COP 3330: Object-Oriented Programming Summer 2007

Introduction to Classes – Part 1

Instructor : Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 823-2790
http://www.cs.ucf.edu/courses/cop3330/sum2007

School of Electrical Engineering and Computer Science
University of Central Florida

Classes

- A **class** is a blueprint or template of its objects.
- We can create many objects from a single class.
- Creating an object from a class is called **instantiation**, and an **object** is an **instance** of a particular class.
- Normally, an object is created from a class using the `new` operator.
  - `new` *ClassName*`(` *parameters* `)`

Classes (cont.)

- *new InputStreamReader(System.in)*

  used to create an InputStreamReader object

- When a *new* operator is executed the constructor method of the class is activated to create an instance of that class, and that instance is initialized by the constructor method.
- The constructor method has same name as the class and does not have any return type.
- There are some short cuts to create objects of certain pre-defined classes in Java API.
  - String class: "abc" creates an object of String class
  - array classes: int[] x = {5,2,1};

    creates an int array object with size 3 and initializes that array.

Class Declaration

[ClassAccesibilityModifiers] [OtherClassModifiers]
  `class` ClassName
  [`extends` SuperClass]
  [`implements` $Interface_1$,...,$Interface_n$]
`{`
  ClassMemberDeclarations
`}`

Class Accessibility Modifiers

- There are three class accessibility modifiers:

`public`

A public class is accessible by any class.

`private`

A private class is accessible only the classes within the same file.

When no modifier is present, (default case) the class is accessible by all the classes within the same package. This accessibility modifier (no modifier) is known as **`package`** accessibility modifier.

Other Class Modifiers

`abstract`

- An *abstract* class contains abstract methods.
- An abstract method is an unimplemented method.

  (we will talk abstract classes later).

`final`

- A *final* class may not have sub-classes.

Class Member Declarations

- Inside of a class, we may declare the following class members:
  - **Fields –** data-variables declared in the class.
  - **Methods –** methods declared in the class.
  - **Constructors** – special methods to create objects of the class, and to initialize fields.
  - **Inner Classes --** classes nested in the class. An inner class cannot have any other inner class. This means that the nesting is only one level. (we are not going to talk too much about inner classes).

Class Member Declarations (cont.)

- The order of the declarations is not important, but it is nice to use the following order. Following this convention will make your program easier to read.

```
class ClassName {
        Fields
        Constructors
        Methods
}
```

Accessibility Modifiers for Class Members

- There are four accessibility modifiers for class members:

`public`

A public member is accessible by any class.

`private`

A private member is accessible only the class itself.

`protected`

A protected member is accessible by the class itself, all its sub-classes, and all the classes within the same package.

When no modifier is present, (by default) the member is accessible by all the classes within the same package. This accessibility modifier (no modifier) is known as **`package`** accessibility modifier.

Accessibility Modifiers for Class Members (cont.)

	`public`	`private`	`protected`	`package`
The class itself	yes	yes	yes	yes
Classes in the same package	yes	no	yes	yes
Sub-classes in a different package	yes	no	yes	no
Non-subclasses in a different package	yes	no	no	no

Fields

- Fields are also known as ***attributes***.
- Fields are the ***data-variables*** declared in that class.
- A data-variable can be:
  - an ***instance variable*** (declared without using keyword `static`), or
  - a ***class variable*** (declared using keyword `static`, it is also known as a static variable).

Fields (cont.)

- An instance variable lives in an object of that class, and each object of that class has its own copy of that variable.
- A static variable is a class-variable and there is only one copy for it. All instances of that class share that single copy.
- A field declared with a `final` modifier, it is a **constant** and its value cannot be changed.

Declarations of Fields

- A field declaration can be in the following form:

  [FieldModifiers] Type

  FieldName_1 [= Initializer_1], ... ,

  FieldName_n [= Initializer_n] ;

Examples:

```
public int a;
int b=1, c=2;
private double x;
protected int y;

private static int x;
public static int y;

public final int CONST1 = 5;
private static final int CONST2 = 6;
```


Methods

- A method can be:
  - an ***instance method*** (declared without using keyword `static`), or
  - a ***class method*** (declared using keyword `static`, it is also known as a static method).

- An instance method is associated with an object. If an instance method accesses an instance variable, it accesses the copy of that instance variable in the current object. It looks like that there are multiple copies of an instance method (one for each instance of that class).

Methods (cont.)

- A static method is a class-method and there is only one copy for it. All instances of that class share that single copy. A static method cannot access an instance variable or an instance method.

