Objects and Classes -- Introduction

- Now that some low-level programming concepts have been established, we can examine objects in more detail

- Chapter 4 focuses on:
  - the concept of objects
  - the use of classes to create objects
  - using predefined classes
  - defining methods and passing parameters
  - defining classes
  - visibility modifiers
  - static variables and methods
  - method overloading

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Objects

- An object has:
  - *state* - descriptive characteristics
  - *behaviors* - what it can do (or be done to it)

- For example, a particular bank account
  - has an account number
  - has a current balance
  - can be deposited into
  - can be withdrawn from

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Classes

- A *class* is a blueprint of an object
- It is the model or pattern from which objects are created
- A class defines the methods and types of data associated with an object
- Creating an object from a class is called *instantiation*; an object is an *instance* of a particular class
- For example, the `Account` class could describe many bank accounts, but `toms_savings` is a particular bank account with a particular balance

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Creating Objects

- The `new` operator creates an object from a class:

```
Account toms_savings = new Account ();
```

- This declaration asserts that `toms_savings` is a variable that refers to an object created from the `Account` class

- It is initialized to the object created by the `new` operator

- The newly created object is set up by a call to a *constructor* of the class

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Constructors

- A constructor is a special method used to set up an object
- It has the same name as the class
- It can take parameters, which are often used to initialize some variables in the object
- For example, the `Account` constructor could be set up to take a parameter specifying its initial balance:

```
Account toms_savings = new Account (125.89);
```


Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Object References

- The declaration of the *object reference* variable and the creation of the object can be separate activities:

```
Account toms_savings;

toms_savings = new Account (125.89);
```

- Once an object exists, its methods can be invoked using the *dot operator*:

```
toms_savings.deposit (35.00);
```


Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The String Class

- A character string in Java is an object, defined by the `String` class

```
String name = new String ("Ken Arnold");
```

- Because strings are so common, Java allows an abbreviated syntax:

```
String name = "Ken Arnold";
```

- Java strings are *immutable*; once a string object has a value, it cannot be changed

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The String Class

- A character in a string can be referred to by its position, or *index*
- The index of the first character is zero
- The `String` class is defined in the `java.lang` package (and is therefore automatically imported)
- Many helpful methods are defined in the `String` class
- See `Carpe_Diem.java`

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The StringTokenizer Class

- The `StringTokenizer` class makes it easy to break up a string into pieces called *tokens*
- By default, the *delimiters* for the tokens are the space, tab, carriage return, and newline characters (white space)
- The `StringTokenizer` class is defined in the `java.util` package
- See `Int_Reader.java`

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The Random Class

- A program may need to produce a random number
- The `Random` class provides methods to simulate a *random number generator*
- The `nextInt` method returns a random number from the entire spectrum of `int` values
- Usually, the number must be *scaled* and *shifted* into a particular range to be useful
- See `Flip.java`

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The Random Class

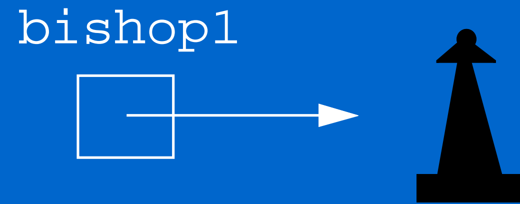
Expression	Range
`Math.abs (rand.newInt()) % 6 + 1`	1 to 6
`Math.abs (rand.newInt()) % 10 + 1`	1 to 10
`Math.abs (rand.newInt()) % 101`	0 to 100
`Math.abs (rand.newInt()) % 11 + 20`	20 to 30
`Math.abs (rand.newInt()) % 11 - 5`	-5 to 5

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

References

- An object reference holds the memory address of an object

```
Chess_Piece bishop1 = new Chess_Piece();
```



- All interaction with an object occurs through a reference variable

- References have an effect on actions such as assignment

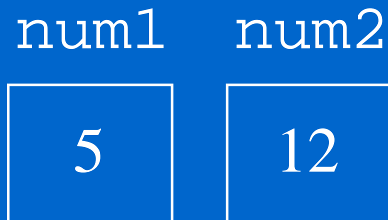
Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Assignment

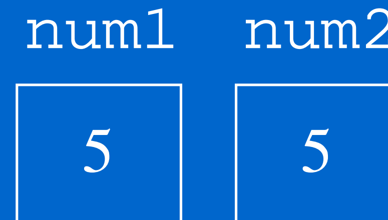
- The act of assignment takes a copy of a value and stores it in a variable
- For primitive types:

```
num2 = num1;
```

