account with 10% interest
collegeFund.deposit(500);
// OK to use BankAccount method with SavingsAccount object
```

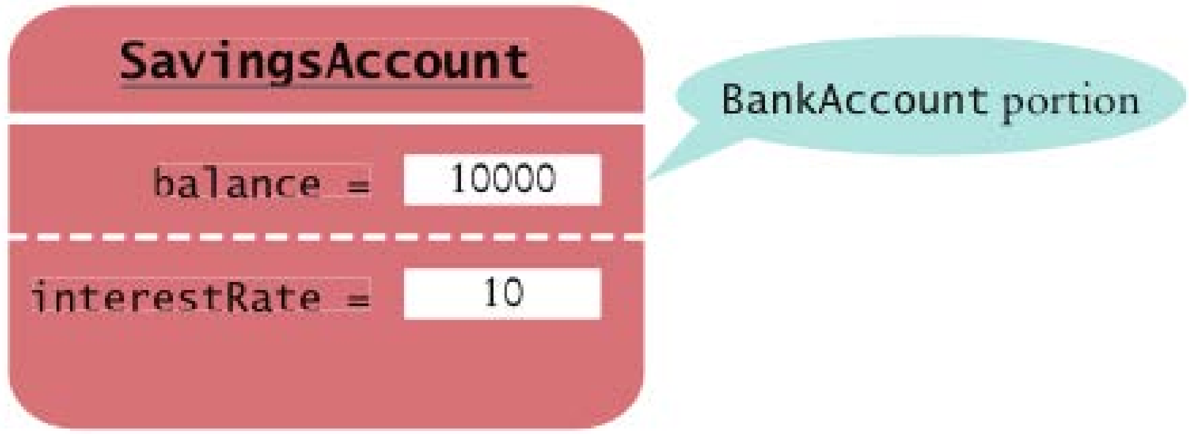
Inheritance Hierarchies

- In subclass, specify added instance variables, added methods, and changed or overridden methods:

```
public class SavingsAccount extends BankAccount
{
   private double interestRate;
   public SavingsAccount(double rate)
   {
      Constructor implementation
   }
   public void addInterest()
   {
      Method implementation
   }
}
```

Inheritance Hierarchies

- Instance variables declared in the superclass are present in subclass objects
- `SavingsAccount` object inherits the balance instance variable from `BankAccount`, and gains one additional instance variable, `interestRate`
- Layout of a subclass object:



Inheritance Hierarchies

- Implement the new `addInterest` method:

```
public class SavingsAccount extends BankAccount
{
   private double interestRate;
   public SavingsAccount(double rate)
   {
      interestRate = rate;
   }
   public void addInterest()
   {
      double interest = getBalance() * interestRate / 100;
      deposit(interest);
   }
}
```

Inheritance Hierarchies

- A subclass has no access to private instance variables of its superclass
- **Encapsulation:** `addInterest` calls `getBalance` rather than updating the `balance` variable of the superclass (variable is `private`)
- Note that `addInterest` calls `getBalance` without specifying an implicit parameter (the calls apply to the same object)
- Inheriting from a class differs from implementing an interface: the subclass inherits behavior from the superclass

SavingsAccount.java

```
/**
   An account that earns interest at a fixed rate.
*/
public class SavingsAccount extends BankAccount
{
   private double interestRate;

   /**
      Constructs a bank account with a given interest rate.
      @param rate the interest rate
   */
   public SavingsAccount(double rate)
   {
      interestRate = rate;
   }

   /**
      Adds the earned interest to the account balance.
   */
   public void addInterest()
   {
      double interest = getBalance() * interestRate / 100;
      deposit(interest);
   }
}
```

Syntax Inheritance

Syntax

```
class SubclassName extends SuperclassName
{
    instance variables
    methods
}
```

Example

Subclass

Superclass

```
public class SavingsAccount extends BankAccount
{
    private double interestRate;
    . . .

    public void addInterest()
    {
        double interest = getBalance() * interestRate / 100;
        deposit(interest);
    }
}
```

Declare instance variables that are added to the subclass.

Declare methods that are specific to the subclass.

The reserved word extends denotes inheritance.

Self Check

Which instance variables does an object of class `SavingsAccount` have?

Self Check

Name four methods that you can apply to `SavingsAccount` objects.

Self Check

If the class `Manager` extends the class `Employee`, which class is the superclass and which is the subclass?

Common Error: Shadowing Instance Variables

- A subclass has no access to the private instance variables of the superclass:

```
public class SavingsAccount extends BankAccount
{
   public void addInterest()
   {
      double interest = getBalance() * interestRate / 100;
      balance = balance + interest; // Error
   }
   . . .
}
```

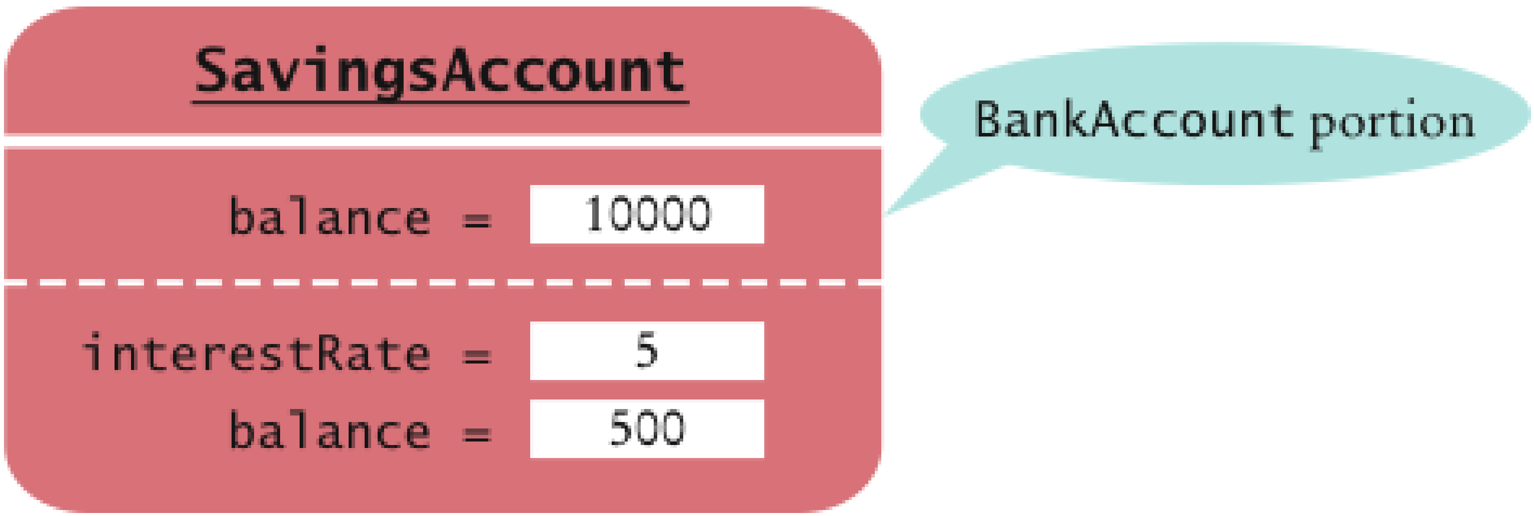
Common Error: Shadowing Instance Variables

- Beginner's error: "solve" this problem by adding another instance variable with same name:

```
public class SavingsAccount extends BankAccount
{
   private double balance; // Don't
   public void addInterest()
   {
      double interest = getBalance() * interestRate / 100;
      balance = balance + interest; // Compiles but doesn't
         // update the correct balance
   }
   . . .
}
```

Common Error: Shadowing Instance Variables

- Now the addInterest method compiles, but it doesn't update the correct balance!



Overriding Methods

- A subclass method **overrides** a superclass method if it has the same name and parameter types as a superclass method
  - *When such a method is applied to a subclass object, the overriding method is executed*

Overriding Methods

- Example: `deposit` and `withdraw` methods of the `CheckingAccount` class override the `deposit` and `withdraw` methods of the `BankAccount` class to handle transaction fees:

```
public class BankAccount
{
   . . .
   public void deposit(double amount) { . . . }
   public void withdraw(double amount) { . . . }
   public double getBalance() { . . . }
}
public class CheckingAccount extends BankAccount
{
   . . .
   public void deposit(double amount) { . . . }
   public void withdraw(double amount) { . . . }
   public void deductFees() { . . . }
}
```

Overriding Methods

- Problem: Overriding method `deposit` can't simply add `amount` to `balance`:

```
public class CheckingAccount extends BankAccount
{
   . . .
   public void deposit(double amount)
   {
      transactionCount++;
      // Now add amount to balance
      balance = balance + amount; // Error
   }
}
```

- If you want to modify a private superclass instance variable, you must use a public method of the superclass
- `deposit` method of `CheckingAccount` must invoke the `deposit` method of `BankAccount`

Overriding Methods

- Idea:

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount)
   {
      transactionCount++;
      // Now add amount to balance
      deposit; // Not complete
   }
}
```

- Won't work because compiler interprets

  ```
  deposit(amount);
  ```

  as

  ```
  this.deposit(amount);
  ```

  which calls the method we are currently writing ⇒ infinite recursion

Overriding Methods

- Use the `super` reserved word to call a method of the superclass:

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount)
   {
      transactionCount++;
      // Now add amount to balance
      super.deposit
   }
}
```

Overriding Methods

- Remaining methods of `CheckingAccount` also invoke a superclass method:

```
public class CheckingAccount extends BankAccount
{
   private static final int FREE_TRANSACTIONS = 3;
   private static final double TRANSACTION_FEE = 2.0;
   private int transactionCount;
   . . .
   public void withdraw(double amount
   {
      transactionCount++;
      // Now subtract amount from balance
      super.withdraw(amount);
   }
   public void deductFees()
   {
      if (transactionCount > FREE_TRANSACTIONS)
      {
         double fees = TRANSACTION_FEE *
            (transactionCount - FREE_TRANSACTIONS);
         super.withdraw(fees);
      }
      transactionCount = 0;
   }
   . . .
}
```

Syntax Calling a Superclass Method

Syntax `super.`*methodName*`(`*parameters*`);`

Example

```
public void deposit(double amount)
{
   transactionCount++;
   super.deposit(amount);
}
```

Calls the method of the superclass instead of the method of the current class.

If you omit `super`, this method calls itself.

Self Check

Categorize the methods of the `SavingsAccount` class as inherited, new, and overridden.

Answer: The `SavingsAccount` class inherits the `deposit`, `withdraw`, and `getBalance` methods. The `addInterest` method is new. No methods override superclass methods.

Self Check

Why does the `withdraw` method of the `CheckingAccount` class call `super.withdraw`?

Self Check

Why does the `deductFees` method set the transaction count to zero?

Subclass Construction

- To call the superclass constructor, use the `super` reserved word in the first statement of the subclass constructor:

```
public class CheckingAccount extends BankAccount
{
   public CheckingAccount(double initialBalance)
   {
      // Construct superclass
      super(initialBalance);
      // Initialize transaction count
      transactionCount = 0;
   }
   ...
}
```

- When subclass constructor doesn't call superclass constructor, the superclass must have a constructor with no parameters
  - *If, however, all constructors of the superclass require parameters, then the compiler reports an error*

CheckingAccount.java

```
/**
   A checking account that charges transaction fees.
*/
public class CheckingAccount extends BankAccount
{
   private static final int FREE_TRANSACTIONS = 3;
   private static final double TRANSACTION_FEE = 2.0;

   private int transactionCount;

   /**
      Constructs a checking account with a given balance.
      @param initialBalance the initial balance
   */
   public CheckingAccount(double initialBalance)
   {
      // Construct superclass
      super(initialBalance);

      // Initialize transaction count
      transactionCount = 0;
   }

```

CheckingAccount.java (cont.)

```
    public void deposit(double amount)
    {
        transactionCount++;
        // Now add amount to balance
        super.deposit(amount);
    }

    public void withdraw(double amount)
    {
        transactionCount++;
        // Now subtract amount from balance
        super.withdraw(amount);
    }

    /**
        Deducts the accumulated fees and resets the
        transaction count.
    */
    public void deductFees()
    {
        if (transactionCount > FREE_TRANSACTIONS)
        {
            double fees = TRANSACTION_FEE *
                    (transactionCount - FREE_TRANSACTIONS);
            super.withdraw(fees);
        }
        transactionCount = 0;
    }
}
```

Syntax Calling a Superclass Constructor

Syntax

```
accessSpecifier ClassName(parameterType parameterName, . . .)
{
   super(parameters);
   . . .
}
```

Example

```
public CheckingAccount(double initialBalance)
{
   super(initialBalance);
   transactionCount = 0;
}
```

Invokes the constructor of the superclass.

Must be the first statement of the subclass constructor.

Subclass constructor

If not present, the superclass is constructed with its default constructor.

Self Check

Why didn't the `SavingsAccount` constructor call its superclass constructor?

Self Check

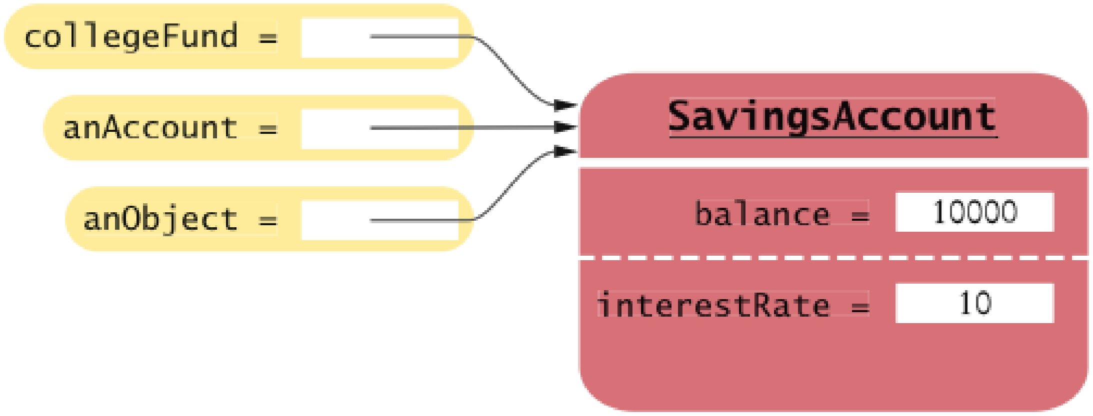
When you invoke a superclass method with the `super` keyword, does the call have to be the first statement of the subclass method?

Converting Between Subclass and Superclass Types

- OK to convert subclass reference to superclass reference:

```
SavingsAccount collegeFund = new SavingsAccount(10);
BankAccount anAccount = collegeFund;
Object anObject = collegeFund;
```

- The three object references stored in `collegeFund`, `anAccount`, and `anObject` all refer to the same object of type `SavingsAccount`
- Variables of different types can refer to the same object:



Converting Between Subclass and Superclass Types

- Superclass references don't know the full story:

```
anAccount.deposit(1000); // OK
anAccount.addInterest();
// No--not a method of the class to which anAccount
// belongs
```

- Why would anyone want to know *less* about an object?
  - *Reuse code that knows about the superclass but not the subclass:*

```
public void transfer(double amount, BankAccount other)
{
   withdraw(amount);
   other.deposit(amount);
}
```

  - *Can be used to transfer money from any type of* `BankAccount`

Converting Between Subclass and Superclass Types

- Occasionally you need to convert from a superclass reference to a subclass reference:

```
BankAccount anAccount = (BankAccount) anObject;
```

- This cast is dangerous: If you are wrong, an exception is thrown
- Solution: Use the `instanceof` operator
- `instanceof`: Tests whether an object belongs to a particular type:

```
if (anObject instanceof BankAccount)
{
   BankAccount anAccount = (BankAccount) anObject;
   ...
}
```

Syntax instanceof

Syntax *object* instanceof *TypeName*

Example

If anObject is null, instanceof returns false.

Returns true if anObject can be cast to a BankAccount.

The object may belong to a subclass of BankAccount.

```
if (anObject instanceof BankAccount)
{
   BankAccount anAccount = (BankAccount) anObject;
   . . .
}
```

You can invoke BankAccount methods on this variable.

Two references to the same object.

Self Check

Why did the second parameter of the `transfer` method have to be of type `BankAccount` and not, for example, `SavingsAccount`?

Self Check

Why can't we change the second parameter of the `transfer` method to the type `Object`?

Polymorphism and Inheritance

- Type of a variable doesn’t completely determine type of object to which it refers:

```
BankAccount aBankAccount = new SavingsAccount(1000);
// aBankAccount holds a reference to a SavingsAccount
```

- `BankAccount anAccount = new CheckingAccount();`

```
anAccount.deposit(1000);
```

  Which deposit method is called?

- *Dynamic method lookup:* When the virtual machine calls an instance method, it locates the method of the implicit parameter's class

Polymorphism and Inheritance

- Example:

```
public void transfer(double amount, BankAccount other)
{
   withdraw(amount);
   other.deposit(amount);
}
```

- When you call

```
anAccount.transfer(1000, anotherAccount);
```

two method calls result:

```
anAccount.withdraw(1000);
anotherAccount.deposit(1000);
```

Polymorphism and Inheritance

- *Polymorphism:* Ability to treat objects with differences in behavior in a uniform way
- The first method call

  ```
  withdraw(amount);
  ```

  is a shortcut for

  ```
  this.withdraw(amount);
  ```
- `this` can refer to a `BankAccount` or a subclass object

AccountTester.java

```
/**
    This program tests the BankAccount class and
    its subclasses.
*/
public class AccountTester
{
   public static void main(String[] args)
   {
      SavingsAccount momsSavings = new SavingsAccount(0.5);

      CheckingAccount harrysChecking = new CheckingAccount(100);

      momsSavings.deposit(10000);

      momsSavings.transfer(2000, harrysChecking);
      harrysChecking.withdraw(1500);
      harrysChecking.withdraw(80);

      momsSavings.transfer(1000, harrysChecking);
      harrysChecking.withdraw(400);

      // Simulate end of month
      momsSavings.addInterest();
      harrysChecking.deductFees();

      System.out.println("Mom's savings balance: "
            + momsSavings.getBalance());
      System.out.println("Expected: 7035");

      System.out.println("Harry's checking balance: "
            + harrysChecking.getBalance());
      System.out.println("Expected: 1116");
   }
}
```

Program Run:

```
Mom's savings balance: 7035.0
Expected: 7035
Harry's checking balance: 1116.0
Expected: 1116
```

Self Check

If `a` is a variable of type `BankAccount` that holds a non-`null` reference, what do you know about the object to which `a` refers?

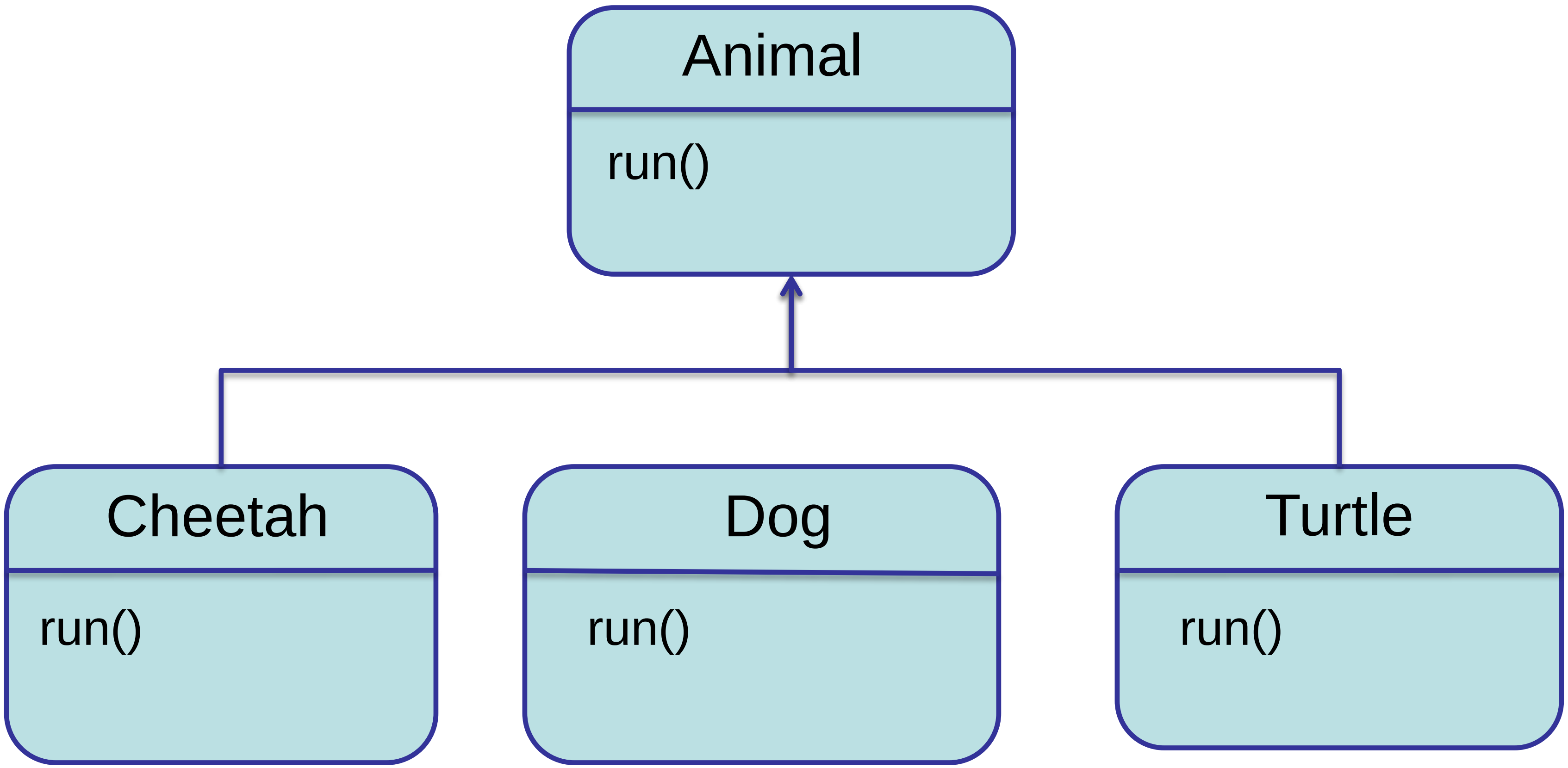
Self Check

If a refers to a checking account, what is the effect of calling `a.transfer(1000, a)`?

Abstract Class

- Sometimes, it is desirable to force programmers to override a method.
- Declare a method as ***abstract.***
- An abstract method has no implementation.
- A class that declares an abstract method, or that inherits an abstract method without overriding it MUST be declared as abstract.
- Abstract class cannot be instantiated.
- Differences with interface?

Abstract Class



Protected Access

- Protected features can be accessed by all subclasses and by all classes in the same package
- Solves the problem that `CheckingAccount` methods need access to the `balance` instance variable of the superclass `BankAccount`:

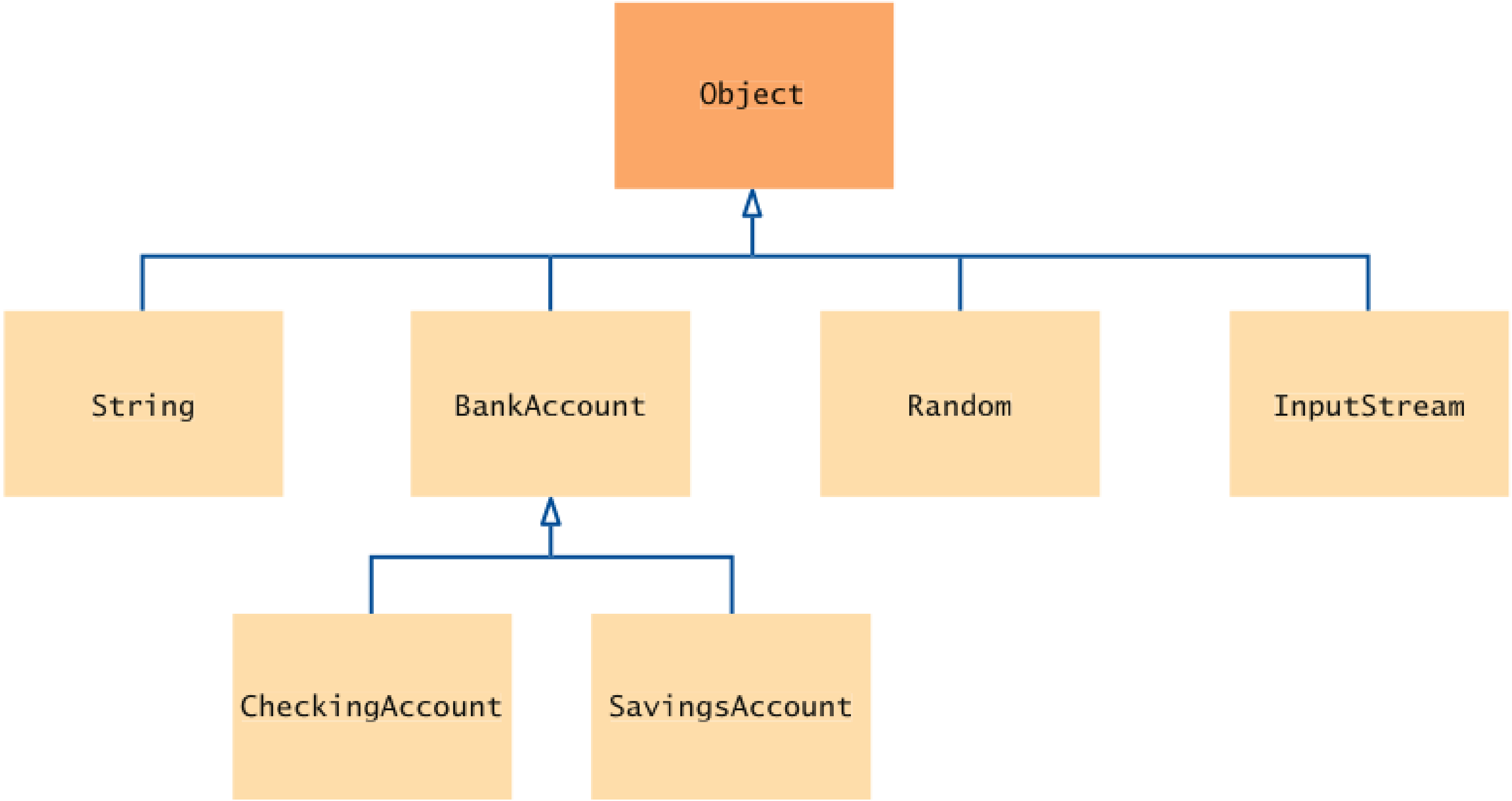
```
public class BankAccount
{
   . . .
   protected double balance;
}
```

Protected Access

- The designer of the superclass has no control over the authors of subclasses:
  - *Any of the subclass methods can corrupt the superclass data*
  - *Classes with protected instance variables are hard to modify - the protected variables cannot be changed, because someone somewhere out there might have written a subclass whose code depends on them*
- Protected data can be accessed by all methods of classes in the same package
- It is best to leave all data private and provide accessor methods for the data

`Object`: *The Cosmic Superclass*

- All classes defined without an explicit `extends` clause automatically extend `Object`
- The object class is the superclass of every java class:



`Object`*: The Cosmic Superclass*

- Most useful methods:
  - *`String toString()`*
  - *`boolean equals(Object otherObject)`*
  - *`Object clone()`*
- Good idea to override these methods in your classes

Overriding the `toString` *Method*

- Returns a string representation of the object
- Useful for debugging:

```
Rectangle box = new Rectangle(5, 10, 20, 30);
String s = box.toString();
// Sets s to "java.awt.Rectangle[x=5,y=10,width=20,
// height=30]"
```

- `toString` is called whenever you concatenate a string with an object:

```
"box=" + box;
// Result: "box=java.awt.Rectangle[x=5,y=10,width=20,
// height=30]"
```

- `Object.toString` prints class name and the *hash code* of the object:

```
BankAccount momsSavings = new BankAccount(5000);
String s = momsSavings.toString();
// Sets s to something like "BankAccount@d24606bf"
```

Overriding the `toString` *Method*

- To provide a nicer representation of an object, override `toString`:

```
public String toString()
{
   return "BankAccount[balance=" + balance + "]";
}
```

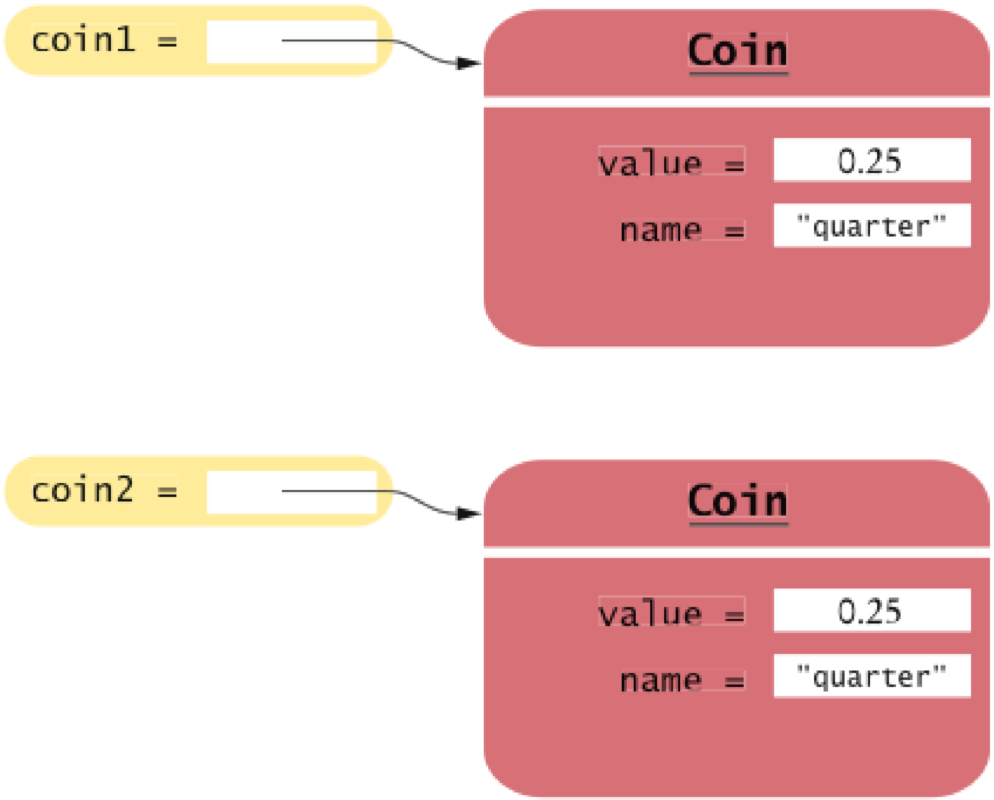
- This works better:

```
BankAccount momsSavings = new BankAccount(5000);
String s = momsSavings.toString();
// Sets s to "BankAccount[balance=5000]"
```

Overriding the `equals` *Method*

- `equals` tests for same *contents* (2 references to equal objects):

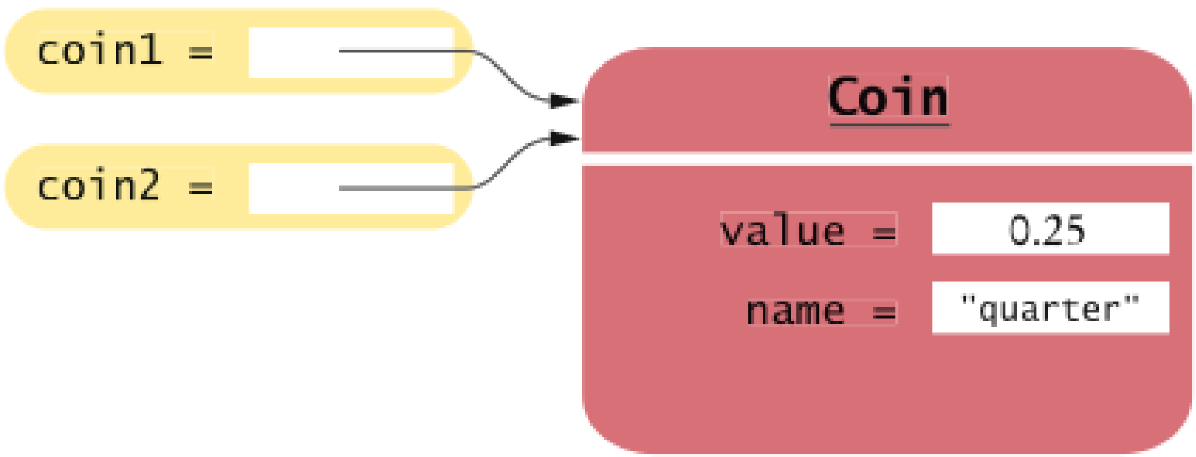
```
if (coin1.equals(coin2)) . . .
// Contents are the same
```



Overriding the `equals` *Method*

- == tests for references to the same object:

```
if (coin1 == (coin2)) . . .
// Objects are the same
```



Overriding the **`equals`** *Method*

- Need to override the `equals` method of the `Object` class:

```
public class Coin
{
   ...
   public boolean equals(Object otherObject)
   {
      ...
   }
   ...
}
```

Overriding the **equals** *Method*

- Cannot change parameter type; use a *cast* instead:

```
public class Coin
{
   ...
   public boolean equals(Object otherObject)
   {
      Coin other = (Coin) otherObject;
      return name.equals(other.name) && value ==
         other.value;
   }
   ...
}
```

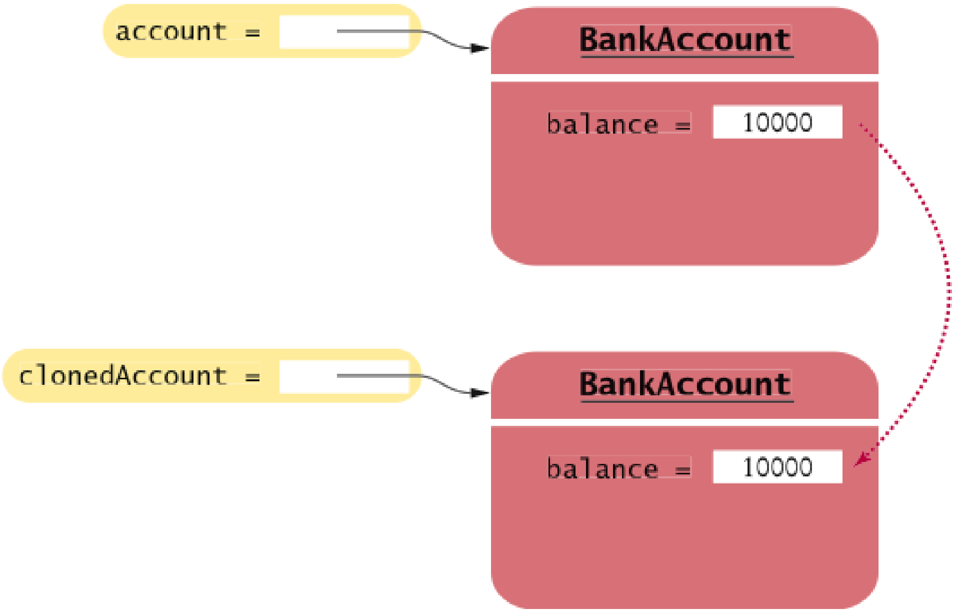
- You should also override the `hashCode` method so that equal objects have the same hash code

The `clone` *Method*

- Copying an object reference gives two references to same object:

```
BankAccount account = newBankAccount(1000);
BankAccount account2 = account;
account2.deposit(500); // Now both account and account2
// refer to a bank account with a balance of 1500
```

- Sometimes, need to make a copy of the object (clone):



The `clone` *Method*

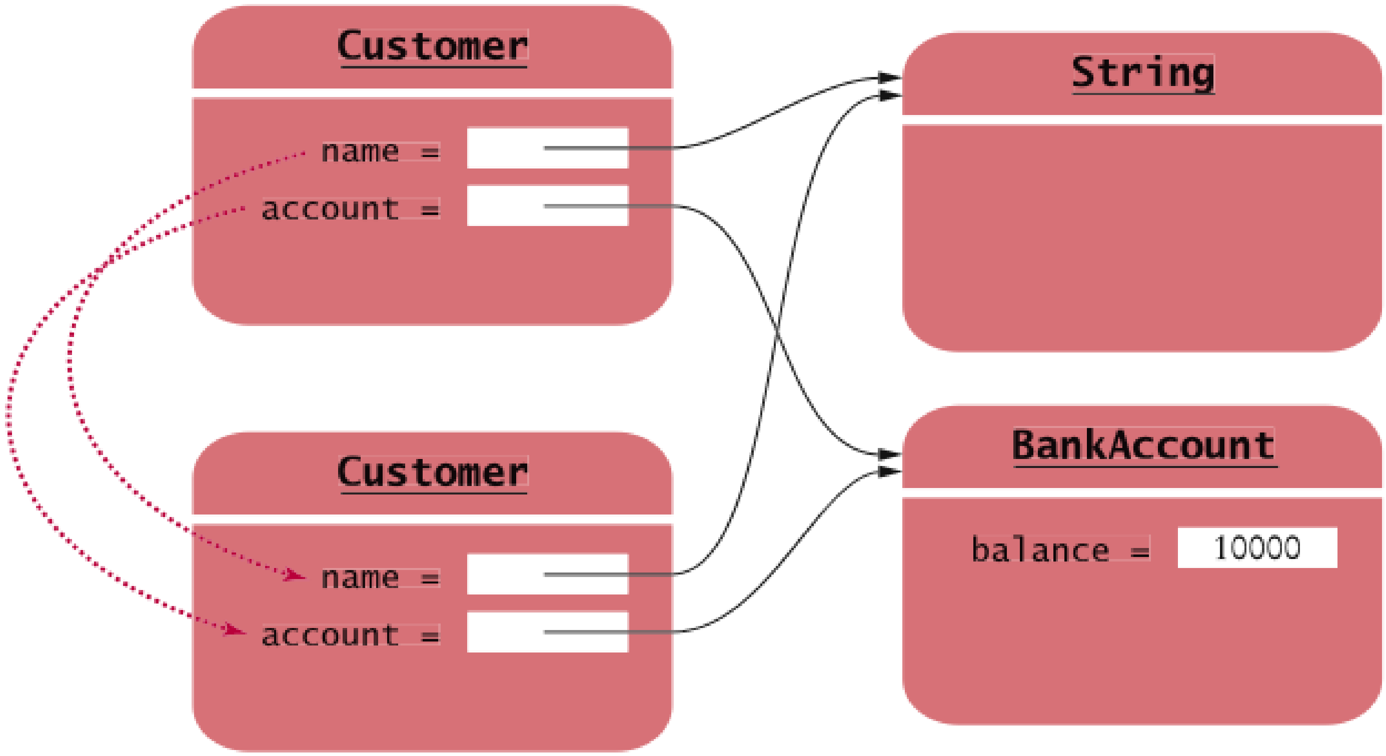
- Implement `clone` method to make a new object with the same state as an existing object
- Use `clone`:

```
BankAccount clonedAccount = (BankAccount) account.clone();
```

- Must cast return value because return type is `Object`

The `Object.clone` *Method*

- Creates *shallow copies*:



The `Object.clone` *Method*

- Does not systematically clone all subobjects
- Must be used with caution
- It is declared as `protected`; prevents from accidentally calling `x.clone()` if the class to which `x` belongs hasn't redefined `clone` to be `public`
- You should override the `clone` method with care

Self Check

Should the call `x.equals(x)` always return `true`?

Self Check

Can you implement `equals` in terms of `toString`? Should you?

Answer: If `toString` returns a string that describes all instance variables, you can simply call `toString` on the implicit and explicit parameters, and compare the results. However, comparing the variables is more efficient than converting them into strings.

Using Inheritance to Customize Frames

- Use inheritance for complex frames to make programs easier to understand
- Design a subclass of `JFrame`
- Store the components as instance variables
- Initialize them in the constructor of your subclass
- If initialization code gets complex, simply add some helper methods

InvestmentFrame.java

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class InvestmentFrame extends JFrame
{
   private JButton button;
   private JLabel label;
   private JPanel panel;
   private BankAccount account;

   private static final int FRAME_WIDTH = 400;
   private static final int FRAME_HEIGHT = 100;

   private static final double INTEREST_RATE = 10;
   private static final double INITIAL_BALANCE = 1000;

   public InvestmentFrame()
   {
      account = new BankAccount(INITIAL_BALANCE);

      // Use instance variables for components
      label = new JLabel("balance: " + account.getBalance());

      // Use helper methods
      createButton();
      createPanel();

      setSize(FRAME_WIDTH, FRAME_HEIGHT);
   }

```

InvestmentFrame.java

```java
    private void createButton()
    {
       button = new JButton("Add Interest");
       ActionListener listener = new AddInterestListener();
       button.addActionListener(listener);
    }

    private void createPanel()
    {
       panel = new JPanel();
       panel.add(button);
       panel.add(label);
       add(panel);
    }

    class AddInterestListener implements ActionListener
    {
       public void actionPerformed(ActionEvent event)
       {
          double interest = account.getBalance() * INTEREST_RATE / 100;
          account.deposit(interest);
          label.setText("balance: " + account.getBalance());
       }
    }
}
```

Example: Investment Viewer Program

Of course, we still need a class with a `main` method:

```
import javax.swing.JFrame;

/**
    This program displays the growth of an investment.
*/
public class InvestmentViewer2
{
   public static void main(String[] args)
   {
      JFrame frame = new InvestmentFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

Self Check

How many Java source files are required by the investment viewer application when we use inheritance to define the frame class?

Answer: Three: `InvestmentFrameViewer`, `InvestmentFrame`, and `BankAccount`.

Self Check

Why does the `InvestmentFrame` constructor call `setSize(FRAME_WIDTH, FRAME_HEIGHT)`, whereas the `main` method of the investment viewer class called `frame.setSize(FRAME_WIDTH, FRAME_HEIGHT)`?

AGENDA

- Arrays & Array Lists
- Interfaces & Polymorphism
- Inheritance
- Input / Output & Exception Handling

AGENDA

- Arrays & Array Lists
- Interfaces & Polymorphism
- Inheritance
- Input / Output & Exception Handling

BASIC CONCEPTS OF JAVA 3

INPUT / OUTPUT & EXCEPTION HANDLING

Chapter Goals

- To be able to read and write text files
- To learn how to throw exceptions
- To be able to design your own exception classes
- To understand the difference between checked and unchecked exceptions
- To know when and where to catch an exception

Reading Text Files

- Simplest way to read text: Use `Scanner` class
- To read from a disk file, construct a `File`
- Then, use the `File` to construct a `Scanner` object

```
File reader = new File ("input.txt");
Scanner in = new Scanner(reader);
```

- Use the `Scanner` methods to read data from file
  - `next`, `nextLine`, `nextInt`, *and* `nextDouble`

Writing Text Files

- To write to a file, construct a `PrintWriter` object:

```
PrintWriter out = new PrintWriter("output.txt");
```

- If file already exists, it is emptied before the new data are written into it
- If file doesn't exist, an empty file is created
- Use `print` and `println` to write into a `PrintWriter`:

```
out.println(29.95);
out.println(new Rectangle(5, 10, 15, 25));
out.println("Hello, World!");
```

- You must close a file when you are done processing it:

```
out.close();
```

Otherwise, not all of the output may be written to the disk file

Class `FileNotFoundException`

- When the input or output file doesn't exist, a `FileNotFoundException` can occur
- To handle the exception, label the main method like this:

```
public static void main(String[] args) throws
   FileNotFoundException
```

A Sample Program

- Reads all lines of a file and sends them to the output file, preceded by line numbers
- Sample input file:

```
Mary had a little lamb
Whose fleece was white as snow.
And everywhere that Mary went,
The lamb was sure to go!
```

- Program produces the output file:

```
/* 1 */ Mary had a little lamb
/* 2 */ Whose fleece was white as snow.
/* 3 */ And everywhere that Mary went,
/* 4 */ The lamb was sure to go!
```

- Program can be used for numbering Java source files

LineNumberer.java

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

/**
   This program applies line numbers to a file.
*/
public class LineNumberer
{
   public static void main(String[] args) throws FileNotFoundException
   {
      // Prompt for the input and output file names

      Scanner console = new Scanner(System.in);
      System.out.print("Input file: ");
      String inputFileName = console.next();
      System.out.print("Output file: ");
      String outputFileName = console.next();

```

LineNumberer.java (cont.)

```
        // Construct the Scanner and PrintWriter objects for reading and writing

        File inputFile = new File(inputFileName);
        Scanner in = new Scanner(inputFile);
        PrintWriter out = new PrintWriter(outputFileName);
        int lineNumber = 1;

        // Read the input and write the output

        while (in.hasNextLine())
        {
            String line = in.nextLine();
            out.println("/* " + lineNumber + " */ " + line);
            lineNumber++;
        }

        in.close();
        out.close();
    }
}
```

Self Check

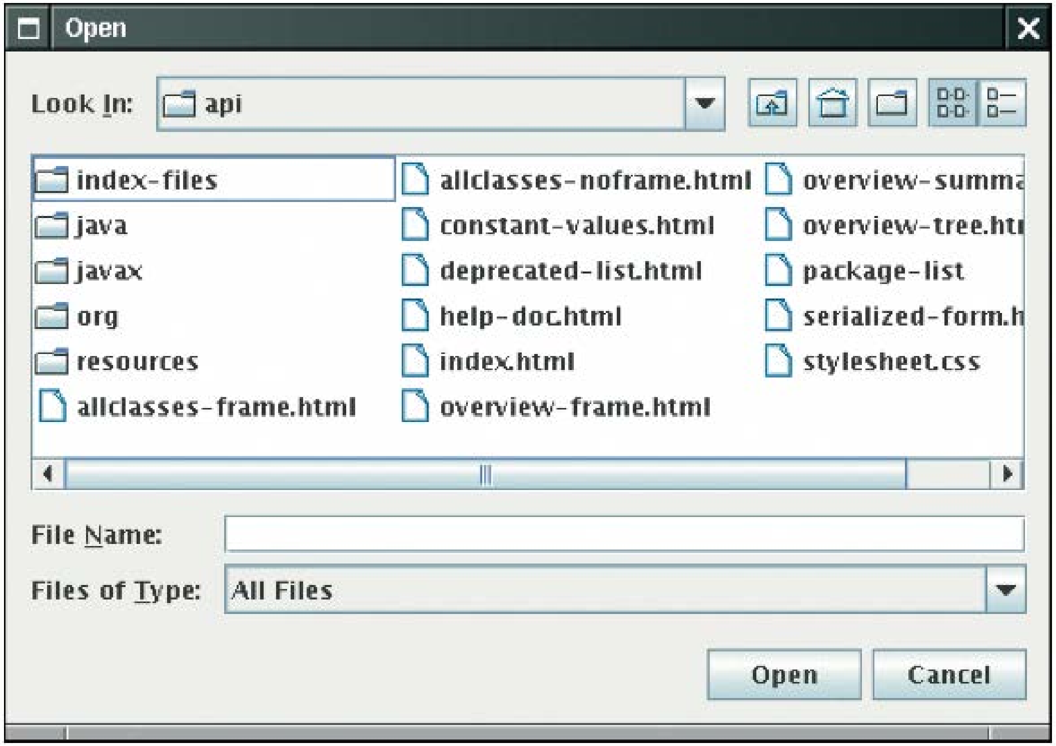
What happens when you supply the same name for the input and output files to the `LineNumberer` program?