- If a method is declared with a **`final`** modifier, it cannot be overridden in the sub-classes of the class.

Method Declaration

- A method declaration can be in the following form:

  [MethodModifiers] ReturnType MethodName
  ([FormalParameterList]) { Statements }

Examples:

```
public int m1(int x) { ... }
public void m2(double x) { ... }
private void m3(int x, double y) { ... }
protected void m4() { ... }
int m5() { ... }
private final void m6() { ... }
public static int m7() { ... }
private static final m8() { ... }
```


Creating Objects

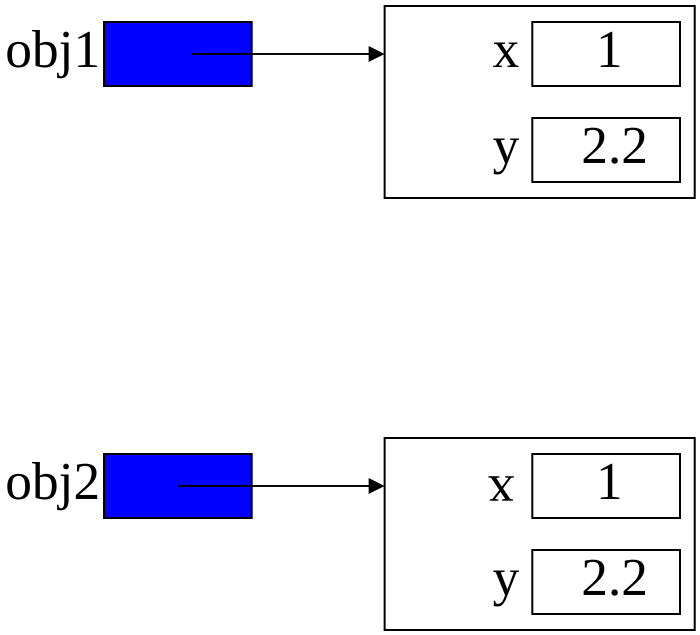
```
class C {
   // fields
   private int x;
   private double y;
   // constructors
   public C() { x=1; y=2.2; }
   // methods
   public void m1 (int val) { x=val; }
   public void m2 (double val) { y=val; }
}
```

- The constructor method must have the same name as the class, and it does not have any return type (not even void).
- Variables x and y are instance-variables, and they can be seen only by the methods of this class.

Creating Objects (cont.)

- In some other class, we may create the objects of the class C. (If we want, we can also create the objects of C in C too).

```
public class C2 {
   .... main (...) {
        C obj1, obj2;
        obj1 = new C();
        obj2 = new C();
   }
        .
        .
}
```



Dot Operator

- Once an object of a class is created, its instance methods can be invoked using the **dot operator**.
- Of course, the method which will be invoked must be accessible from that class.
- To invoke a method:

object.methodname(actual-parameters)

Example: (in a method of C2)

```
obj1.m1(4);
obj2.m1(3);
obj1.m2(3.3);
String s = stdin.readLine();
int a = Integer.parseInt(stdin.readLine().trim());
```


Dot Operator (cont.)

- Using dot operator, we may also access the fields of an object.
- To access a field: object.field

Example: (in a method of C2)

```
obj1.x = 4;
```
will not work, because x was private

if C is declared as follows, the above assignment will be okay.

```
class C {
   public int x;
        ...
}
```


Dot Operator (cont.)

- For static fields and methods, we can use the dot operator.
- We can access static fields and methods as follows:

  ClassName.FieldName

  ClassName.MethodName(ActualParameters)

- To access static members, we do not need to create an object from that class.

Dot Operator (cont.)

- We may also access static members using objects as follows.

  Object.FieldName

  Object.MethodName(ActualParameters)

- All the objects will access the single copy of a static member.

Dot Operator (cont.)

```
class C1 {
   public int x;
   public static int y=5;
   public C1() { x=1; }
   public void setX(int val) { x=val; }
   public static void printY() { System.out.println("y: " +
   y); }
}
```

```
// in a method of some other class
   C1 o1,o2;
   C1.y = 10;
   C1.x = 10;    ➔ ILLEGAL
   C1.printY();
   C1.setX(10); ➔ ILLEGAL
   o1 = new C1();
   o2 = new C1();
```

```
   o1.x = 2;
   o2.x = 3;
   o1.y = 4;
   o2.y = 5;
   C1.y = 6;
   o1.setX(7);
   o2.setX(8);
   o1.printY();
   o2.printY();
```

