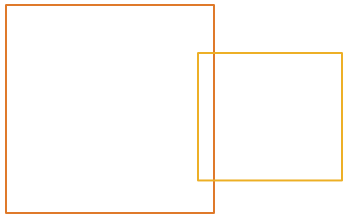
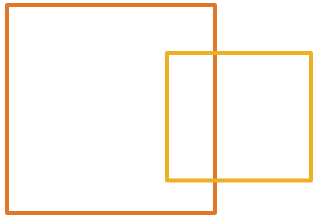
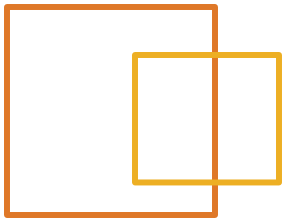
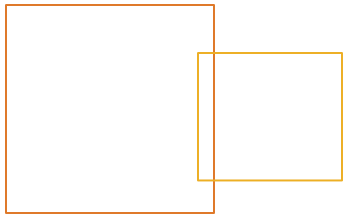
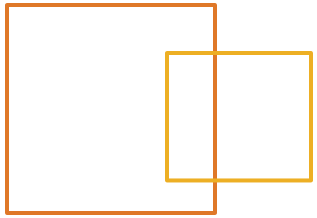
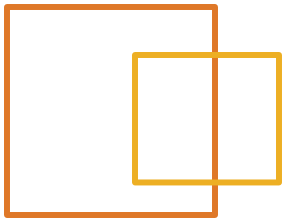





Fast Track to Java

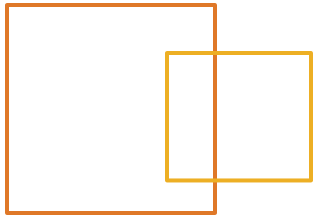
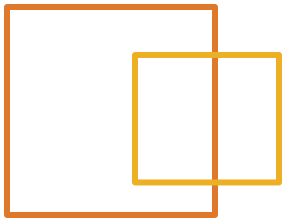
Customized for Starbucks

Delivered by DevelopIntelligence


Objects And Classes (Part 1)

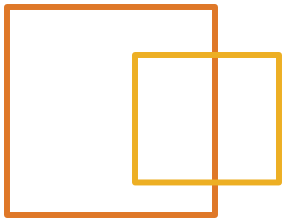
Objectives


At the end of this module, you should be able to

- Use the new operator to create objects
- Describe how reference variables work
- Use instance variables and methods
- Discuss the use of constructors

Creating Objects



- Every OOP language must have a mechanism for creating objects from the class definitions
- Java uses the instantiation mechanism found in other OOP languages -- the ***new*** operator
- There is one normal way to create an object in Java - by using the ***new*** operator
- The ***new*** operator is used in conjunction with a *constructor* to create, *instantiate*, an object
- The virtual machine is responsible for creating the memory associated with the object and initializes the memory through a constructor

Creating a BankApp Object

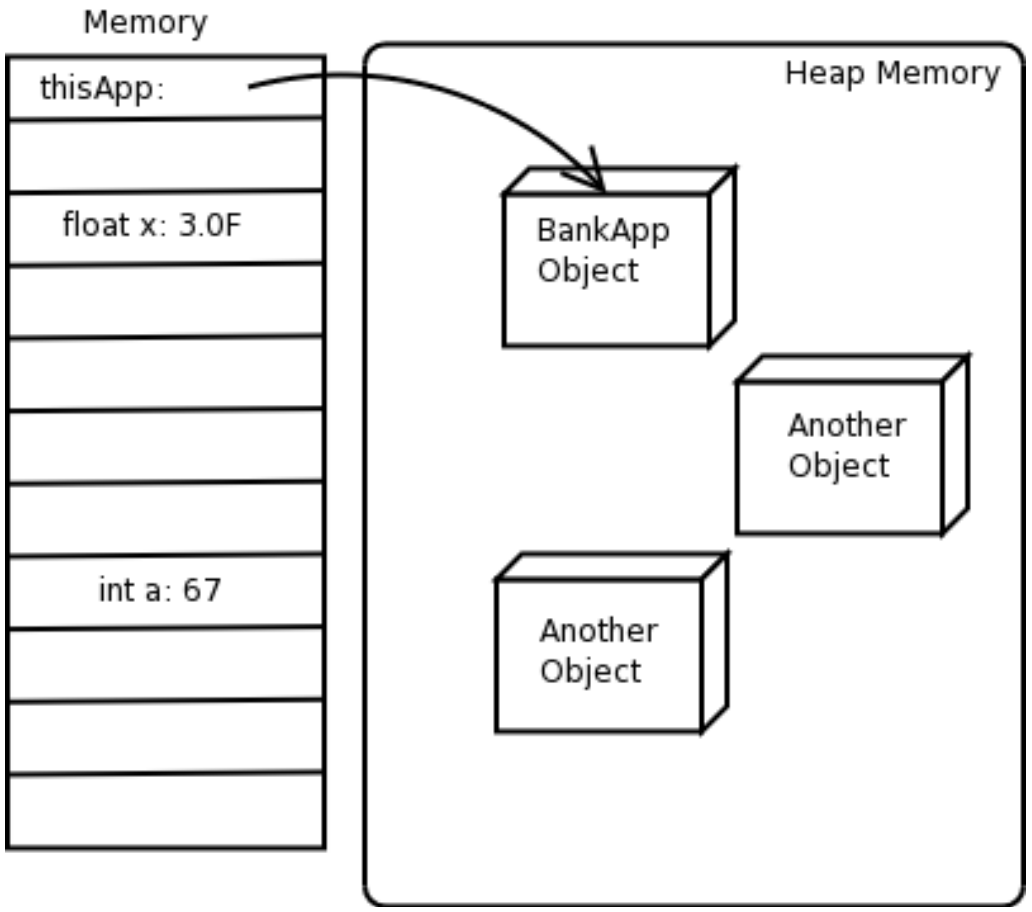
```
public class BankApp {
  public static void main(String [] args) {
     // create the BankApp object
     BankApp thisApp = new BankApp();
  }
}
```

Sequence of Instantiation

A lot goes on behind the scenes when you create a new object

1. The JVM determines what type of object to create
    1. Looks at the *type* following the new operator
    2. We will look at constructors in detail a bit later
2. The JVM loads the associated class (if it is not already loaded)
3. The JVM allocates enough memory in the *Heap* to hold the newly created object
4. The new object is initialized by
    1. Performing default initialization of all instance variables
    2. Executing explicit initialization of instance variables
    3. Executing the specified constructor
5. A *reference*, which we can think of as a pointer to a newly created object, is then returned and assigned to the *reference variable* **`thisApp`**

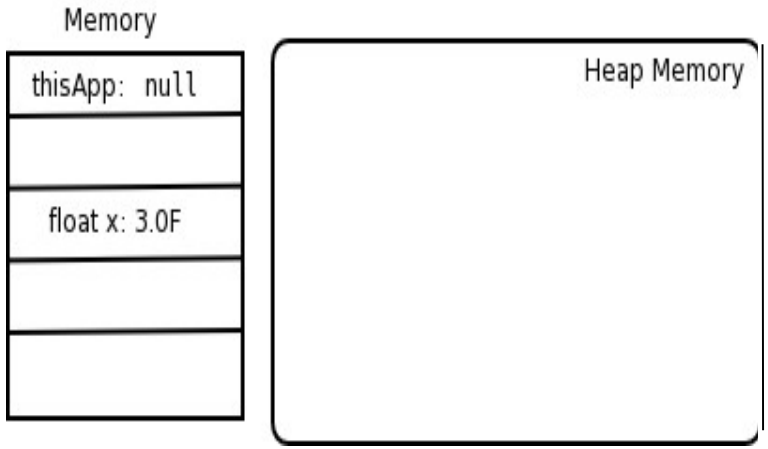
Reference Variables



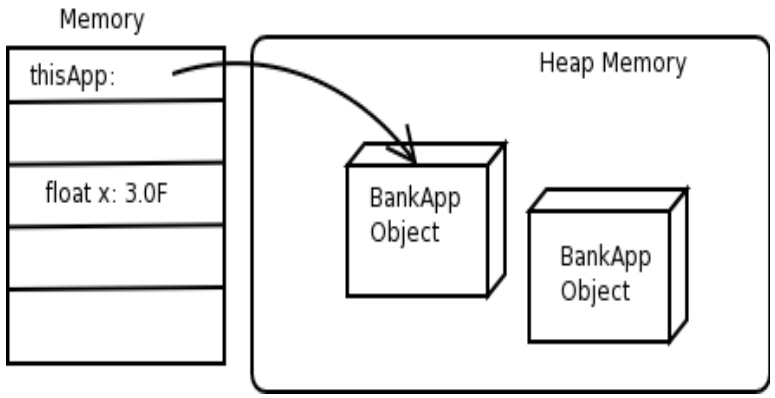
Memory allocation for reference variable thisApp

Reference Variables

```
public class BankApp {
    public static void main(String [] args){
      // Declare the variable.
      BankApp  thisApp = null;
      // Create and assign the object
      thisApp = new BankApp();
      // Create another BankApp object
      // don't assign IT to a variable
      // now we have no way to refer to it!
      new BankApp();
    }
  }
```

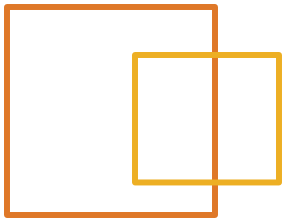


Reference variable with no associated object



Final state of the example

Object Description



- Objects are normally described by two things
  - Instance variables (*state*)
  - Instance methods (*behavior*)
- Both instance variables and methods are defined in class
- Their availability for use occurs once an object has been instantiated
- Referred to using the *dot-notation*

Object Description (cont.)

- Instance variables are known by many names:
  - *Attributes, States, Instance Variables, Member Variables, Members*
  - Members do not exist until an object of that type is instantiated
- Instance methods are also known by many names:
  - *Behaviors, Methods, Instance Methods*
  - Methods cannot be invoked until an object of that type is instantiated

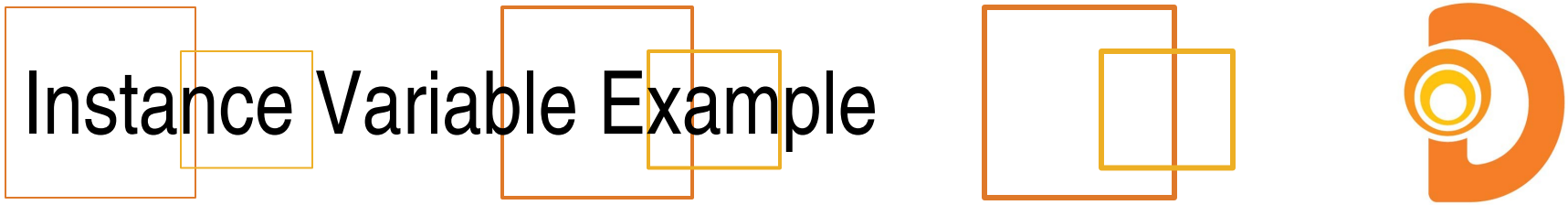
Instance Variables

- Hold data for a specific object
  - Each object has its own memory for the instance variables
  - Instance variables exist as long as the containing object exists
  - Instance variables live and die with their instance
- Can be either primitive data types or reference types
- Instance variables are initialized several ways (in order):
  - Default initialization
  - Explicit initialization
  - Initializer
  - Constructor(s)

Instance Variables (cont.)