Self Check

What happens when you supply the name of a nonexistent input file to the `LineNumberer` program?

File Dialog Boxes

```
JFileChooser chooser = new JFileChooser();
FileReader in = null;
if (chooser.showOpenDialog(null) ==
   JFileChooser.APPROVE_OPTION)
{
   File selectedFile = chooser.getSelectedFile();
   reader = new FileReader(selectedFile);
   ...
}
```



Reading Text Input: Reading Words

- The `next` method reads a word at a time:

```
while (in.hasNext())
{
   String input = in.next();
   System.out.println(input);
}
```

- With our sample input, the output is:

```
Mary
had
a
little
lamb
…
```

- A *word* is any sequence of characters that is not white space

Reading Text Input: Reading Words

- To specify a pattern for word boundaries, call
  `Scanner.useDelimiter`
- Example: discard anything that isn't a letter:

```
Scanner in = new Scanner(. . .);
in.useDelimiter("[^A-Za-z]+");
...
```

- The notation used for describing the character pattern is called a *regular expression*

Reading Text Input: Processing Lines

- The `nextLine` method reads a line of input and consumes the newline character at the end of the line:

```
String line = in.nextLine();
```

- Example: process a file with phone number data like this:

```
Surya  727283
Egidius  865891
Ashar 898790
Khairia Rais 898769
```

- First read each input line into a string

Reading Text Input: Processing Lines

- Then use the `isDigit` and `isWhitespace` methods to find out where the name ends and the number starts. E.g. locate the first digit:

```
int i = 0;
while (!Character.isDigit(line.charAt(i))) { i++; }
```

- Then extract the name and phone number:

```
String name = line.substring(0, i);
String phoneNumber = line.substring(i);
```

Reading Text Input: Processing Lines

- Use the `trim` method to remove spaces at the end of the country name:

```
name = name.trim();
```

- To convert the phone number string to a number, first trim it, then call the `Integer.parseInt` method:

```
int phoneNumberValue =
    Integer.parseInt(phoneNumber.trim());
```

Reading Text Input: Processing Lines

- Occasionally easier to construct a new `Scanner` object to read the characters from a string:

```
Scanner lineScanner = new Scanner(line);
```

- Then you can use `lineScanner` like any other `Scanner` object, reading words and numbers:

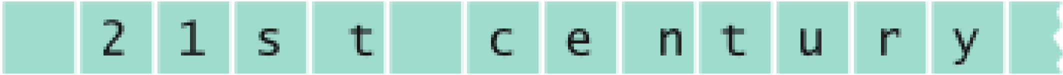
```
String name = lineScanner.next();
while (!lineScanner.hasNextInt())
{
   name = name+ " " + lineScanner.next();
}
int phoneNumberValue = lineScanner.nextInt();
```

Reading Text Input: Reading Numbers

- `nextInt` and `nextDouble` methods consume white space and the next number:

  ```
  double value = in.nextDouble();
  ```

- If there is no number in the input, then a `InputMismatchException` occurs; e.g.



- To avoid exceptions, use the `hasNextDouble` and `hasNextInt` methods to screen the input:

  ```
  if (in.hasNextDouble())
  {
     double value = in.nextDouble();
     . . .
  }
  ```

Reading Text Input: Reading Numbers

- `nextInt` and `nextDouble` methods do not consume the white space that follows a number
- Example: file contains student IDs and names in this format:

```
1729
Harry Morgan
1730
Diana Lin
. . .
```

- Read the file with these instructions:

```
while (in.hasNextInt())
{
   int studentID = in.nextInt();
   String name = in.nextLine();
   Process the student ID and name
}
```

Reading Text Input: Reading Numbers

- Initially, the input contains

1	7	2	9	\n	H	a	r	r	y	

- After the first call to `nextInt`, the input contains

\n	H	a	r	r	y	

- The call to `nextLine` reads an empty string! The remedy is to add a call to `nextLine` after reading the ID:

```
int studentID = in.nextInt();
in.nextLine(); // Consume the newline
String name = in.nextLine();
```

Reading Text Input: Reading Characters

- To read one character at a time, set the delimiter pattern to the empty string:

```
Scanner in = new Scanner(. . .);
in.useDelimiter("");
```

- Now each call to next returns a string consisting of a single character
- To process the characters:

```
while (in.hasNext())
{
   char ch = in.next().charAt(0);
   Process ch
}
```

Self Check

Suppose the input contains the characters `6,995.0`. What is the value of `number` and `input` after these statements?

```
int number = in.nextInt();
String input = in.next();
```

Answer: `number` is `6`, `input` is `",995.0"`.

Self Check

Suppose the input contains the characters `6,995.00 12`. What is the value of `price` and `quantity` after these statements?

```
double price = in.nextDouble();
int quantity = in.nextInt();
```

Answer: `price` is set to `6` because the comma is not considered a part of a floating-point number in Java. Then the call to `nextInt` causes an exception, and `quantity` is not set.

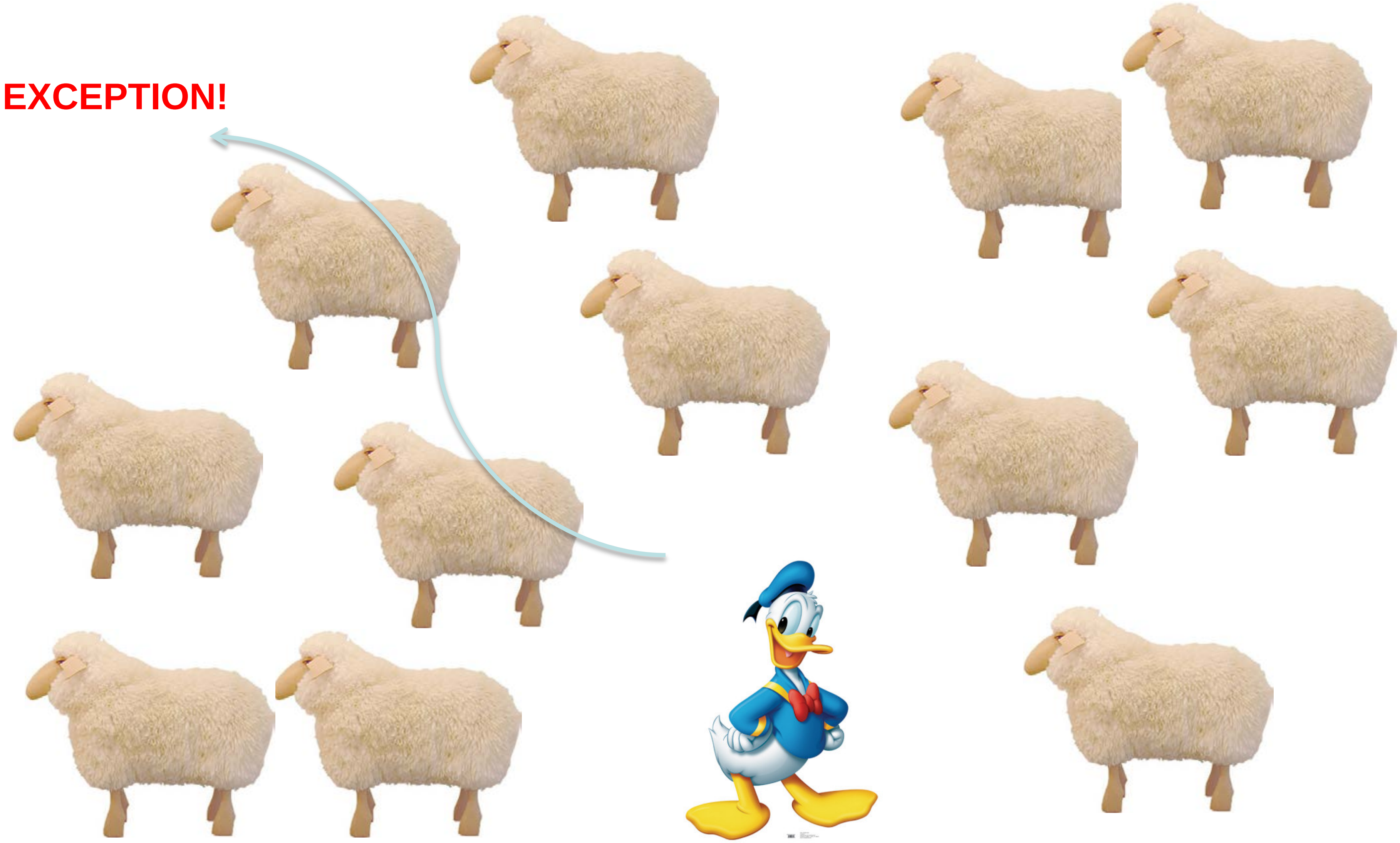
Self Check

Your input file contains a sequence of numbers, but sometimes a value is not available and marked as N/A. How can you read the numbers and skip over the markers?