Before



After



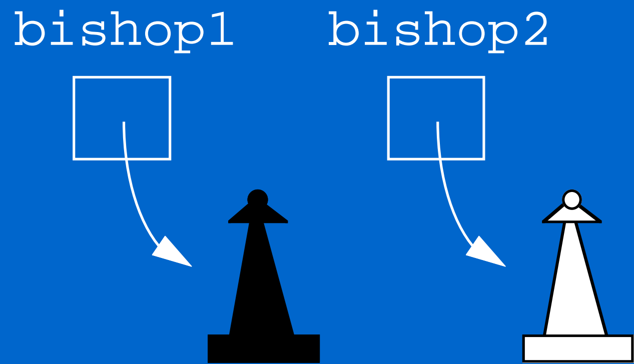
Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Reference Assignment

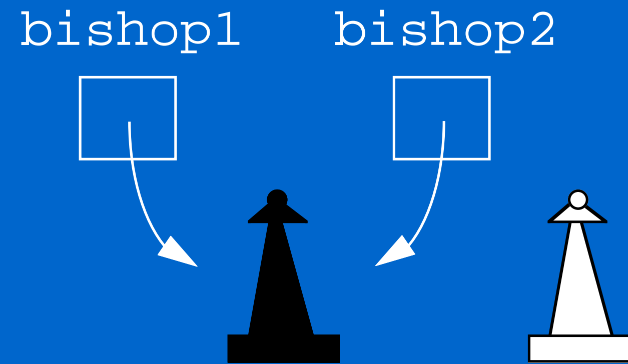
- For object references, the value of the memory location is copied:

```
bishop2 = bishop1;
```

<u>Before</u>



<u>After</u>



Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Aliases

- Two or more references that refer to the same object are called *aliases* of each other
- There is only one copy of the object (and its data), but with multiple ways to access it
- Aliases can be useful, but should be managed carefully
- Affecting the object through one reference affects it for all aliases, because they refer to the same object

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Garbage Collection

- When an object no longer has any valid references to it, it can no longer be accessed by the program
- It is useless, and therefore called *garbage*
- Java performs *automatic garbage collection* periodically, returning an object's memory to the system for future use
- In other languages, the programmer has the responsibility for performing garbage collection

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Methods

- A class contains methods; prior to defining our own classes, we must explore method definitions
- We've defined the `main` method many times
- All methods follow the same syntax:

```
return-type method-name ( parameter-list ) {
    statement-list
}
```


Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Methods

- A method definition:

```
int third_power (int number) {
   int cube;
   cube = number * number * number;
   return cube;
}  // method third_power
```


Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Methods

- A method may contain *local declarations* as well as executable statements
- Variables declared locally can only be used locally
- The `third_power` method could be written without any local variables:

```
int third_power (int number) {
   return number * number * number;
}  // method third_power
```


Copyright 1997 by John Lewis and William Loftus. All rights reserved.