- The instance variable values can be adjusted either by
  - Accessing them directly
    objectVariable.variableName = xxx;
  - Invoking a method that manipulates them
    objectVariable.setVariableName(xxx);
- The manner in which you access instance variables will depend on class design

Instance Variable Example



```
class BankAccount {
  float balance;
  String acNum;
}
class BankApp {
  public static void main(String [] args) {
    BankAccount account1 = new BankAccount();
    BankAccount account2 = new BankAccount();
    System.out.println("Account1 =" + account1);
    System.out.println("Account2 =" + account2);
    // Set the balances and account numbers
    account1.balance= 34.50F;      account2.balance = 100.00F;
    account1.acNum= "888888";      account2.acNum = "337722";
    // Print out the data
    System.out.println("Account 1:\nAccount Number:"+
      account1.acNum + "  Balance=" + account1.balance);
    System.out.println("Account 2:\nAccount Number:"+
      account2.acNum + "  Balance=" + account2.balance);
  }
}
```

Instance Variable Example Output

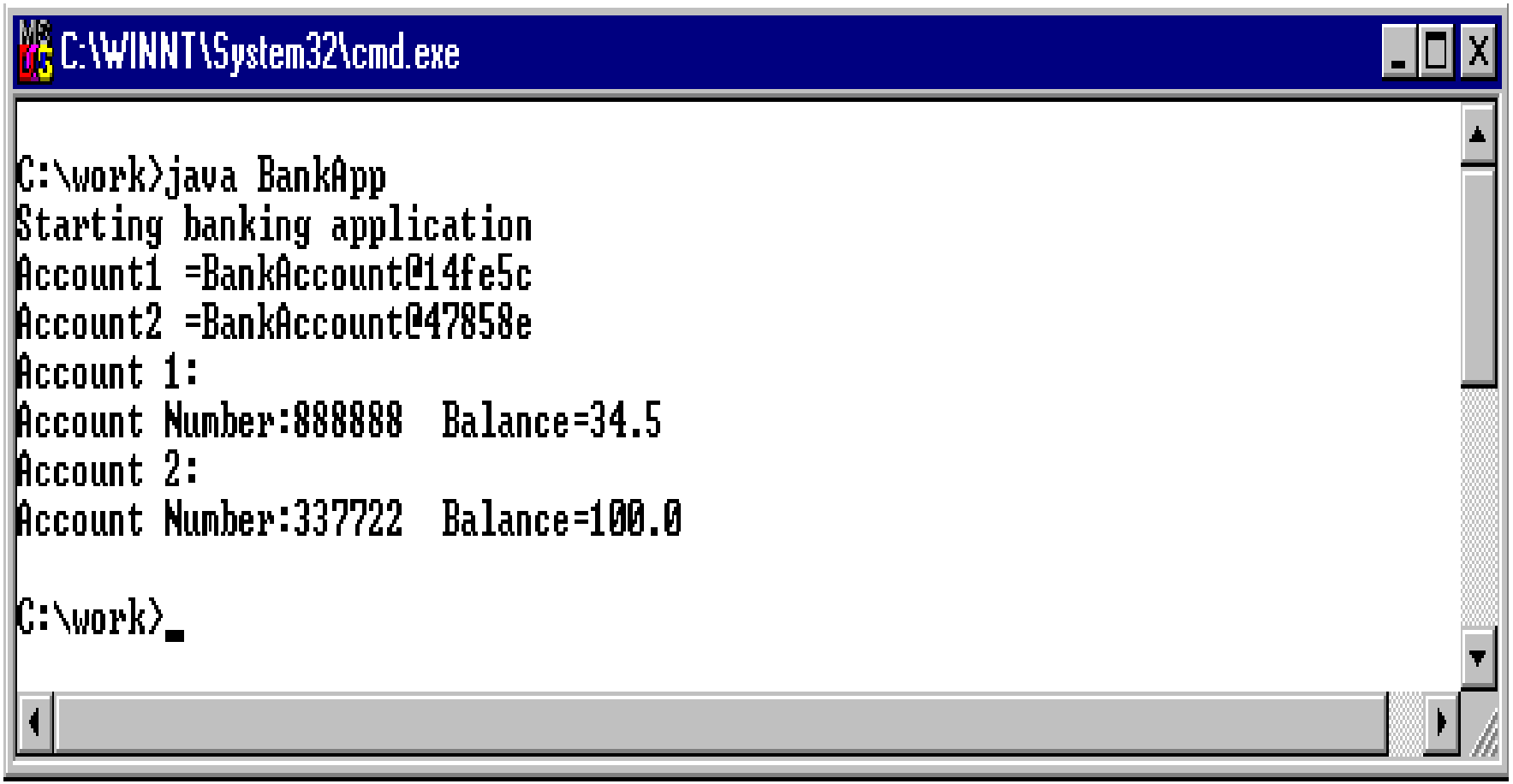
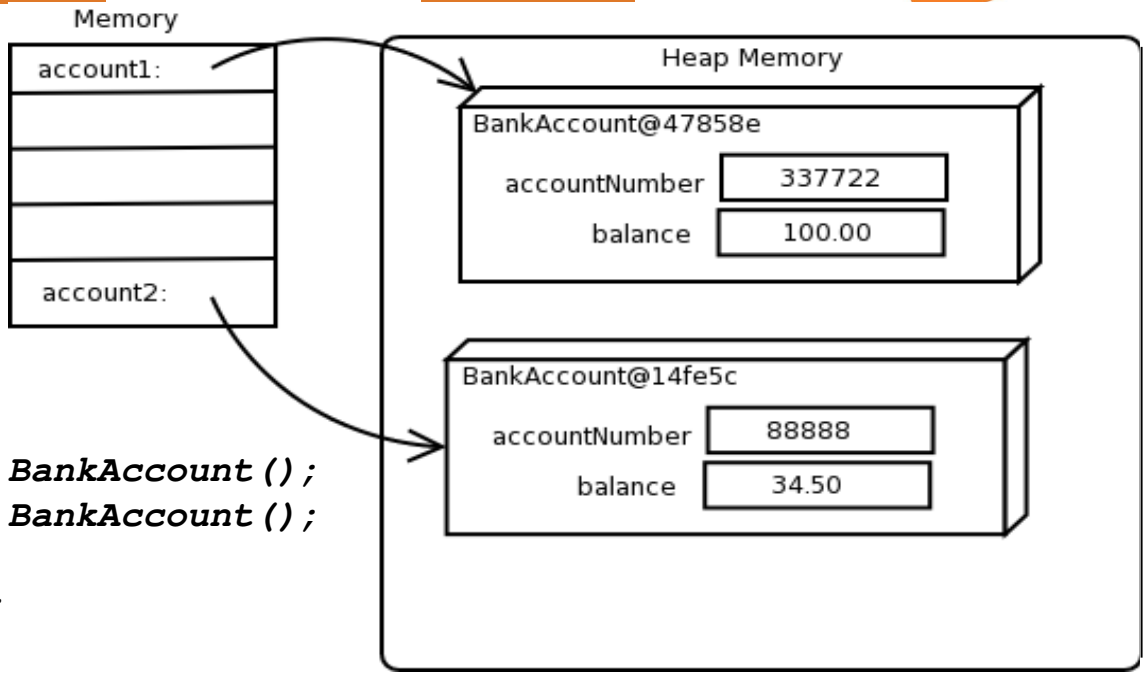


Fig. 5-4: **Output from example 5-3**

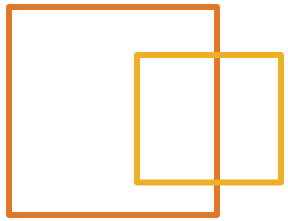
Referencing Instance Variables




```
BankAccount account1 = new BankAccount();
BankAccount account2 = new BankAccount();
account1.balance= 34.50F;
account2.balance = 100.00F;
account1.acNum = "888888";
account2.acNum = "337722";
```

Objects at the end of example 5.3

Instance Methods



- Perform some functionality of the object
- Commonly associated with underlying instance variables
- Simple instance methods might be:
    - Accessors – retrieve values
    - Mutators – change values
- Instance methods are “initialized” as part of class loading
    - Unlike instance variables, instance methods are shared by all instances of a specific class (they’re “execute-only”)
    - When an instance method is invoked, it is invoked on a specific object
    - Sharing the definitions is like three chefs working from one recipe book, there’s no reason to duplicate methods in memory

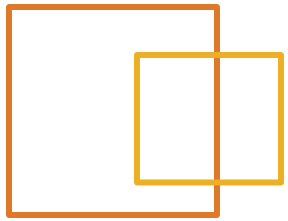
Instance Methods

- The instance methods are invoked using the dot-notation
  objectVariable.setVariableName(xxx);
  objectVariable.methodName(arg1,arg2..);
- Remember, since they are associated with an instance, the instance must exist before calling a method
  - To call the method, ***reference.method()***
  - Inside the method, keyword ***this*** refers to the current reference (what was ***reference*** on the outside)
- Instance methods follow the method syntax we discussed earlier

```
<access_modifier> <return> <identifier>(<parameter list>)
float deposit(float amt)
```