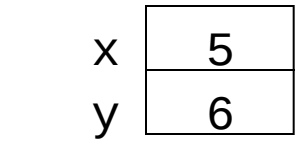

Another Example with Static Fields

```
class C {
   private static int count = 0;
   private int objIndex;
   public C() { count=count+1; objIndex=count; }
   public static int numOfObjs() { return count; }
   public int objID() { return objIndex; }
}
// in a method of some other class
   C o1,o2,o3;
   o1 = new C();
   o2 = new C();
   o3 = new C();
   System.out.println(o1.objID());
   System.out.println(o2.objID());
   System.out.println(o3.objID());
   C.numOfObjs();
```


Reference Assignment

- Assignment takes a copy of a value and stores it in a variable.

```
int x,y;
x=5;  y=6;  y=x;
```



before assignment

after assignment

```
public C {
   public int x,y;
   public C() {x=1;y=2;}
}

// in a method of another class
   C c1,c2;
   c1=new C();
   c2=c1;
// c1 and c2 will point to the same object
```


Aliases

- Two or more references may refer to the same object. They are called aliases of each other.
- Aliases can be useful, but they should be managed carefully.
- Affecting the object through one reference affects its all aliases, because they refer to the same object.

Example:

```
c1.x = 5;          //c2 is affected too.
```


Garbage Collection

- Objects are allocated on the heap (a part of memory space reserved for our programs to run).
- When an object no longer has any valid references to it, it can no longer be accessed by the program.
- In this case, it is useless, and it is called *garbage*.
- Java performs automatic *garbage collection* periodically to collect garbage for future use.

Super-Quick Review of Objects

- In C, you have typedef statement which allows you to give meaningful names to built-in types or to "compound" types.

```
typedef Money double;   //this is C not Java
typedef struct {
   int balance;
   char* name;
} Account;
```

- We can think of objects as structs. We create a new type every time we define a new class. And classes are just chunks of data (like structs)…but with behavior. In other words, we package data along with the code that operates on it.

Super-Quick Review: Creating an Object

- Assume we have a class called **`Money`**
- Then one could "instantiate" an object of type **`Money`** by calling

```
new Money();
```

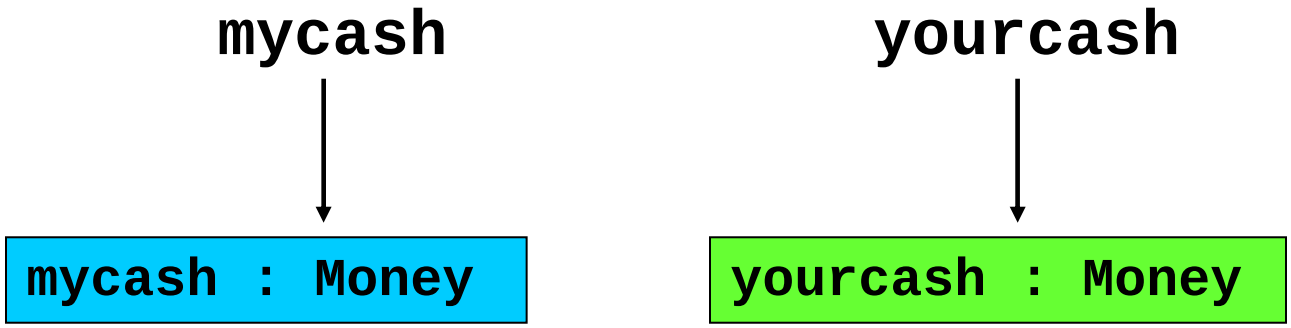
- But an instantiation by itself does not associate the newly constructed object with a name!
- The assignment operator (=) is usually used together with **new** to create a new object and give it a name.

```
cash = new Money();
```

- The right hand side of above statement creates object and assignment operator "binds" **`cash`** to that object.
- One can think of **`cash`** as a kind of pointer to an object of type **`Money`** that you don't have to de-reference.

Super Quick Review – Example

```
mycash = new Money();
yourcash = new Money();
```



Super Quick Review (cont.)