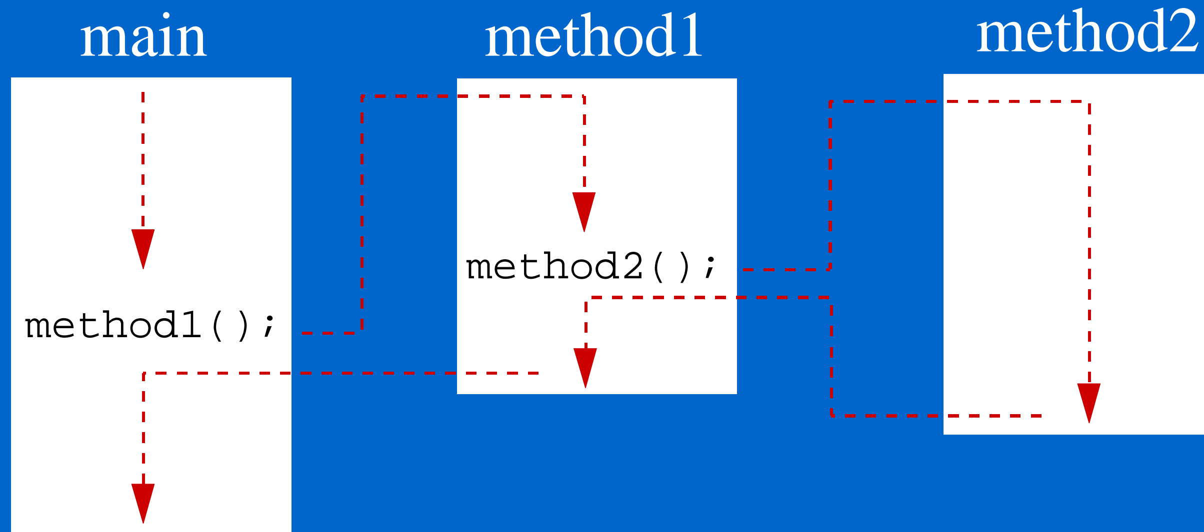
The return Statement

- The *return type* of a method indicates the type of value that the method sends back to the calling location
- A method that does not return a value (such as `main`) has a `void` return type
- The *return statement* specifies the value that will be returned
- Its expression must conform to the return type

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Method Flow of Control

- The `main` method is invoked by the system when you submit the bytecode to the interpreter
- Each method call returns to the place that called it



Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Parameters

- A method can be defined to accept zero or more parameters
- Each parameter in the parameter list is specified by its type and name
- The parameters in the method definition are called *formal parameters*
- The values passed to a method when it is invoked are called *actual parameters*

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Parameters

- When a parameter is passed, a copy of the value is made and assigned to the formal parameter
- Both primitive types and object references can be passed as parameters
- When an object reference is passed, the formal parameter becomes an alias of the actual parameter
- See `Parameter_Passing.java`
- Usually, we will avoid putting multiple methods in the class that contains the `main` method

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Defining Classes

- The syntax for defining a class is:

```
class class-name {
    declarations
    constructors
    methods
}
```

- The variables, constructors, and methods of a class are generically called *members* of the class

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Defining Classes

```
class Account {

   int account_number;
   double balance;

   Account (int account, double initial) {
      account_number = account;
      balance = initial;
   }  // constructor Account

   void deposit (double amount) {
      balance = balance + amount;
   }  // method deposit

}  // class Account
```


Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Constructors

- A constructor:
  - is a special method that is used to set up a newly created object
  - often sets the initial values of variables
  - has the same name as the class
  - does not return a value
  - has no return type, not even `void`

- The programmer does not have to define a constructor for a class

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

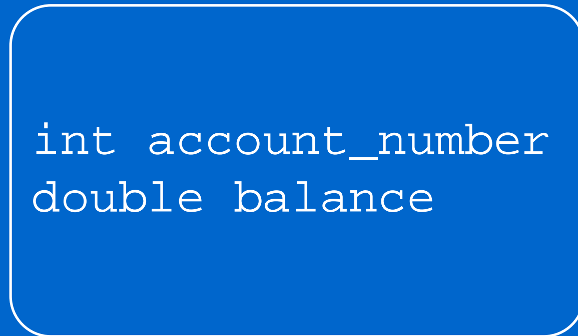
Classes and Objects

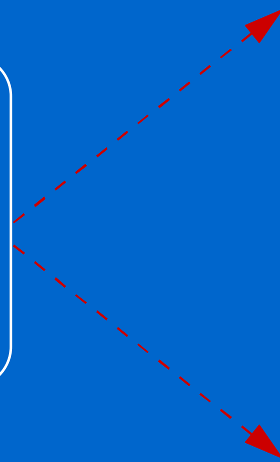
- A class defines the data types for an object, but a class does not store data values
- Each object has its own unique data space
- The variables defined in a class are called *instance variables* because each instance of the class has its own
- All methods in a class have access to all instance variables of the class
- Methods are shared among all objects of a class

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

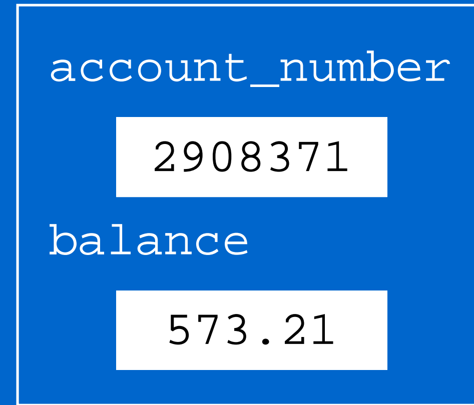
Classes and Objects

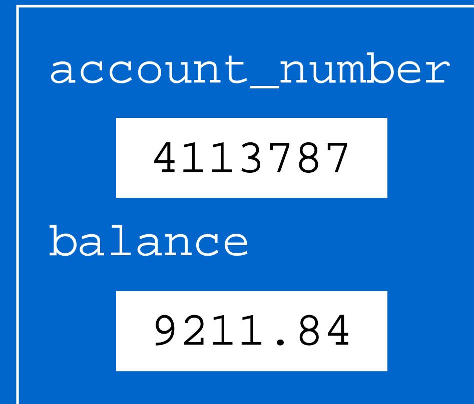
Class





Objects





Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Encapsulation

- You can take one of two views of an object:
  - internal - the structure of its data, the algorithms used by its methods
  - external - the interaction of the object with other objects in the program
- From the external view, an object is an *encapsulated* entity, providing a set of specific services
- These services define the *interface* to the object