We will cover access modifiers later

Instance Method Example

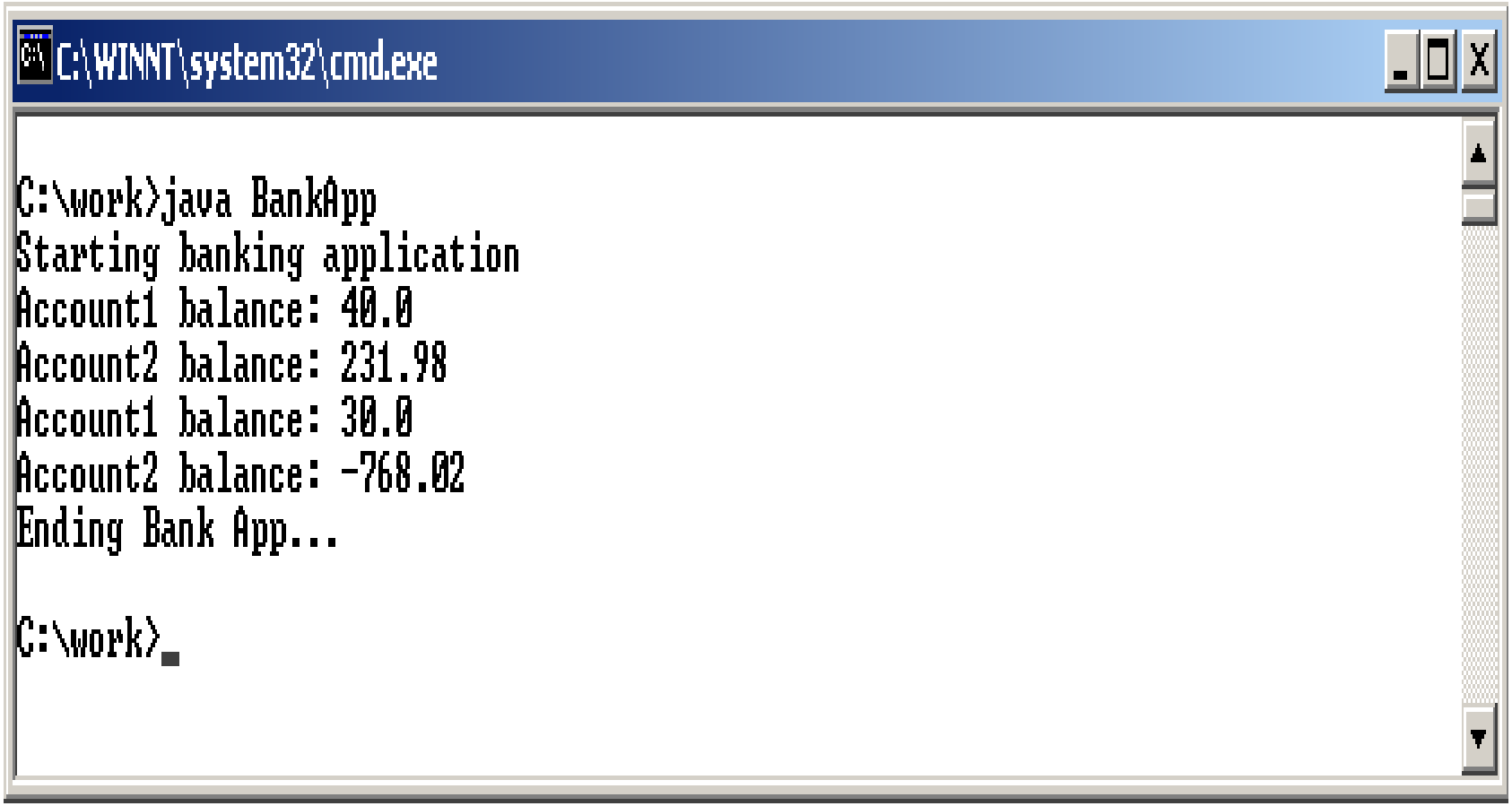


```
class BankAccount {
  float balance;
  String accountNumber;

  float queryBalance()
  {
    return balance;
  }

  float deposit(float amt) {
    balance = balance + amt;
    return balance;
  }

  float withdraw (float amt) {
    balance = balance - amt;
    return balance;
  }
}
```

Instance Method Example (cont.)

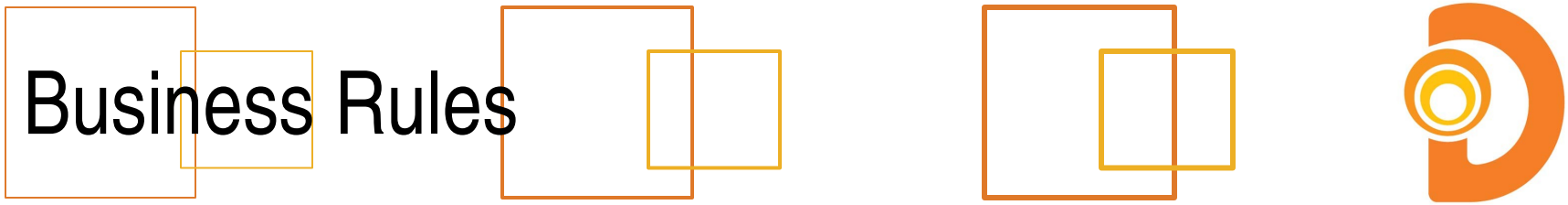
```
class BankApp {
  public static void main(String [] args) {
    System.out.println("Starting banking application");
    // create two new bank accounts.
    BankAccount account1 = new BankAccount();
    BankAccount account2 = new BankAccount();
    // Make deposits into each
    account1.deposit(40.00F);
    account2.deposit(231.98F);
    // Display their balances
    System.out.println("Account1 balance: " + account1.queryBalance());
    System.out.println("Account2 balance: " + account2.queryBalance());
    // Make a withdrawal from each account
    account1.withdraw(10.00F);
    account2.withdraw(1000.00F);
    // Display their balances
    System.out.println("Account1 balance: " + account1.queryBalance());
    System.out.println("Account2 balance: " + account2.queryBalance());
    System.out.println("Ending Bank App...");
  }
}
```

Instance Method Example Output



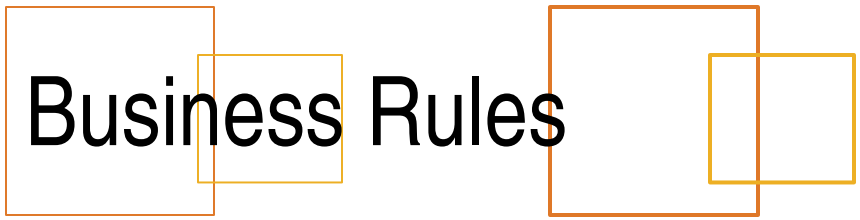
Output from example

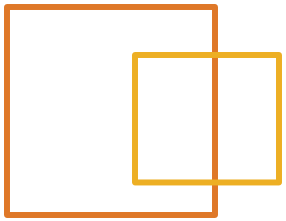
Business Rules



- Policies, procedures, and workflows automated by the application are commonly referred to as business rules
    - It is important to ensure objects in the application conform to those rules
    - Business rules are represented in program logic
- Business rules can also place execution and environment constraints on an application
- It should not be the programmer's job to determine business rules
    - Domain experts are responsible for defining the rules

Business Rules





- Methods implement business rules
- Some business rules are called guards
    - They guard against a method executing improperly
- OOP suggests incorporation of:
    - Preconditions: boolean conditions that must be true before the method can be executed
    - Postcondiitions: boolean conditions that must be true after a method executes
    - Invariant conditions: boolean conditions that must always be true

Business Rules and Methods

```
class BankAccount {
  float balance;
  String accountNumber;
  int accountStatus;

  float deposit(float amt) {
     if (accountStatus != 0) {
        return 0.0F;
     }
     balance = balance + amt;
     return balance;
  }

  float withdraw (float amt) {
     if (accountStatus != 0) {
         return 0.0F;
     }
     if (amt <= balance) {
         balance = balance - amt;
     }
     return balance;
   }
 }
```

Initialization of Instance Variables

- All variables must be initialized before they can be read
  - This rule is strictly and obviously enforced with local variables
  - This rule is not obvious with instance variables, because instance variables are always initialized
    - Implicit initialization sets value to “zero”
    - You can provide more explicit initialization

Initialization of Instance Variables

- Default initialization
  - Unavoidable/automatic
- Initialize in the declaration
  - `int x = 99;`
  - Referred to as “explicit initialization”
- Initialize a variable in an initializer
  - Code in an unnamed, unlabeled block
- Initialize a variable in the constructor
  - Allows arguments to constructor to be used to calculate initializing value

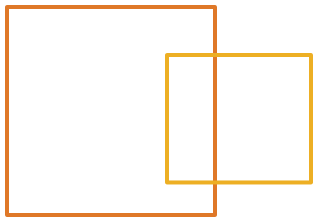
Initialization of Instance Variables (cont.)

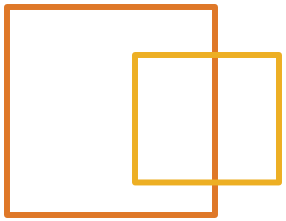
- Initialization mechanism have different results
- Default initialization
  - Reference variables, including ***`String`*** variables, are all initialized to ***`null`***
  - Numeric values are initialized to the appropriate zero value and
  - ***`boolean`*** types are initialized to ***`false`***
- Explicit initialization
  - Variables initialized to some specific value
  - Value may be computed, but limited data are available for this computation
  - Default initialization is overwritten
- Initializer/Constructor initialization
  - Initializer follows explicit, constructor follows initializer
  - Can overwrite previous initializations
  - Values may be computed, constructor args may be used
- Normally, instance variables are initialized in the constructor

Explicit Initialization Example

```
class BankAccount {
  float balance = -1.0F;
  String accountNumber = "NotSet";
  int accountStatus = -1;
  char accountType = " "
}
```

Constructors





- Objects are creating through a ***`new SomeType()`*** call
- After the memory has been created, the object is initialized in the constructor
- Constructors may be thought of as initialization methods
    - Note there's no return value

Constructor Purpose

- The purpose of the constructor is to initialize the newly created
  - Allows correct instance variable initialization
  - Any other initialization or startup code can be executed
  - Perform complex initialization logic that can not be done as an explicit initialization, e.g. loops
- Constructors allow us to call new
  - What follows the keyword “new” must match the signature of a constructor
- If you provide no constructors, compiler provides one
  - Referred to as the default constructor
  - No arguments
  - Compiler doesn’t know about your class’ semantics, so there is no behavior in the default constructor

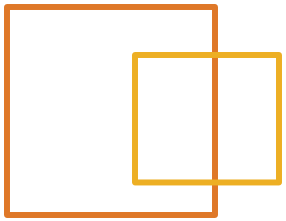
Constructor Rules

- Constructors must abide by some specific rules
- The constructor *always*
  - Has the same name as the class
  - Remember Java is case sensitive
- Constructors can be overloaded
  - Similar to method overloading
  - May be multiple constructors with different argument lists
- Does not declare a return value
  - This is not the same as returning **`void`**
  - A constructor initializes the newly created object

Constructor Example

```
class BankAccount {
    float balance = -1.0F;
    String accountNumber = "NotSet";
    int accountStatus = -1;
    char accountType = " ";

    BankAccount(String num, char type) {
      accountNumber = num;
      accountType = type;
      balance = 0.0F;
      accountStatus = (type == 'p')? 100: 0;
    }
    BankAccount(String num, char type, float bal) {
      accountNumber = num;
      accountType = type;
      balance = bal;
      accountStatus = (type == 'p')? 100: 0;
    }
    BankAccount(String num) {
      accountNumber = num;
    }
    /* -- rest of class -- */
  }
```

Proper Constructor Form

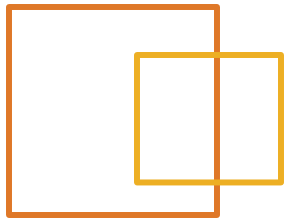


- Typically a class defines multiple constructors
  - Each constructor varies by argument list
  - Though the constructors are different, they should perform the same level of initialization
- Having many constructors
  - Provides flexibility
  - Can be error prone if done wrong
- Constructors can refer to other constructors
  - To minimize redundant code
  - Provide centralized initialization
  - Simplify maintenance

Proper Constructor Form (cont.)