- We have to declare **mycash** somewhere before using it. What data type is **mycash**?

```
Money mycash;
```

- What value does **mycash** have after above statement is executed?
- Its value is **null** (a reserved word). Any "unbound" variable in Java is said to be **null**.
- What happens if one tries to send a message to an unbound (null) variable?
- Run-time error called **NullPointerException**

Declare – Instantiate – Bind

1. Declare the variable, specifying the class to which it belongs.

2. Instantiate an object with a class conforming to the variable.

3. Bind the variable (declared in step 1) to the object instantiated in step 2.

Variables

- In a Java program, we can access three kinds of variables in the methods.
  - **instance variables –** declared in the class (without using static keyword)
  - **class variables** (static variables) – declared in the class (with using static keyword)
  - **local variables** – declared in a method or as its formal parameters.

Variables (cont.)

- An instance method of a class can refer (just using their names) to all instance variables, all static variables declared in the class, and all its local variables.
- A static method of a class cannot refer to any instance variable declared in that class. It can only refer to static variables and its local variables.

Variables (cont.)

```
class C {
   int x;
   static int y;
   public void printX() { System.out.println("x: "+x); }
   public static void printY() { System.out.println("y: "+y); }
   public void m1(int a, int b) {
        int c=a+b;
        x=a; y=b;
        printX();  printY();
   }
   public static m2(int a, int b) {
        x=a;      ➔ ILLEGAL – static method refers to instance variable x.
        y=b;
        printX(); ➔ ILLEGAL – static method refers to instance method.
        printY();
  }
}
```


Methods

- A class contains methods.
- A method is a group of statements that are given a name.
- Each method will be associated with a particular class (or with an instance of that class).
- We may define methods and invoke them with different parameters. When its parameters are different, their behavior will be different.

Method Definition

- All methods follow the same syntax when they are defined:

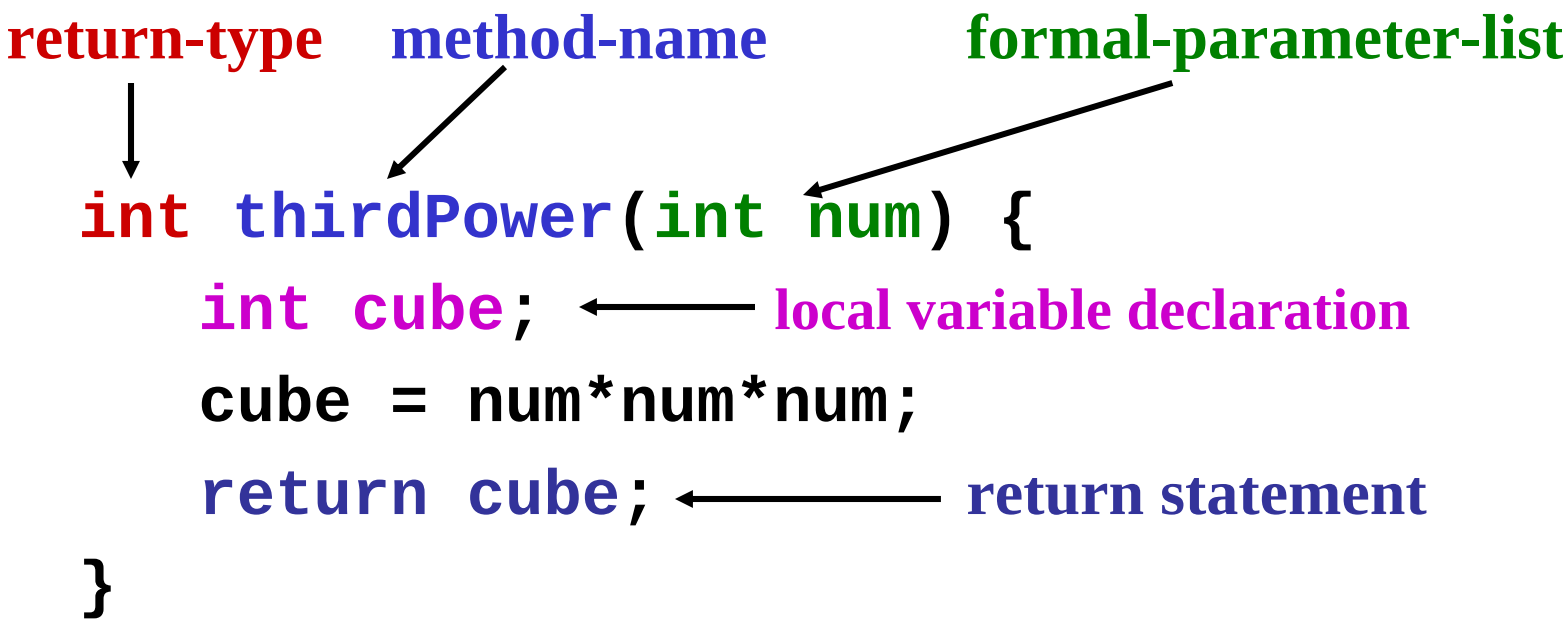
return-type method-name (***formal-parameter-list***) {

statements

}

- ***return-type*** indicates the type of the value returned from this method.
- ***method-name*** is the name of this method (an identifier),
- ***formal-parameter-list*** indicates:
  - how many parameters will be taken by this method
  - the names of the formal parameters (how do we use them in the inside of this method)
  - the data types of these parameters
- ***statements*** are the executable statements (and declarations for local variables) of this method. These statements will be executed when this method is invoked.