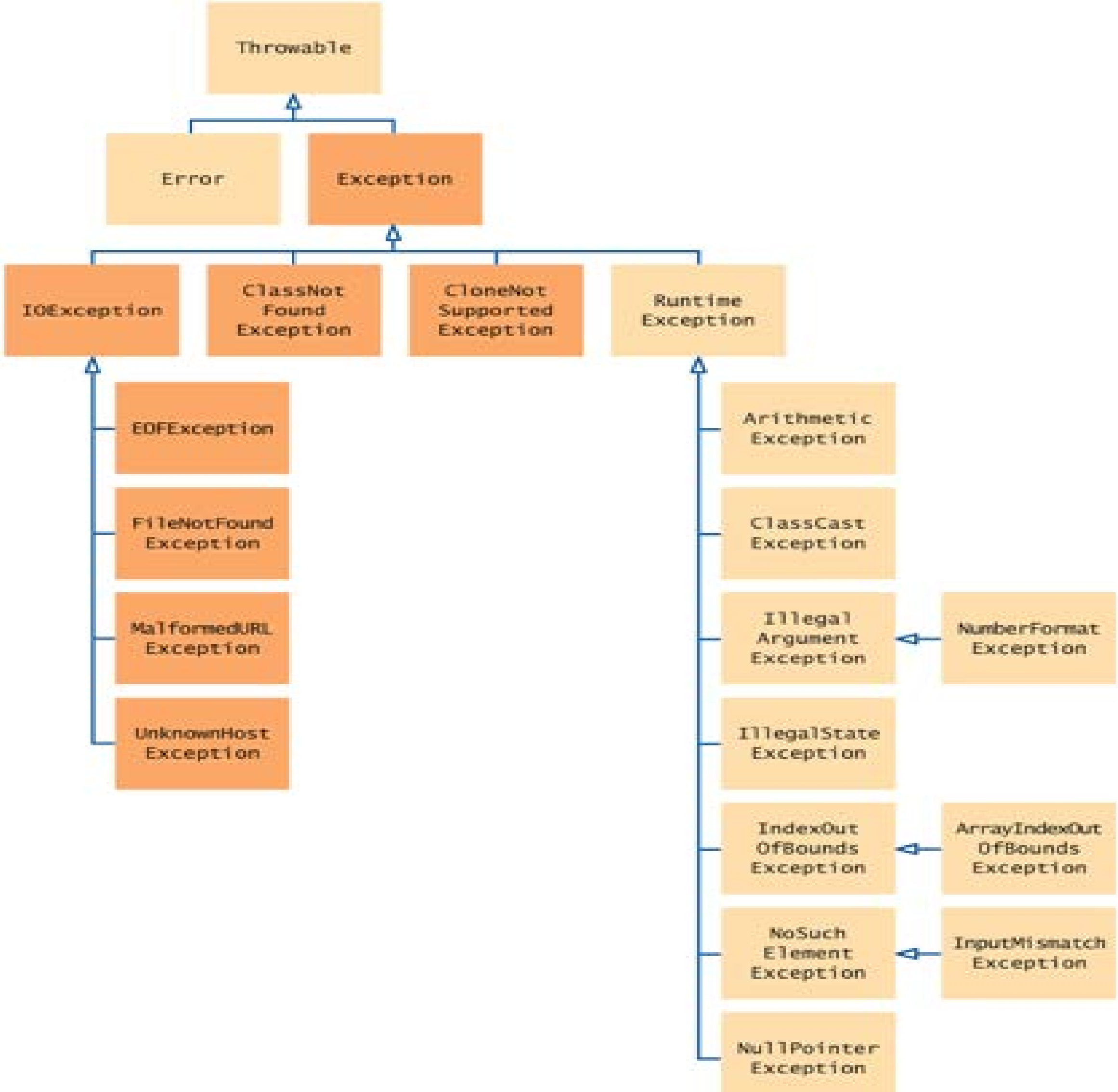
Answer: Read them as strings, and convert those strings to numbers that are not equal to N/A:

```
String input = in.next();
if (!input.equals("N/A"))
{
   double value = Double.parseDouble(input);
   Process value
}
```

Exception



Hierarchy of Exception Classes



Example

```
public class BankAccount
{
   public void withdraw(double amount)
   {
      if (amount > balance)
      {
         IllegalArgumentException exception
            = new IllegalArgumentException("Amount
            exceeds balance");
         throw exception;
      }
      balance = balance - amount;
   }
   . . .
}
```

Throwing Exceptions

- Throw an exception object to signal an exceptional condition
- Example: `IllegalArgumentException`: Illegal parameter value:

```
IllegalArgumentException exception
   = new IllegalArgumentException("Amount exceeds
   balance");
throw exception;
```

- No need to store exception object in a variable:

```
throw new IllegalArgumentException("Amount exceeds
   balance");
```

- When an exception is thrown, method terminates immediately
  - *Execution continues with an exception handler*

Syntax Throwing an Exception

Syntax `throw` *exceptionObject*;

Example

```
if (amount > balance)
{
   throw new IllegalArgumentException("Amount exceeds balance");
}
balance = balance - amount;
```

A new exception object is constructed, then thrown.

Most exception objects can be constructed with an error message.

This line is not executed when the exception is thrown.

Self Check

How should you modify the `deposit` method to ensure that the balance is never negative?

Self Check

Suppose you construct a new bank account object with a zero balance and then call `withdraw(10)`. What is the value of `balance` afterwards?

Checked and Unchecked Exceptions

- Two types of exceptions:
  - *Checked*
    - *The compiler checks that you don't ignore them*
    - *Due to external circumstances that the programmer cannot prevent*
    - *Majority occur when dealing with input and output*
    - *For example,* `IOException`
  - *Unchecked*
    - *Extend the class* `RuntimeException` or `Error`
    - *They are the programmer's fault*
    - *Examples of runtime exceptions:*
      `NumberFormatException`
      `IllegalArgumentException`
      `NullPointerException`
    - *Example of error:*
      `OutOfMemoryError`

Checked and Unchecked Exceptions

- Categories aren't perfect:
  - *`Scanner.nextInt` throws unchecked `InputMismatchException`*
  - *Programmer cannot prevent users from entering incorrect input*
  - *This choice makes the class easy to use for beginning programmers*
- Deal with checked exceptions principally when programming with files and streams
- For example, use a `Scanner` to read a file:

```
String filename = ...;
File reader = new File (filename);
Scanner in = new Scanner(reader);
```

- But, `Scanner` constructor can throw a `FileNotFoundException`

Checked and Unchecked Exceptions

- Two choices:
  1. *Handle the exception*
  2. *Tell compiler that you want method to be terminated when the exception occurs*
     - *Use* `throws` *specifier so method can throw a checked exception*

       ```
       public void read(String filename) throws
          FileNotFoundException
       {
          File reader = new File (filename);
          Scanner in = new Scanner(reader);
          ...
       }
       ```
     - *For multiple exceptions:*

       ```
       public void read(String filename)
          throws IOException, ClassNotFoundException
       ```
     - *Keep in mind inheritance hierarchy: If method can throw an* `IOException` *and* `FileNotFoundException`*, only use* `IOException`
- Better to declare exception than to handle it incompetently

Syntax `throws` *Clause*

Syntax *accessSpecifier returnType methodName(parameterType parameterName, . . .)*
`throws` *ExceptionClass, ExceptionClass, . . .*

Example

```
public void read(String filename)
      throws FileNotFoundException, NoSuchElementException
```

You must specify all checked exceptions that this method may throw.

You may also list unchecked exceptions.

Self Check

Suppose a method calls the `Scanner` constructor, which can throw a `FileNotFoundException`, and the `nextInt` method of the `Scanner` class, which can cause a `NoSuchElementException` or `InputMismatchException`. Which exceptions should be included in the `throws` clause?

Answer: You must include the `FileNotFoundException` and you may include the `NoSuchElementException` if you consider it important for documentation purposes. `InputMismatchException` is a subclass of `NoSuchElementException`. It is your choice whether to include it.

Self Check

Why is a `NullPointerException` not a checked exception?

Catching Exceptions

- Install an exception handler with `try/catch` statement
- `try` block contains statements that may cause an exception
- `catch` clause contains handler for an exception type

Catching Exceptions

- Example:

```
try
{
   String filename = ...;
   File reader = new File (filename);
   Scanner in = new Scanner(reader);
   String input = in.next();
   int value = Integer.parseInt(input);
   ...
}
catch (IOException exception)
{
   exception.printStackTrace();
}
catch (NumberFormatException exception)
{
 System.out.println("Input was not a number");
}
```

Catching Exceptions

- Statements in `try` block are executed
- If no exceptions occur, `catch` clauses are skipped
- If exception of matching type occurs, execution jumps to `catch` clause
- If exception of another type occurs, it is thrown until it is caught by another `try` block
- `catch (IOException exception)` *block*
  - *`exception` contains reference to the exception object that was thrown*
  - *`catch` clause can analyze object to find out more details*
  - *`exception.printStackTrace()`: Printout of chain of method calls that lead to exception*

Syntax Catching Exceptions

Syntax

```
try
{
   statement
   statement
   . . .
}
catch (ExceptionClass exceptionObject)
{
   statement
   statement
   . . .
}
```

Example

```
try
{
   Scanner in = new Scanner(new File("input.txt"));
   String input = in.next();
   process(input);
}
catch (IOException exception)
{
   System.out.println("Could not open input file");
}
```

This constructor can throw a FileNotFoundException.

This is the exception that was thrown.

When an IOException is thrown, execution resumes here.