- When referring to other constructors
  - Utilize a built-in mechanism - ***`this(…)`***
  - Think of ***`this(…)`*** as constructor calling another constructor
  - Like a method call
  - JVM determines which constructor to call
- Use ***`this(…)`***
  - As the first execution in your constructor
  - Can perform other operations once ***`this(…)`*** “returns”

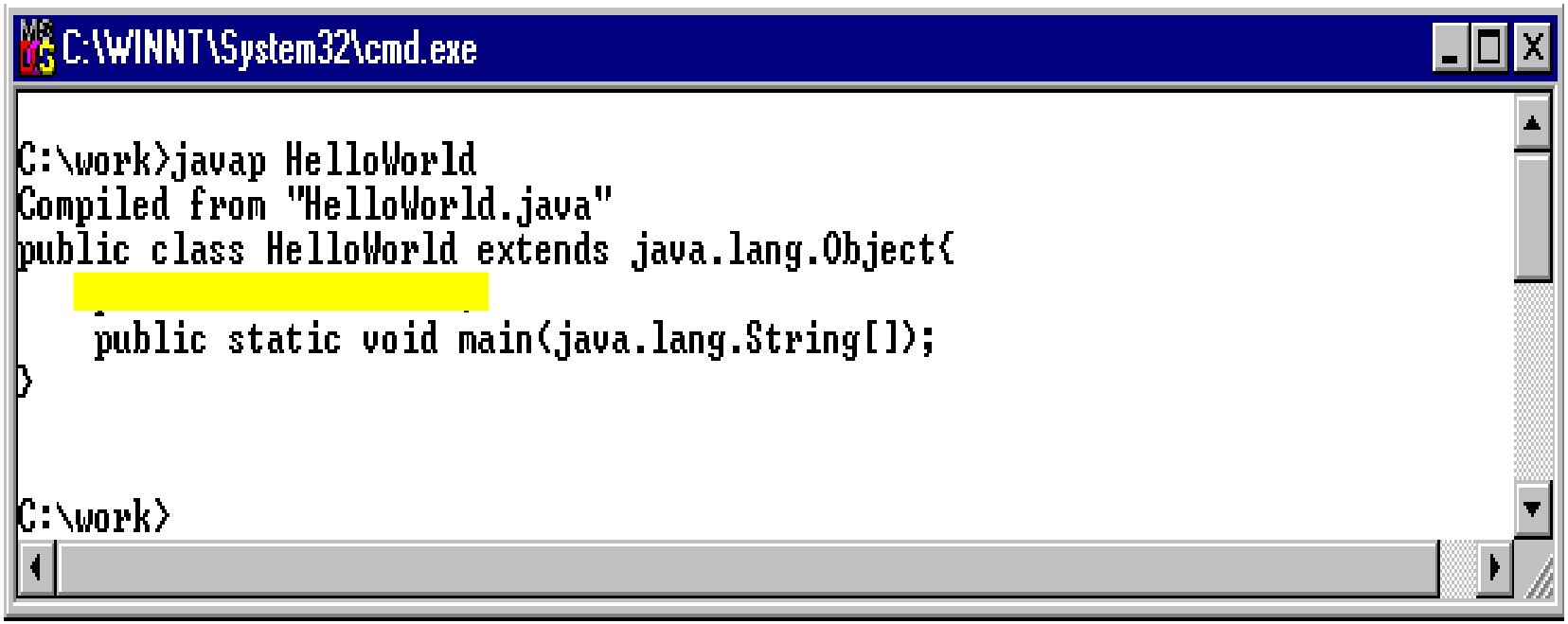
Proper Constructor Form



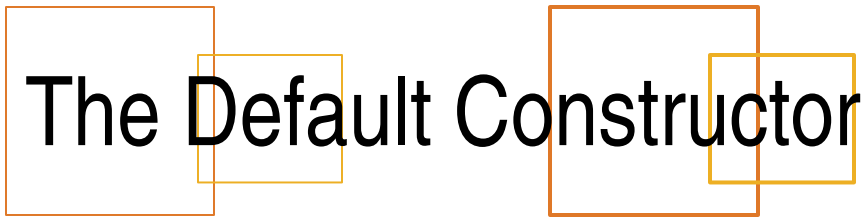
```
class BankAccount {
  float balance = -1.0F;
  String accountNumber = "NotSet";
  int accountStatus = -1;
  char accountType = " ";

  BankAccount(String num, char type, float bal) {
    accountNumber = num;
    accountType = type;
    balance = bal;
    accountStatus = (type == 'p')? 100: 0;
  }
  BankAccount(String num, char type) {
    this(num, type, 0.0F);
  }
  BankAccount(String num) {
    this (num, 'p');
  }
  /* -- rest of class -- */
}
```

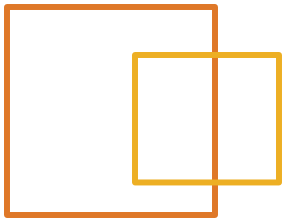
The Default Constructor

In the first module, we used the disassembler (***javap***) to look into our ***HelloWorld*** class



The Default Constructor





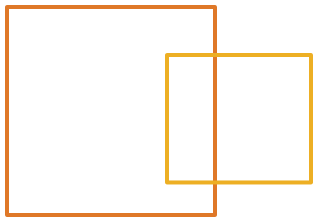
```
//This compile and runs
class Test1 {
  int x;
  public static void main(String [] args) {
    Test1 t = new Test1();  // this is the default constructor
    System.out.println(t);
  }
}

//This does not compile
class Test2 {
  int x;
  // Adding this constructor prevents the default
  // constructor is not provided
  Test2(int xs) {
    x = xs;
  }
  public static void main(String [] args) {
    Test2 t = new Test2();  // this constructor no longer exists.
    System.out.println(t);
  }
}
```

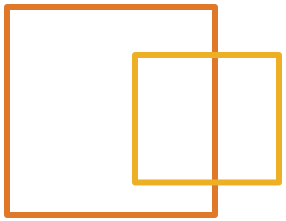
The Instance Initializer

- Initializers are invoked prior to constructors
- Multiple initializers are permitted, they're executed from top to bottom of the class
- The syntax is simply an unadorned block in the class

```
class ThingOne {
  int someNumber;
  {
    someNumber = (int)(Math.random() * 1000);
    if (Math.random() > 0.9) { someNumber = 0; }
  }
  /* Rest of class definition */
}
```

Summary

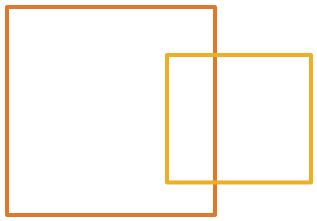


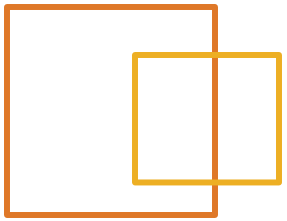


We covered

- Using the ***new*** operator to create objects
- Describing how reference variables work
- Using instance variables and methods
- Describing and using constructors

Lab 3

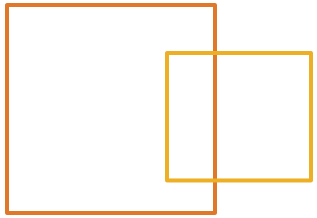


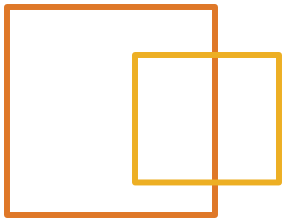


Simple Objects

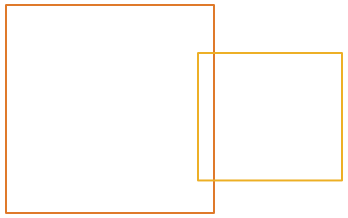
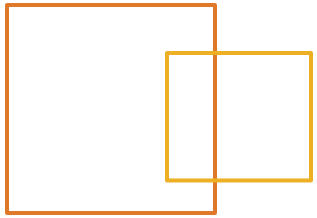
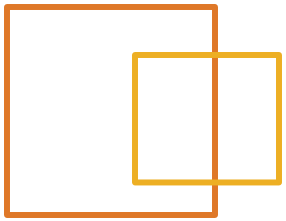
- Create a class that describes a Person. Pay attention to Java's coding conventions. The class should have fields for first name, last name, gender and birthdate (day and month).
- Provide a constructor that initializes first name, last name, and gender, and a second that initializes all the fields.
- Create a method that gets the person's full name (first and last name concatenated), and another method that returns the prefix "Mr" or "Ms" depending on gender.

Lab 3



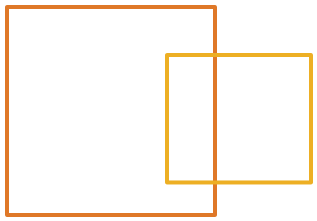


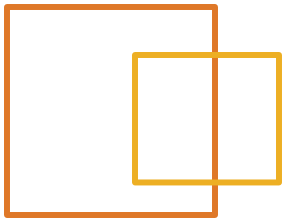
- Simple Objects (contd)
  - Create a third method that gets the "formal address" of the person by concatenating the prefix and the full name.
  - Create a method that takes arguments for day and month, and indicates if the day given is the person's birthday.
  - Create an “application” class with a main method that exercises the Person class by creating an array of four Person objects with different names and genders, and print out the formal addresses for each.
  - Solution: SimpleObjectLab


Objects And Classes (Part 2)

Objectives





At the end of this section, you should be able to:

- Using ***`public`*** and ***`private`*** access modifiers
- Class methods and variables
- The ***`final`*** keyword with variables
- Using ***`String`*** and ***`StringBuffer`*** objects
- Using arrays
- Using wrapper classes, autoboxing

Object Reference Semantics

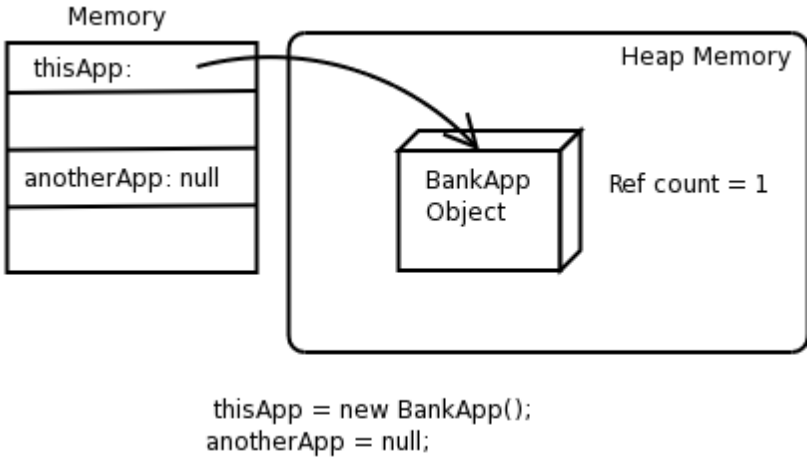
- Objects are accessed using references
- References are variables contains a “pointer” to an object
- A number, often a 32-bit integer, identifying the heap location of your object
- The reference value is hidden from you
- Copying a reference value only copies the reference value
- You do not actually copy the underlying object
- The end result is two references with the same value, referring to the same object in the heap

Object Reference Semantics Example

Example 5-11: Reference variable assignment

```
class BankApp {
  public static void main(String [] args) {
    // Declare the reference variables
    BankApp thisApp = null;
    Bankapp anotherApp = null;
  // Create the object and assign the reference value
    thisApp = new BankApp();
    // Now assignment of reference variables
    anotherApp = thisApp;
  }
}
```

Object Reference Semantics



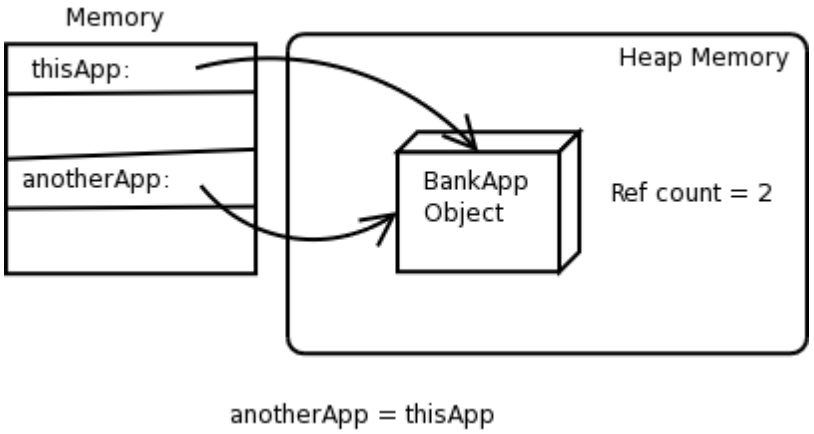


Figure 5-8: **Reference Semantics of a variable assignment.**

Object Life Cycle & Garbage Collection

- Java provides built in memory management
- Used for allocating memory
- Used for de-allocating memory – a.k.a. garbage collector
- The garbage collector (*gc*) is a daemon (background) thread that runs in the virtual machine
- There are many different types of garbage collection algorithms; in general the gc check on a regular basis which objects have become moribund

Object Life Cycle & Garbage Collection (cont.)