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

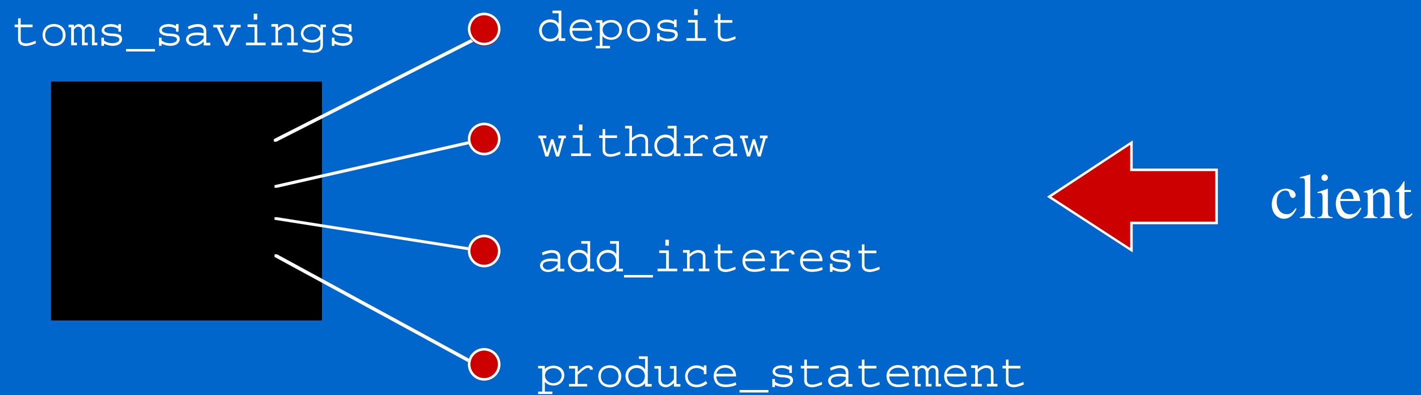
Encapsulation

- An object should be self-governing; any changes to the object's state (its variables) should be accomplished by that object's methods
- We should make it difficult, if not impossible, for another object to "reach in" and alter an object's state
- The user, or *client*, of an object can request its services, but it should not have to be aware of how those services are accomplished

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Encapsulation

- An encapsulated object can be thought of as a *black box*; its inner workings are hidden to the client



Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Abstraction

- Encapsulation is a powerful abstraction
- An *abstraction* hides the right details at the right time
- We use abstractions every day:
  - driving a car
  - using a computer
- Encapsulation makes an object easy to manage mentally because its interaction with clients is limited to a set of well-defined services

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Visibility Modifiers

- We accomplish encapsulation through the appropriate use of *visibility modifiers*
- A *modifier* is a Java reserved word that specifies particular characteristics of a programming construct
- We've used the modifier `final` to define a constant
- Java has three visibility modifiers: `public`, `private`, and `protected`
- We will discuss the `protected` modifier later

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Visibility Modifiers

- Members of a class that are declared with *public visibility* can be accessed from anywhere
- Members of a class that are declared with *private visibility* can only be accessed from inside the class
- Members declared without a visibility modifier have *default visibility* and can be accessed by any class in the same package
- Java modifiers are discussed in detail in Appendix F

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Visibility Modifiers

- As a general rule, no object's data should be declared with public visibility
- Methods that provide the object's services are usually declared with public visibility so that they can be invoked by clients
- Public methods are also called *service methods*
- Other methods, called *support methods*, can be defined that assist the service methods; they should not be declared with public visibility

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Classes and Objects

- See `Tunes.java`

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The static Modifier

- The `static` modifier can be applied to variables or methods
- It associates a variable or method with the class rather than an object
- This approach is a distinct departure from the normal way of thinking about objects