Additional catch clauses can appear here.

A FileNotFoundException is a special case of an IOException.

Self Check

Suppose the file with the given file name exists and has no contents. Trace the flow of execution in the `try` block in this section.

Answer: The `File` constructor succeeds, and `in` is constructed. Then the call `in.next()` throws a `NoSuchElementException`, and the `try` block is aborted. None of the `catch` clauses match, so none are executed. If none of the enclosing method calls catch the exception, the program terminates.

Self Check

Is there a difference between catching checked and unchecked exceptions?

Answer: No - you catch both exception types in the same way, as you can see from the above code example. Recall that `IOException` is a checked exception and `NumberFormatException` is an unchecked exception.

Clause `finally`

- Exception terminates current method
- Danger: Can skip over essential code
- Example:

```
reader = new File (filename);
Scanner in = new Scanner(reader);
readData(in);
reader.close(); // May never get here
```

- Must execute `reader.close()` even if exception happens
- Use `finally` clause for code that must be executed “no matter what”

Clause **`finally`**

```
File reader = new File (filename);
try
{
   Scanner in = new Scanner(reader);
   readData(in);
}
finally
{
   reader.close();
   // if an exception occurs, finally clause
   // is also executed before exception
   // is passed to its handler
}
```

Clause `finally`

- Executed when `try` block is exited in any of three ways:
  1. *After last statement of `try` block*
  2. *After last statement of catch clause, if this `try` block caught an exception*
  3. *When an exception was thrown in `try` block and not caught*
- Recommendation: Don't mix `catch` and `finally` clauses in same `try` block

Syntax Clause `finally`

Syntax

```
try
{
   statement
   statement
   . . .
}
finally
{
   statement
   statement
   . . .
}
```

Example

This variable must be declared outside the `try` block so that the `finally` clause can access it.

This code may throw exceptions.

This code is always executed, even if an exception occurs.

```
PrintWriter out = new PrintWriter(filename);
try
{
   writeData(out);
}
finally
{
   out.close();
}
```

Self Check

Why was the `out` variable declared outside the `try` block?

Self Check

Suppose the file with the given name does not exist. Trace the flow of execution of the code segment in this section.

Designing Your Own Exception Types

- You can design your own exception types - subclasses of `Exception` or `RuntimeException`

```
if (amount > balance)
{
   throw new InsufficientFundsException(
      "withdrawal of " + amount + " exceeds balance of "
      + balance);
}
```

- Make it an unchecked exception - programmer could have avoided it by calling `getBalance` first
- Extend `RuntimeException` or one of its subclasses
- Supply two constructors
  1. *Default constructor*
  2. *A constructor that accepts a message string describing reason for exception*

Designing Your Own Exception Types

```
public class InsufficientFundsException
      extends RuntimeException
{
   public InsufficientFundsException() {}

   public InsufficientFundsException(String message)
   {
      super(message);
   }
}
```

Self Check

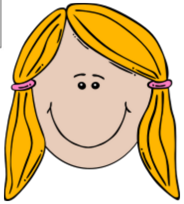
What is the purpose of the call `super(message)` in the second `InsufficientFundsException` constructor?

Self Check

Suppose you read bank account data from a file. Contrary to your expectation, the next input value is not of type `double`. You decide to implement a `BadDataException`. Which exception class should you extend?

> **Answer:** Because file corruption is beyond the control of the programmer, this should be a checked exception, so it would be wrong to extend `RuntimeException` or `IllegalArgumentException`. Because the error is related to input, `IOException` would be a good choice.

Morale from today's story



Competent exception handler **catch** it.

Throw your problem if you don't know how to handle it!

Case Study: A Complete Example

- Program
  - Asks user for name of file
  - File expected to contain data values
  - First line of file contains total number of values
  - Remaining lines contain the data
  - Typical input file:

```
3
1.45
-2.1
0.05
```

Case Study: A Complete Example

- What can go wrong?
  - *File might not exist*
  - *File might have data in wrong format*
- Who can detect the faults?
  - *`Scanner` constructor will throw an exception when file does not exist*
  - *Methods that process input need to throw exception if they find error in data format*
- What exceptions can be thrown?
  - *`FileNotFoundException` can be thrown by `Scanner` constructor*
  - *`IOException` can be thrown by `close` method of `Scanner`*
  - *`BadDataException`, a custom checked exception class*
- Who can remedy the faults that the exceptions report?
  - *Only the `main` method of `DataSetTester` program interacts with user*
  - *Catches exceptions*
  - *Prints appropriate error messages*
  - *Gives user another chance to enter a correct file*

DataAnalyzer.java

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Scanner;

/**
   This program reads a file containing numbers and analyzes its contents.
   If the file doesn&apos;t exist or contains strings that are not numbers, an
   error message is displayed.
*/
public class DataAnalyzer
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      DataSetReader reader = new DataSetReader();

      boolean done = false;
      while (!done)
      {
```

DataAnalyzer.java (cont.)

```
         try
         {
            System.out.println("Please enter the file name: ");
            String filename = in.next();

            double[] data = reader.readFile(filename);
            double sum = 0;
            for (double d : data) sum = sum + d;
            System.out.println("The sum is " + sum);
            done = true;
         }
         catch (FileNotFoundException exception)
         {
            System.out.println("File not found.");
         }
         catch (BadDataException exception)
         {
            System.out.println("Bad data: " + exception.getMessage());
         }
         catch (IOException exception)
         {
            exception.printStackTrace();
         }
      }
   }
}
```

The `readFile` *Method of the* `DataSetReader` *Class*

- Constructs `Scanner` object
- Calls `readData` method
- Completely unconcerned with any exceptions
- If there is a problem with input file, it simply passes the exception to caller:

```
public double[] readFile(String filename)
      throws IOException
      // FileNotFoundException is an IOException
{
   File reader = new File (filename);
   Scanner in = new Scanner(reader);
   try
   {
      readData(in);
         return data;
   }
   finally
   {
      reader.close();
   }
}
```

The `readFile` *Method of the* `DataSetReader` *Class*

- Reads the number of values
- Constructs an array
- Calls `readValue` for each data value:

```
private void readData(Scanner in) throws BadDataException
{
   if (!in.hasNextInt())
      throw new BadDataException("Length expected");
   int numberOfValues = in.nextInt();
   data = new double[numberOfValues];

   for (int i = 0; i < numberOfValues; i++)
      readValue(in, i);

   if (in.hasNext())
      throw new BadDataException("End of file expected");
}
```

The `readFile` *Method of the* `DataSetReader` *Class*

- Checks for two potential errors
  1. *File might not start with an integer*
  2. *File might have additional data after reading all values*
- Makes no attempt to catch any exceptions

```
private void readValue(Scanner in, int i) throws
   BadDataException
{
   if (!in.hasNextDouble())
      throw new BadDataException("Data value expected");
   data[i] = in.nextDouble();
}
```

Scenario

1. `DataSetTester.main` calls `DataSetReader.readFile`
2. `readFile` calls `readData`
3. `readData` calls `readValue`
4. `readValue` doesn't find expected value and throws `BadDataException`
5. `readValue` has no handler for exception and terminates
6. `readData` has no handler for exception and terminates
7. `readFile` has no handler for exception and terminates after executing `finally` clause
8. `DataSetTester.main` has handler for `BadDataException`; handler prints a message, and user is given another chance to enter file name

DataSetReader.java

```
import java.io.File;
import java.io.IOException;
import java.util.Scanner;

/**
   Reads a data set from a file. The file must have the format
   numberOfValues
   value1
   value2
   . . .
*/
public class DataSetReader
{
   private double[] data;

   /**
      Reads a data set.
      @param filename the name of the file holding the data
      @return the data in the file
   */
   public double[] readFile(String filename) throws IOException
   {
      File inFile = new File(filename);
      Scanner in = new Scanner(inFile);
      try
      {
         readData(in);
         return data;
      }
      finally
      {
         in.close();
      }
   }

```

DataSetReader.java (cont.)

```
   /**
      Reads all data.
      @param in the scanner that scans the data
   */
   private void readData(Scanner in) throws BadDataException
   {
      if (!in.hasNextInt())
         throw new BadDataException("Length expected");
      int numberOfValues = in.nextInt();
      data = new double[numberOfValues];

      for (int i = 0; i < numberOfValues; i++)
         readValue(in, i);

      if (in.hasNext())
         throw new BadDataException("End of file expected");
   }

   /**
      Reads one data value.
      @param in the scanner that scans the data
      @param i the position of the value to read
   */
   private void readValue(Scanner in, int i) throws BadDataException
   {
      if (!in.hasNextDouble())
         throw new BadDataException("Data value expected");
      data[i] = in.nextDouble();
   }
}
```

BadDataException.java

```
import java.io.IOException;

/**
   This class reports bad input data.
*/
public class BadDataException extends IOException
{
   public BadDataException() {}
   public BadDataException(String message)
   {
      super(message);
   }
}
```

Self Check

Why doesn’t the `DataSetReader.readFile` method catch any exceptions?

Answer: It would not be able to do much with them. The `DataSetReader` class is a reusable class that may be used for systems with different languages and different user interfaces. Thus, it cannot engage in a dialog with the program user.

Self Check

Suppose the user specifies a file that exists and is empty. Trace the flow of execution.

Answer: `DataSetAnalyzer.main` calls `DataSetReader.readFile`, which calls `readData`. The call `in.hasNextInt()` returns `false`, and `readData` throws a `BadDataException`. The `readFile` method doesn’t catch it, so it propagates back to `main`, where it is caught.

THANK YOU FOR YOUR ATTENTION !