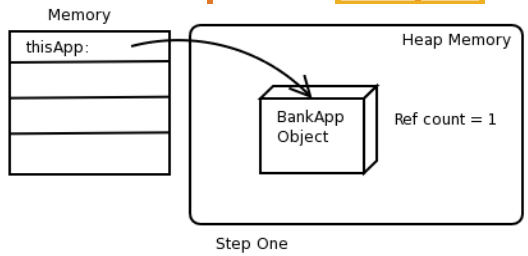
- The garbage collector frees up occupied but unreferenced memory automatically
    - An object referenced as long as there is at least one usable reference to it
    - As soon as zero accessible references exist, then the object can be garbage collected
- Objects can lose accessible references when:
    - Their reference variables become ***`null`*** (through assignment)
    - Their reference variables are reassigned with a new value
    - Their reference variables go out of scope (local reference variables)

Object Life Cycle & Garbage Collection Example

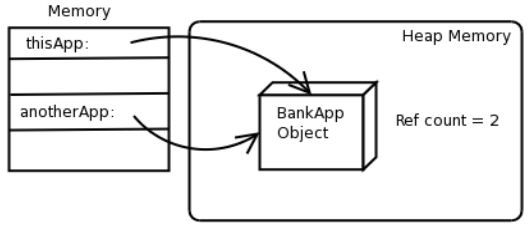
```
class BankApp {
  public static void main(String [] args) {
    // thisApp is now a local variable to this block
    {
      BankApp thisApp = new BankApp();
      // BankApp object now has a reference count of 1
      {
        BankApp anotherApp = thisApp;
        // BankApp object now has a reference count of 2
        // Step two in fig. 5-9
      }
      // anotherApp is out of scope. Reference count is 1 again
      // Step three in fig. 5-9
    }
    // thisApp is now out of scope, Reference count is 0
    // BankApp object is moribund waiting for garbage collection

    // Step four in fig. 5-9
  } // end of main method
}
```

Object Life Cycle & Garbage Collection

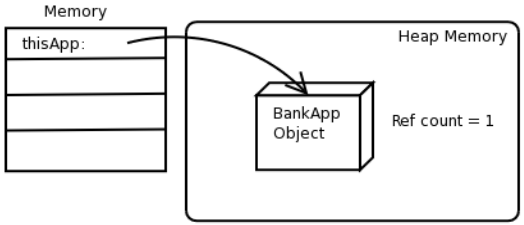
```
{
  BankApp thisApp = new BankApp();

      {
        BankApp anotherApp = thisApp;

      }

}
```



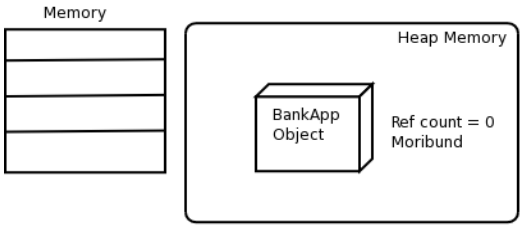
Step One



Step Two



Step Three



Step Four

The finalize() Method

- Before an object is garbage collected, the JVM system calls its ***`finalize()`*** method
- The ***`finalize()`*** method gives the object a chance to return allocated or in-use system resources
    - Kind of the opposite of a constructor
    - There is no guarantee that the ***`finalize()`*** method will actually be called
    - The original intent of the ***`finalize()`*** method was to free up resources allocated by native code, e.g. file handles, windows
- You can include a finalize method in your class

```
protected void finalize()
```

- *Finalize is generally not recommended*

finalize() Method Example

```
class BankApp {
  BankApp() {
    System.out.println("Creating BankApp");
  }
  protected void finalize() throws Throwable {
    System.out.println("Finalizing BankApp");
    return;
  }
  public static void main(String [] args) {
    new BankApp();
  }
}
```

Access Modifiers and Encapsulation

- Object Oriented Analysis and Design encourages the use of encapsulation
- Encapsulation is defined as data hiding
    - Think of encapsulation as a black box
    - Hide the “dirty” or sensitive details of objects
    - Prevents misuse
    - Thwarts “hacking”
    - Encapsulation should be applied to objects
- Objects are logical containers of
    - Data or state
    - Functionality or behavior

Access Modifiers and Encapsulation (cont.)

- An object's variables should be not be exposed outside the object
    - Instance variable exposure can allow direct variable access
    - This would circumvent any business rules you have in place
    - Would also allow object to obtain and corrupt sensitive data
- Sensitive and critical behaviors should also be hidden from other objects

Access Modifiers and Encapsulation (cont.)

- In Java, we use *access modifiers* to create encapsulation
- Access modifiers define a level of accessibility for classes
    - Class variables
    - Class methods
- Access modifiers define a level of accessibility for instances
    - Instance variables
    - Instance methods
    - Constructors
- Basic syntax is:
    - Variables

```
<access_modifier> Type identifier;
```

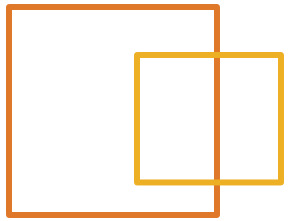
    - Methods

```
<access_modifier> <return_type> identifier(<parameter list>)
```

Access Modifiers and Encapsulation (cont.)

- Java has four access modifiers
  - **`private`** – only the class and object can access
  - *default* – only class, object, and subclass in same library (*package*)
  - **`protected`** – the class, object and subclasses in any package
  - **`public`** – any class, object, subclass
- We will cover access modifiers in more detail when we discuss packages
  - The default access modifier is automatically added if an access modifier is not explicitly specified

Private Variable Example



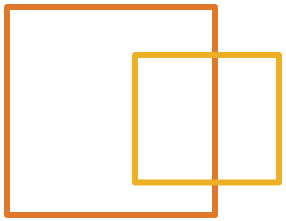
```
class BankAccount {
    // Instance Variable
    private float balance;
    // Constructor
    BankAccount(){
      balance = 0.0F;
    }
    // Instance Methods
    float queryBalance() {
      return balance;
    }
    float withdraw(float amt) {
      if ((amt > 0.0F) && (amt <= balance)) {
        balance -= amt;
      }
  return balance;
    }
    float deposit(float amt) {
      if (amt > 0.0F) {
        balance += amt;
      }
  return balance;
    }
 }
```

Private Variable Example (cont.)

```
class BankApp {
  public static void main(String [] args) {
    System.out.println("Starting banking application...");
    // Create a bank account
    BankAccount act = new BankAccount();
    act.deposit(100.00F);
    System.out.println("Balance is " + act.queryBalance());
    // This following line will not compile !!
    System.out.println("Balance is "+ act.balance);
    System.out.println("Ending banking application...");
  }
}
```

```
//producers the compiler error output
BankApp.java:9: balance has private access in BankAccount
System.out.println("Balance is "+ act.balance);
                                        ^
1 error
```

Private Variable Example

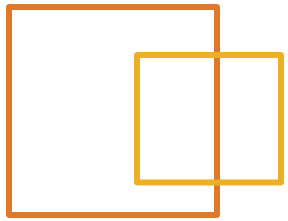


```
class BankAccount {
  // Instance Variables
  String accountNumber = null;
  char accountType;
  float balance;
  int accountStatus;
  // Instance Methods
  float queryBalance() {
    return balance;
  }
  float withdraw(float amt) {
    if ((amt > 0.0F) && (amt <= balance)) {
      if ((accountType == 's' && accountStatus == 100) ||
      (accountType == 'c' && accountStatus == 0)) {
    balance -= amt;
      }
    }
    return balance;
  }
. . .
```

Private Variable Example (cont.)

```
    float deposit(float amt) {
      if (amt > 0.0F) {
        if ((accountType == 's' && accountStatus == 100) ||
            (accountType == 'c' && accountStatus == 0)) {
      balance += amt;
    }
      }
      return balance;
    }
}
```

Private Method Example

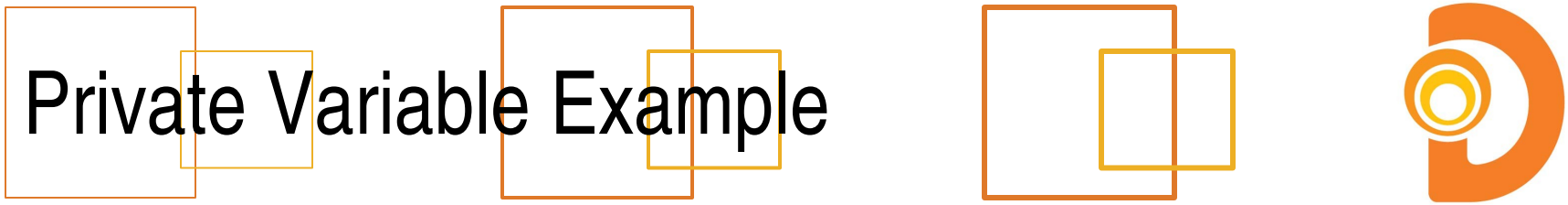


```
class BankAccount {
  // Instance Variables
  String accountNumber = null;
  char accountType;
  float balance;
  int accountStatus;
  // Private Instance Methods
  private boolean isAccountOK() {
    return ((accountType == 's' && accountStatus == 100) ||
       (accountType == 'c' && accountStatus == 0));
  }
  // Instance Methods
  float queryBalance() {
    return balance;
  }
. . .
```

Private Method Example

```
    float withdraw(float amt) {
      if ((amt > 0.0F) && (amt <= balance)) {
        if (isAccountOK()) {
          balance -= amt;
        }
      }
      return balance;
    }
float deposit(float amt) {
  if (amt > 0.0F) {
    if (isAccountOK()){
          balance += amt;
        }
   }
   return balance;
    }
}
```

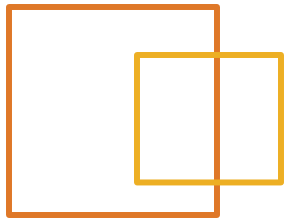
Private Variable Example



Private access is permitted between objects of the same type

```
class A {
  private int var = 0;
  void changeVar( A otherA) {
    otherA.var++;
  }
  public static void main(String [] args) {
    // create an two A objects
    A firstA = new A();
    A secondA = new A();
    // use the first A object to change the private data in
    // the second A object
    firstA.changeVar(secondA);
  }
}
```

Public Method Example



Public Access

```
class BankAccount {
  // Instance Variables
  String accountNumber = null;
  char accountType;
  private float balance;
  int accountStatus;
  // Private Instance Methods
  private boolean isAccountOK() {
    return ((accountType == 's' && accountStatus == 100) ||
         (accountType == 'c' && accountStatus == 0));
  }
  // Public Instance Methods
  public float queryBalance() {
    return balance;
  }
}
. . .
```

Public Method Example (cont.)