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Static Variables

- Normally, each object has its own data space
- If a variable is declared as static, only one copy of the variable exists for all objects of the class

```
private static int count;
```

- Changing the value of a static variable in one object changes it for all others
- Static variables are sometimes called *class variables*

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Static Methods

- Normally, we invoke a method through an instance (an object) of a class
- If a method is declared as static, it can be invoked through the class name; no object needs to exist
- For example, the `Math` class in the `java.lang` package contains several static mathematical operations

`Math.abs (num)` -- absolute value

`Math.sqrt (num)` -- square root

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Static Methods

- The `main` method is static; it is invoked by the system without creating an object
- Static methods cannot reference instance variables, because instance variables don't exist until an object exists
- However, they can reference static variables or local variables
- Static methods are sometimes called *class methods*

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Overloaded Methods

- *Method overloading* is the process of using the same method name for multiple methods
- The *signature* of each overloaded method must be unique
- The signature is based on the number, type, and order of the parameters
- The compiler must be able to determine which version of the method is being invoked by analyzing the parameters
- The return type of the method is <u>not</u> part of the signature

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Overloaded Methods

- The `println` method is overloaded:

```
println (String s)
println (int i)
println (double d)
```

etc.

- The lines

```
System.out.println ("The total is:");
System.out.println (total);
```

invoke different versions of the `println` method

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Overloaded Methods

- Constructors are often overloaded to provide multiple ways to set up a new object

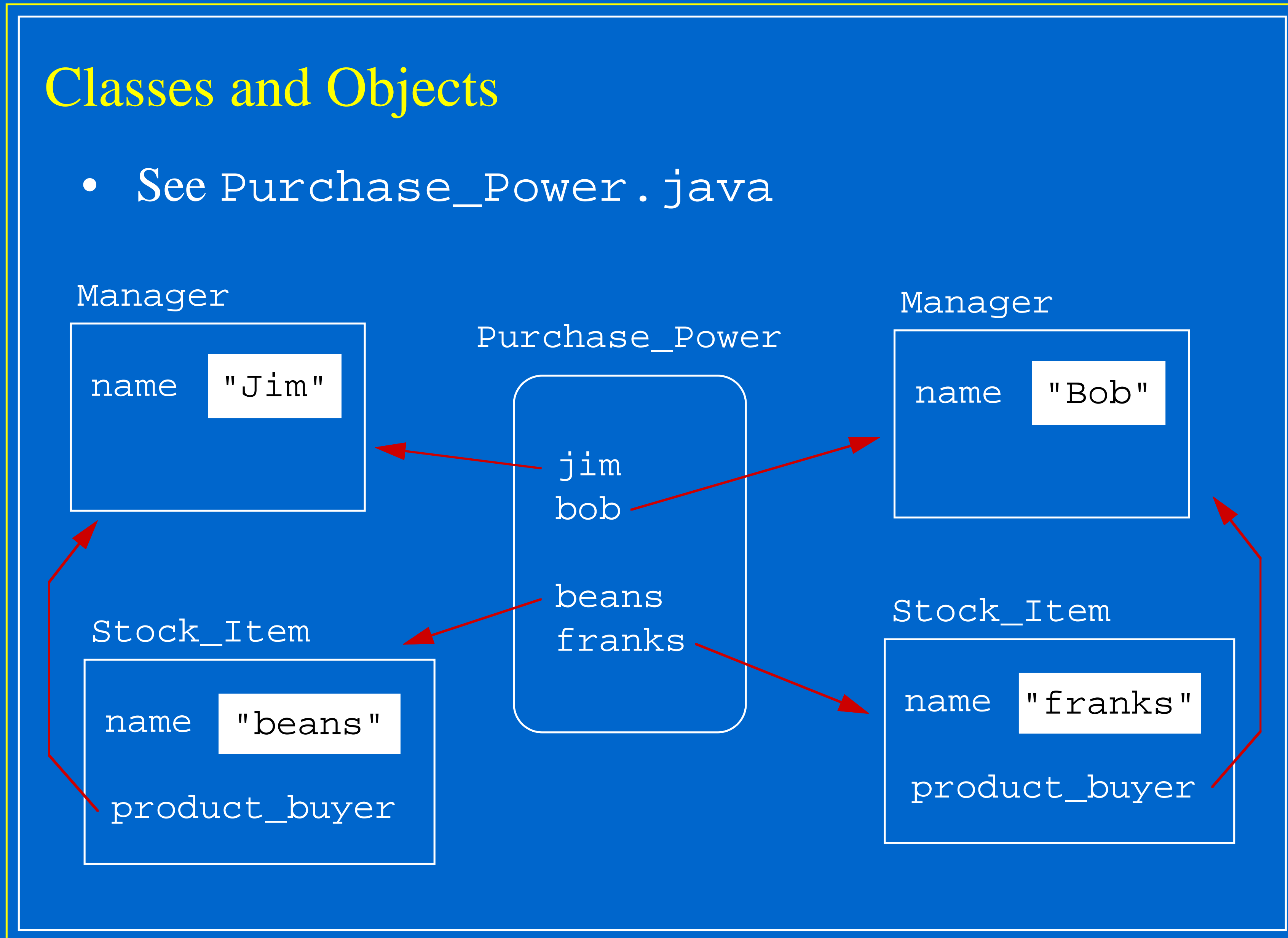
```
Account (int account) {
   account_number = account;
   balance = 0.0;
}   // constructor Account

Account (int account, double initial) {
   account_number = account;
   balance = initial;
}  // constructor Account
```