Public Access (continued)

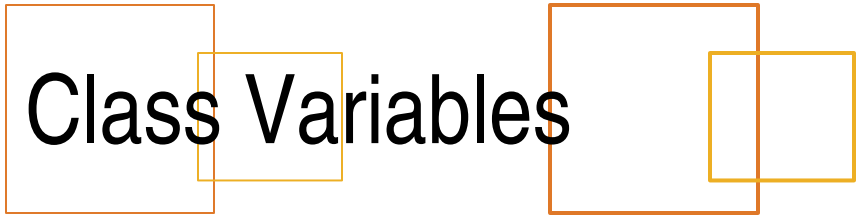
```
        public float withdraw(float amt) {
          if ((amt > 0.0F) && (amt <= balance)) {
            if (isAccountOK()) {
              balance -= amt;
            }
         }
          return balance;
        }
        public float deposit(float amt) {
          if (amt > 0.0F) {
           if (isAccountOK())balance += amt;
         }
          return balance;}
        }
}
```

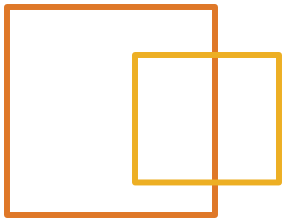
Class Variables and Methods

- Java provides a mechanism to declare variables that belong to the class as a whole not a specific object
  - Called ***static*** or class variables
  - Provide much of the functionality that was provided by global variables in structured languages
  - ***static*** variables belong to the class and are shared by all instances of the class
- Think of class or ***static*** variables as being variables that are global within a class
- Static variables can be accessed by either instance methods or static methods
- Use the dot-notation to access ***static*** variables

```
class_name.staticVariableName
BankAccount.NumberOfAccounts
```

Class Variables





- To make an instance variable static, you simply place the keyword static before the variable definition
- For example, the following defines a static data member and initializes it:

```
class BankAccount {
  // Class Variables
  static int NumberOfAccounts = 0;

  // Instance Variables
  String accountNumber = null;
  char accountType;
  private float balance;
  int accountStatus;
  // rest of class defintion
}
```

Referencing Static Variables

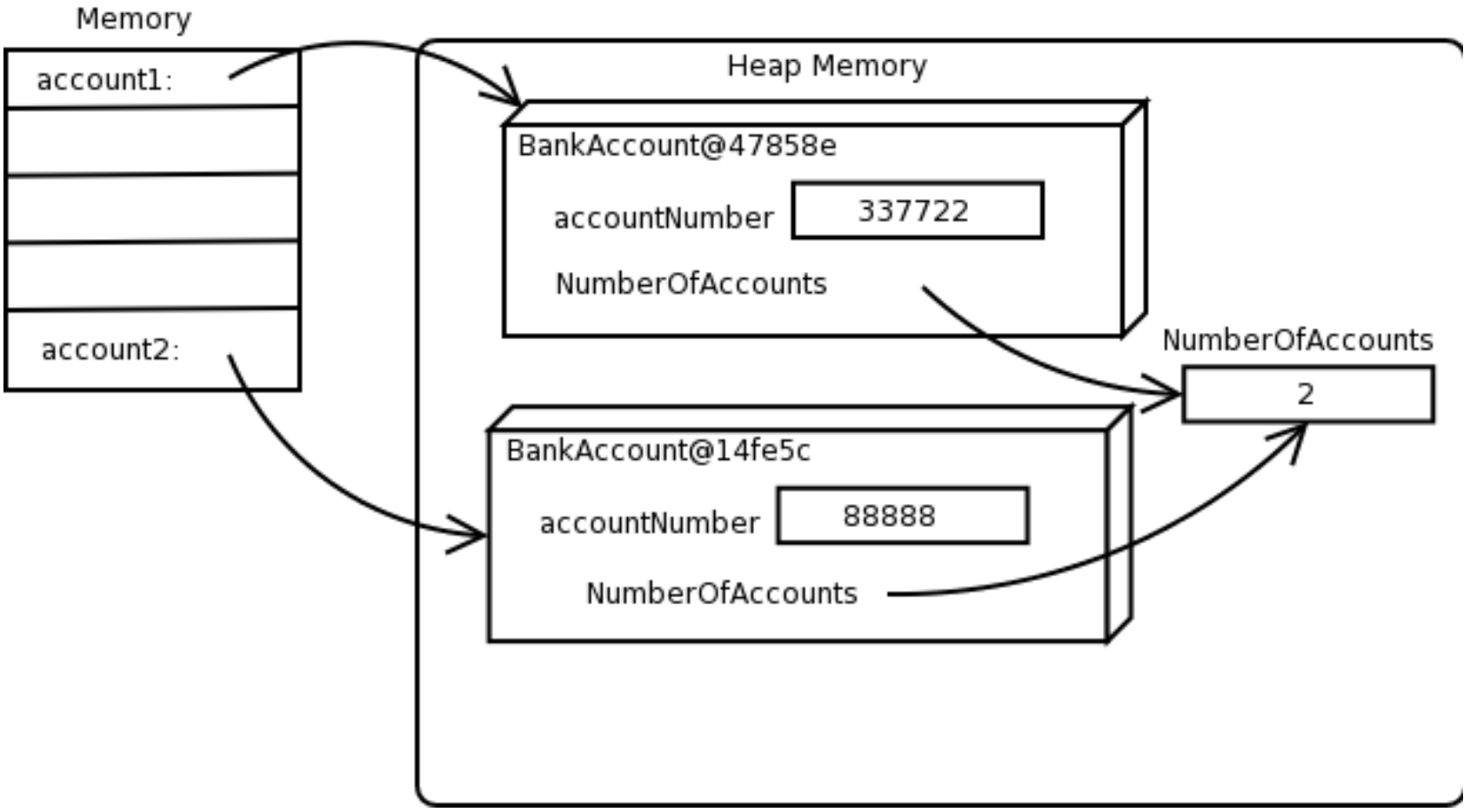
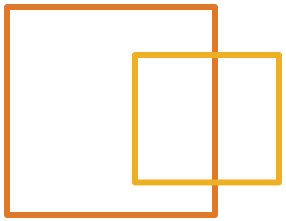


Fig 5-10: **Shared class variable NumberOfAccounts**

Referencing Static Variables

```
class BankAccount {
  // Class Variables
  static int NumberOfAccounts = 0;
// Instance Variables
  String accountNumber = null;
  char accountType;
  private float balance;
  int accountStatus;

// Using static variable in a constructor
  BankAccount(String num, char type, float bal) {
    accountNumber = num;
    accountType = type;
    balance = bal;
    accountStatus = (type == 's')? 100: 0;
    NumberOfAccounts++;
  }
}
```

Static Variable Initialization

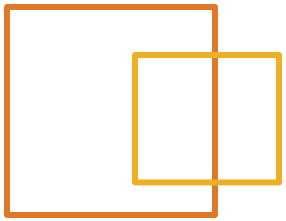


- There are three mechanisms in Java to initialize static variables
  - Default initialization – exactly like default instance variable initialization
  - Explicit initialization – exactly like explicit instance variable initialization
  - *Static Initializer* – similar to an initializer
- A static initializer
  - Initializes the static member variables of the class
  - Allows complex computations, such as loops and conditions
- A block of code, identified with the keyword ***`static`***, is allowed in the class definition
  - This block of code is executed when the ***`static`*** variables actually come into existence
  - The ***`static`*** block is intended to be used to initialize the ***`static`*** variables, nothing more
  - A ***`static`*** initializer is executed last, after default and explicit initialization of the class

Static Initialization Block

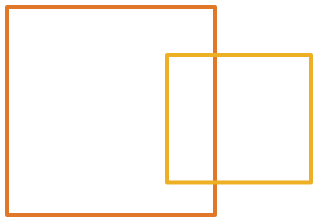
```
class BankAccount {
    // Class Variables
    static int NumberOfAccounts;
    // Instance Variables
    String accountNumber = null;
    char accountType;
    private float balance;
    int accountStatus;

    static { //static initializer
      NumberOfAccounts = 0;
    }

    // Using static variable in a constructor
    BankAccount(String num, char type, float bal) {
      accountNumber = num;
      accountType = type;
      balance = bal;
      accountStatus = (type == 's')? 100: 0;
      BankAccount.NumberOfAccounts++;
   }
}
```

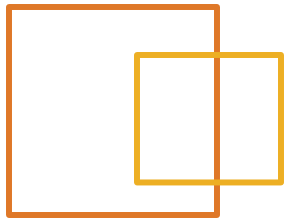
Static Methods



- Java allows methods that are associated with the class
  - They are not associated with any specific object
  - You can access the methods without an object
  - All ***`static`*** methods exist independently of any objects
- Static methods are typically used for
  - Library functionality (***`Math.abs()`***, ***`Integer.parseInt()`***)
  - Implementing certain design patterns (*Factory*, *Singleton*, etc)

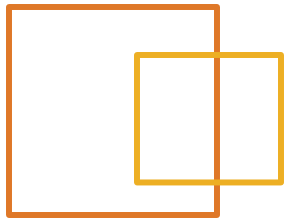
Static Methods





- There are some rules when dealing with ***`static`*** methods
  - There is no “current object” in a ***`static`*** method, therefore you cannot directly reference instance variables and methods inside a ***`static`*** method
  - Conversely, however, instance methods can access ***`static`*** variables and ***`static`*** methods
  - Static methods can access instance variables if a reference to an instance is used
- Static methods are accessed using the dot-notation
  - The reference variable becomes the class name
    - ***`Math.abs(-1234);`***
    - ***`BankAccount.incrementCount();`***
- It is also possible to use an instance as the reference variable, though it is consider poor programming style

Static Method Example



```
class BankAccount {
  // Class Variables
  private static int NumberOfAccounts;
  // Instance Variables
  String accountNumber = null;
  char accountType;
  private float balance;
  int accountStatus;
  static {
    NumberOfAccounts = 0;
  }
  // Class Methods
  private static void incrementCount() {
    BankAccount.NumberOfAccounts++;
  }
  public static int numActs() {
    return BankAccount.NumberOfAccounts;
  }
. . .
```

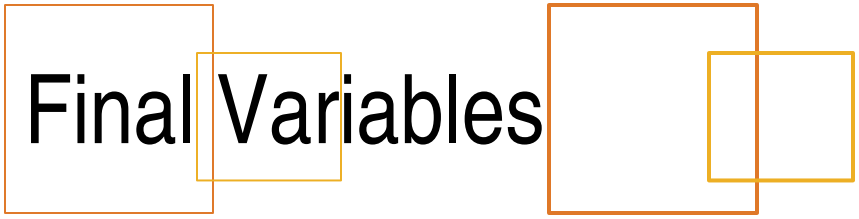
Static Method Example (cont.)

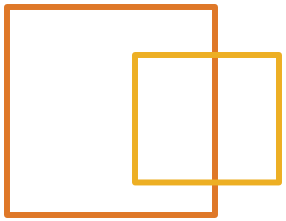
```
    // Using static variable in a method
    BankAccount(String num, char type, float bal) {
      accountNumber = num;
      accountType = type;
      balance = bal;
      accountStatus = (type == 's')? 100: 0;
      incrementCount();
    }
}
class Test{
  public static void main(String [] args) {
    BankAccount b = new BankAccount();
    System.out.println("Number of Accounts: "+ b.numActs());
    System.out.println("Number of Accounts: "+
    BankAccount.numActs());
  }
}
```

Static Access to Instance Example

```
class ThingTwo {
  int x; // instance variable
  static void whatsX() {
    // This would fail, doesn't know what x we mean:
    System.out.println("x is " + x);
    // This works fine:
    ThingTwo aThing = new ThingTwo();
    System.out.println("x is " + aThing.x);
  }
}
```

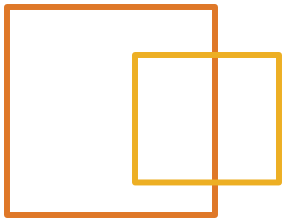
Final Variables





- Variables can be marked with the ***`final`*** keyword
- This indicates that once they are initialized, they cannot be changed
- Typically, there are two types of data that should be ***`final`***
    - Data representing a true constant (like the value of PI)
    - Data that is initialized once and never should be changed again during execution of the application
- Variables can also be ***`final`*** and ***`static`***
    - This means they are class wide constants
    - These variables are also typically ***`public`***
- So-called *blank finals* allow a ***`final`*** variable to be initialized in a constructor but then never changed again
    - Note that a blank ***`final`*** *must* be initialized *only* in a constructor, and must be initialized exactly once

Blank Finals Example



```
class BankAccount {
      private static int NumberOfAccounts;
      final String accountNumber;  // blank final
      final char accountType;      // blank final
      private float balance;
      int accountStatus;
      static {
        NumberOfAccounts = 0;
      }
      BankAccount(String num, char type, float bal) {
        accountNumber = num;
        accountType = type;
        balance = bal;
        accountStatus = (type == 's')? 100: 0;
        incrementCount();
      }
      /*--- more class code ---*/
    }
```

Comparing Reference Variables

- Reference variables contain a value which describes the location of an object in the heap
- This value is hidden from you
- If you want to compare the value of two references, you can use the equality operator
    - Use the standard **==** operator for this comparison
    - ***if(refVar1 == refVar2)***
    - ***if(refVar1 != refVar2)***

Comparing Reference Variables Example

```
class Test {
  public static void main(String [] args) {
    String s1 = new String("Hello");
    String s2 = new String("Hello");
    System.out.println
     ("Before assignment: s1 == s2 ->"+ (s1 == s2));
    String s3 = s1;
    System.out.println
     ("After assignment: s1 == s3 ->"+ (s1 == s3));
  }
}
```

Representing Text

Java has classes for dealing with strings of text

java.lang.String

java.lang.StringBuilder & java.lang.StringBuffer

String objects can be

Literals

```
String literal = "String Literal";
```

Objects

```
String object = new String("String Object");
```

The ***String*** class provides many methods

length

substring

toLowerCase / toUpperCase

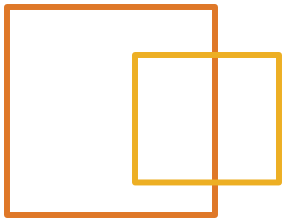
startsWith

Etc.

Strings, StringBuffers, StringBuilders

- ***`String`*** cannot be changed, which can be wasteful
  - Literals are kept around, and not garbage collected
  - Concatenation is costly
- StringBuffer and StringBuilder provide mutable text
  - They do not provide a literal mechanism
  - StringBuilder object = new StringBuilder(“StringBuilder”);
- No operator overloading for +
  - Use append(…) method.

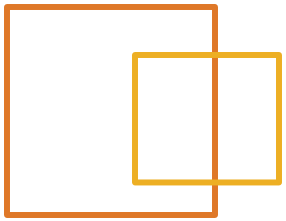
String Object Example



String class methods

```
class Test {
    public static void main(String [] args) {
        String s1 = new String("Hello");
        String s2 = s1.toUpperCase();
        String s3 = s1.toLowerCase();
        System.out.println("s1 -> "+ s1);
        System.out.println("s2 -> "+ s2);
        System.out.println("s3 -> "+ s3);
    }
```

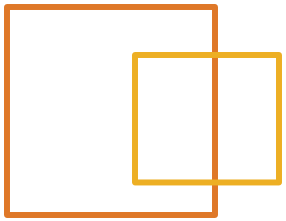
StringBuffer Example



StringBuilder used to reverse a String

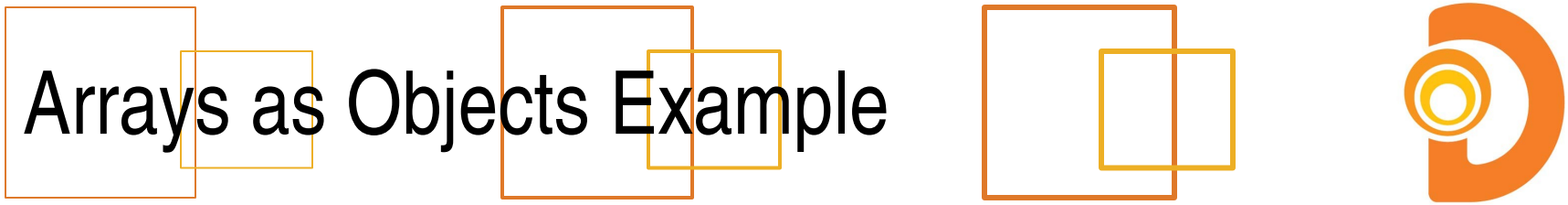
```
class ReverseString {
    public static String reverseString(String s) {
        // Use a String method to get length of String
        int size = s.length();
        StringBuilder sb = new StringBuilder();

        // Use the charAt() String method to get the character
        // from the String, and then use a StringBuilder method
        // to append it to the StringBuilder.
        for (int index = (size - 1); index >= 0; index--) {
            char c = s.charAt(index);
            sb.append(c);
        }

        // Convert the StringBuilder to a String
        return sb.toString();
    }
  }
```

Arrays as Objects



- Early we discussed arrays as
  - Basic “data structure”
  - With an inherent attributed called **`length`**
- Arrays in Java are objects
- Objects are created with the new keyword
  - Arrays have a literal form too
- Objects contain attributes; arrays contain attributes called elements
- There is no “constructor” when dealing with an array

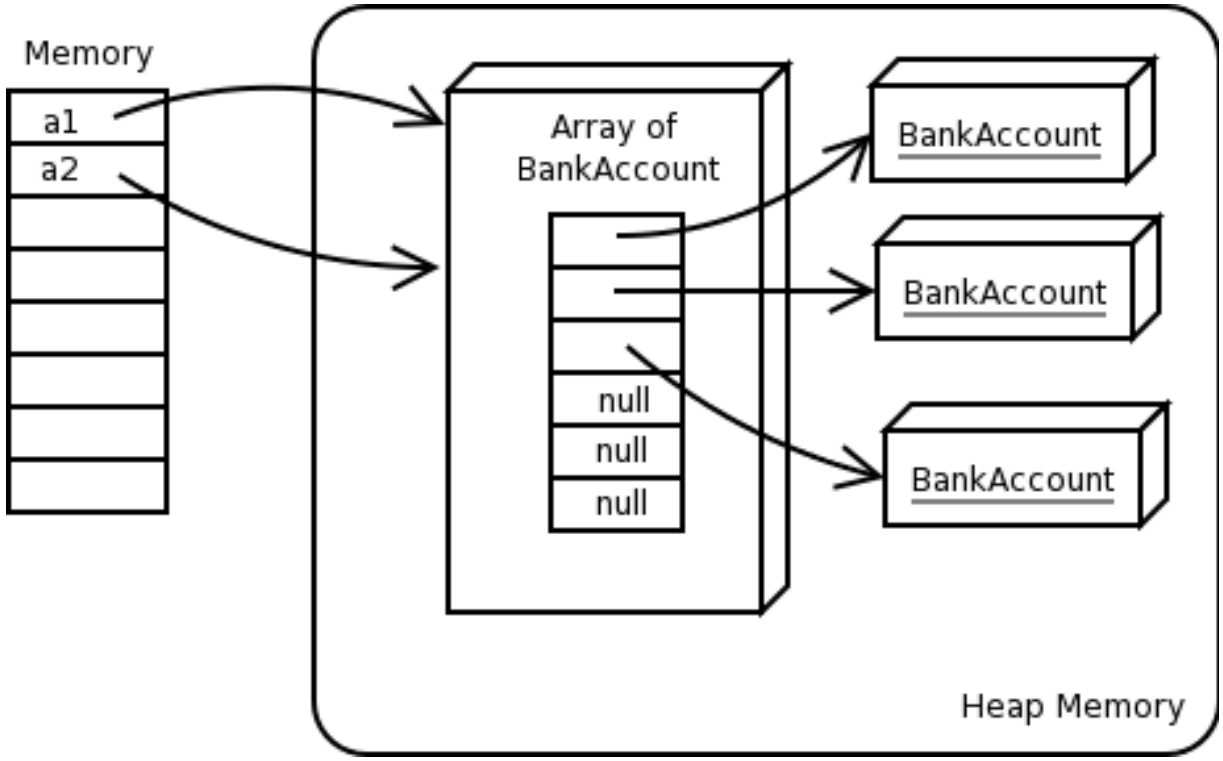
Arrays as Objects Example



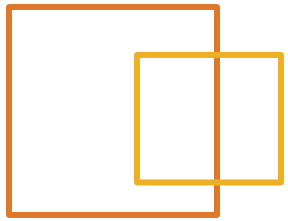
Creating arrays

```
class Test {
  public static void main(String [] args) {
    // create the array references
    BankAccount [] a1;
    BankAccount a2 [];
    // create an array
    a1 = new BankAccount[6];
    for (int k = 0; k < 3; k++) {
        a1[k] = new BankAccount();
    }
    // make a1 and a2 point to the same array
    a2 = a1;
}
```