- See `Casino.java`

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Classes and Objects

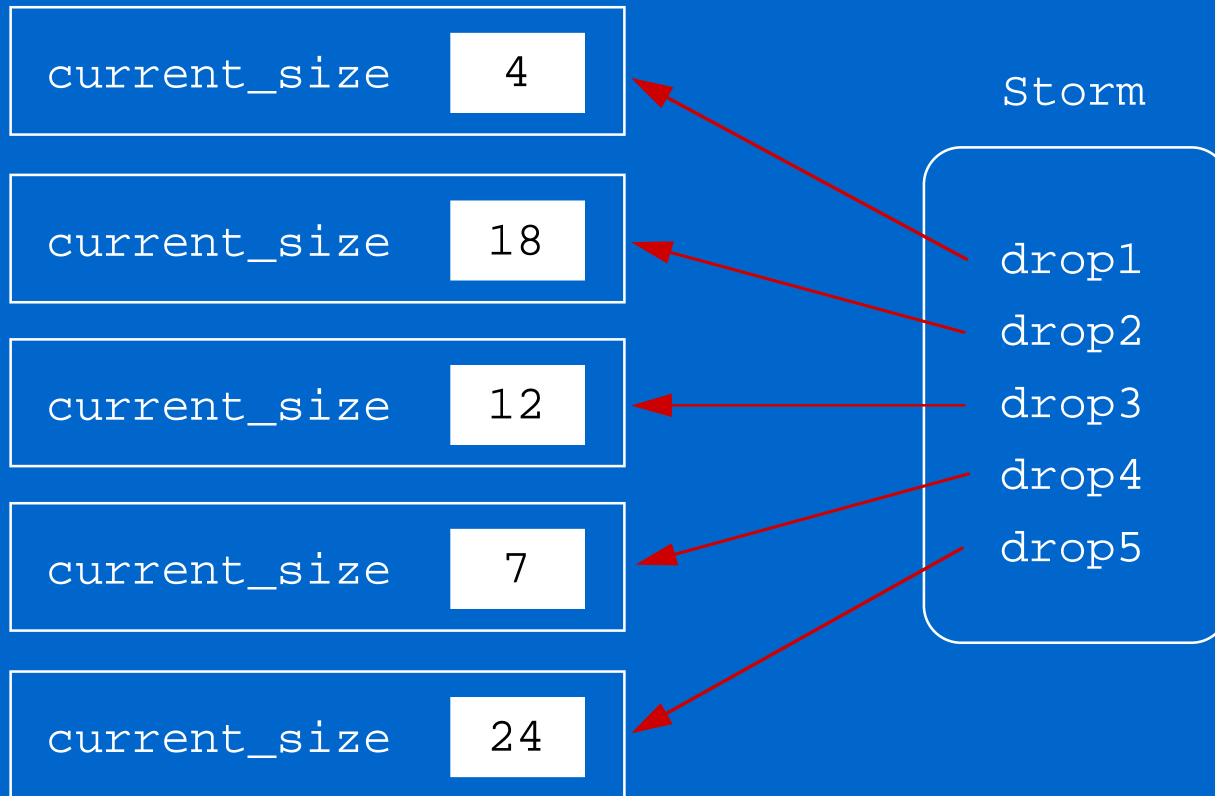
- See `Purchase_Power.java`



Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Classes and Objects

- See `Storm.java`



Copyright 1997 by John Lewis and William Loftus. All rights reserved.