Arrays as Objects



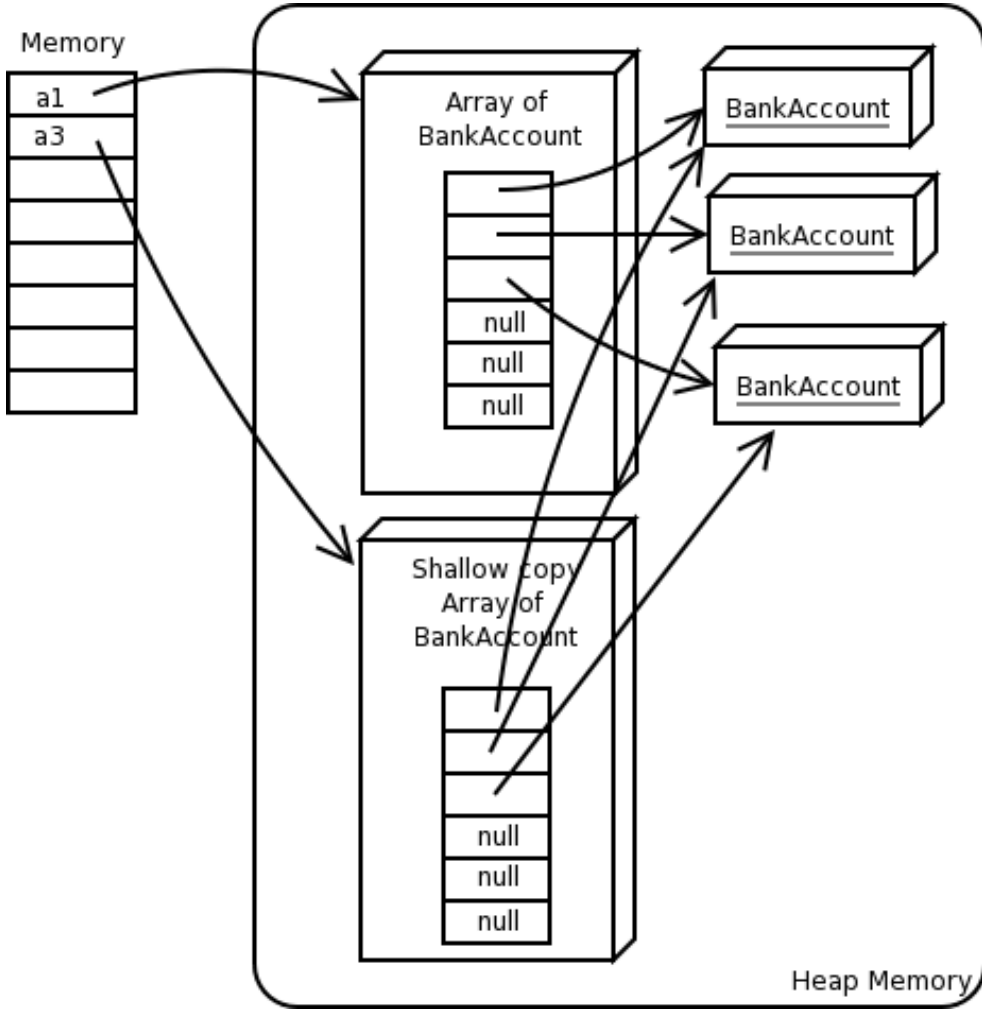
Copying an Array



Copying arrays

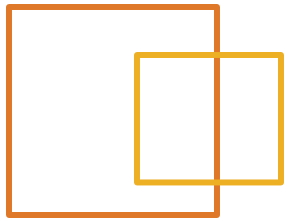
```
class Test {
  public static void main(String [] args) {
    // create the array references
    BankAccount [] a1;
    BankAccount a2[];
    BankAccount [] a3;
    // create an array
    a1 = new BankAccount[6];
    for (int k = 0; k < 3; k++) {
      a1[k] = new BankAccount();
    }
    // make a2 into a copy of a1 - manual/wrong way
    a2 = new BankAccount[a1.length];
    for (int k = 0; k < a1.length; k++) {
      a2[k] = a1[k];
    }
    a3 = new BankAccount[a1.length];
    // make a3 into a copy of a1 -- easy way
    System.arraycopy(a1,0,a3,0,a1.length);
  }
}
```

Copying an Array



Result of shallow copy

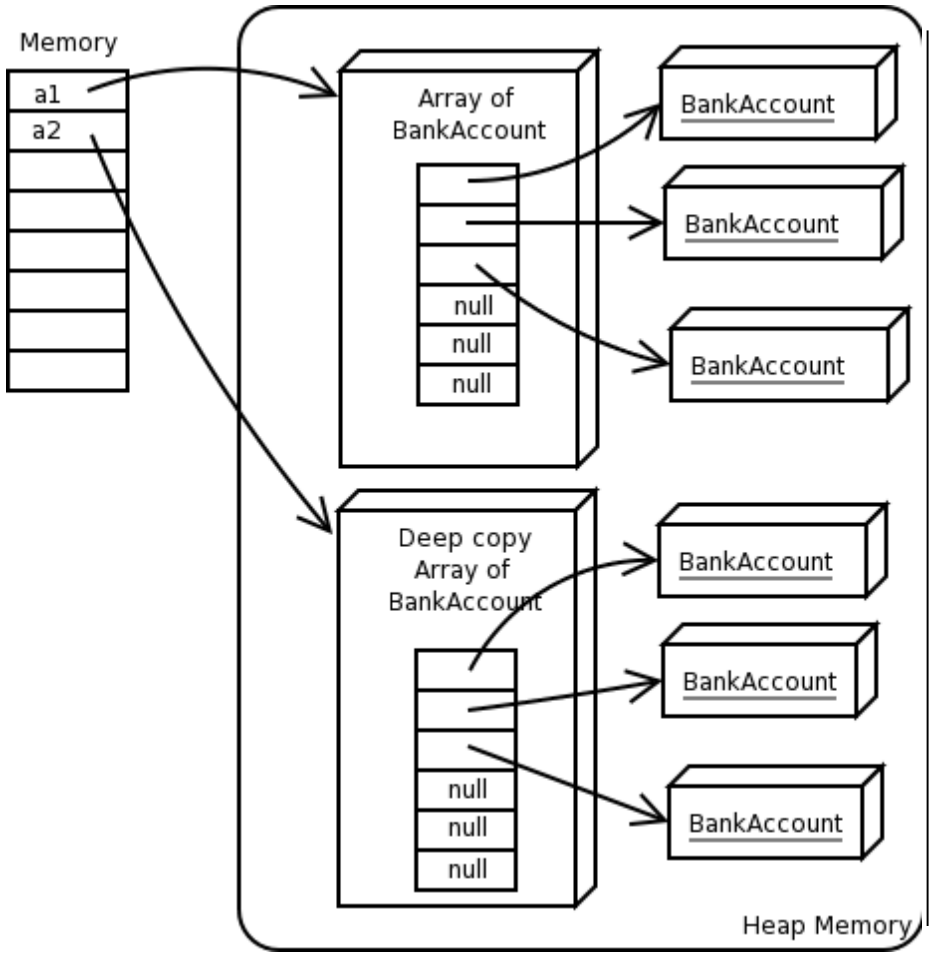
Deep Copying Example



Deep copying arrays

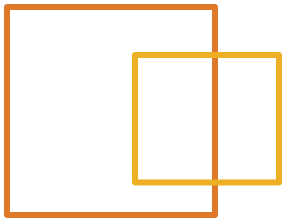
```
BankAccount a1 = new BankAccount[6];
for (int k = 0; k < 3; k++) {
  a1[k] = new BankAccount();
  // make a2 into a deep copy of a1 --
  a2 = new BankAccount[a1.length];
  for (int k = 0; k < a1.length; k++)
  {
    a2[k] = a1[k].clone();
  }
}
```

Deep Copying an Array



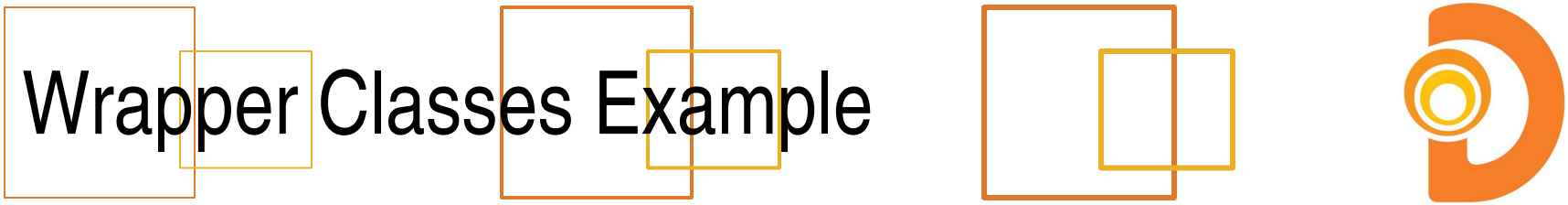
Result of deep copy

Wrapper Classes



- Sometimes it is useful to treat primitives as objects
- Java provides wrapper classes for all primitive types
    - ***`java.lang.Long`***
    - ***`java.lang.Integer`***
    - ***`java.lang.Short`***
    - ***`java.lang.Boolean`***
    - Etc.
- The wrapper classes provide some useful functionality like
    - Converting primitives to and from ***`String`***
    - Retrieving ***`System`*** properties as the primitive value

Wrapper Classes Example



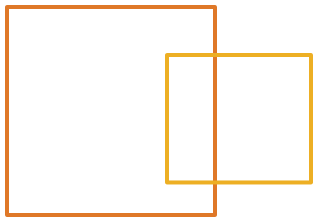
Wrapper classes

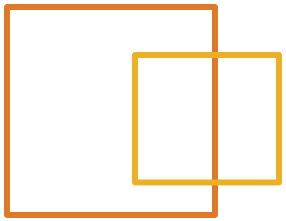
```
class Test {
  public static void main(String [] args) {
    String s = "7839276";
    long var = Long.parseLong(s);
    System.out.println("Value of var is "+var);
// Create a Long object to wrap this value
    Long obj = new Long(var);
// This is an object but we can still get the data
System.out.println("obj wraps "+ obj.longValue());
  }
}
```

Autoboxing/unboxing

- Since Java 1.5 assignment between primitives and their wrappers is transparent
    - // executes as x = new Integer(5);
      Integer x = 5;
    - // executes as y = new Integer(5).intValue();
      int y = new Integer(5);
- Take care, the computations still happen, which can be wasteful
- These features are particularly valuable with the collections API
    - Allows us to store primitives in containers made for objects

Summary

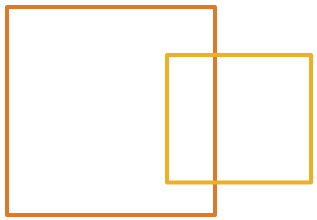


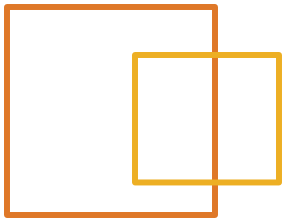


We covered

- Using ***`public`*** and ***`private`*** access modifiers
- Class methods and variables
- The ***`final`*** keyword with variables
- Using ***`String`*** and ***`StringBuffer`*** objects
- Using arrays
- Using wrapper classes, autoboxing

Lab 4





- Add an ID to the Person class
  - Make sure that your Person class properly encapsulates it's state variables.
  - Add an **id** property to the person class.
  - Create an automatic way to assign a unique id to every new Person object created. (Hint – use a **static** variable to hold the next ID)
  - Create two or more instances of your new Person class, and make sure that they all have unique **id**s.