Method Definition -- Example



Return-Type and Return Statement

- The return type can be any data type (a primitive data type or an object data type) or `void`.
- A return statement will be in the following form:

```
return expression ;
```

- where the type of the expression must be the same as the return type of that method.

Return-Type and Return Statement (cont.)

- If the return type of a method is different from `void`, that method must contain at least one return statement.
  - When a return statement is executed in that method , we exit from that method by returning the value of the expression in that return statement.
  - Normally, method calls for these methods are parts of expressions.
- If the return type of a method is `void`, that method does not need to have a return statement.
  - In this case, when the last statement in that method is executed, we return from that method. (Or, it may contain a return statement without an expression `return;`)
  - Normally, a method call for the method with `void` return type is a statement.

NumberFormat Class (java.text)

This class is abstract which means that an object can not be instantiated using a new operator. You request an object from one of the methods invoked through class itself.

`String format(double number)`

returns a string containing the number in the format specified by the object

converts `double_n` *into string in accordance with* `objNumberFormat`

Example:

```
String s=objNumberFormat.format(double_n);
```

an instance of NumberFormat object

Methods in NumberFormat Class

•**Each of these methods return an object in the specified format.**

`static NumberFormat getCurrencyInstance()`

returns an instance of `NumberFormat` *object, that represents currency format*

`static NumberFormat getPercentInstance()`

returns an instance of `NumberFormat` *object, that represents percentage format*

Example Using NumberFormat

```
import java.text.NumberFormat;
public class Price
{
  double total=19.35;
  double tax_rate=0.06;
  NumberFormat money=

NumberFormat.getCurrencyInstance();
  String formatted_total= money.format(total);
  NumberFormat percent=
           NumberFormat.getPercentInstance();
  String
formatted_tax=percent.format(tax_rate);
}
```


DecimalFormat Class (java.txt)

`DecimalFormat (String pattern)`

constructor: creates a DecimalFormat object with specified pattern

argument of String type is sent to constructor

name of object

```
DecimalFormat fmt=new DecimalFormat("0.###");
```

type of object

an object is created

constructor is called

Method – applyPattern

```
void applyPattern (String pattern)
```

applies the specified pattern to the DecimalFormat object

```
fmt.applyPattern("0.####");
```


Method - format

`String format (double number)`

returns a string containing the number, formatted in accordance with the NumberFormat object

```
int radius=5;
double area=Math.PI*Math.pow(radius,2);
System.out.println(fmt.format(area));
```

Output: 78.5398

An Example – Designing A Class

- The following few pages will present a complete example of designing a class from the ground up for a specific application.
- Our application is a banking enterprise where we need to keep track of customer accounts.
- We will design a class called `accounts`.

Design of the class Account

- Variables to describe the state of an Account object:
  - name of the owner
  - account number
  - balance in the account
  - interest rate
- Data we are going to provide to create a new account
  - name of the owner
  - opening balance
  - account number
- Methods (service) provided for maintaining accounts
  - deposit an amount
  - withdraw an amount
  - add interest to the account
  - get name of the owner
  - get balance

```
// Developer:  Mark Llewellyn     Date: May 2007
// A banking account class

import java.text.NumberFormat;
public class Account{
  private NumberFormat fmt =
                      NumberFormat.getCurrencyInstance();
  private final double RATE = 0.045;
  private long acctNumber;
  private double balance;
  private String name;
//-----------------------------------------------
//  Sets up the account by defining its owner,
//  account number, and initial balance.
//-----------------------------------------------
  public Account (String owner, long account,
                      double initial)
  {
      name = owner;
      acctNumber = account;
      balance = initial;
  }
```


```
//-----------------------------------------------
//  Validates the transaction, then deposits
//  the specified amount into the account.
//  Returns the new balance.
//-----------------------------------------------
   public double deposit (double amount)   {
       if (amount < 0)  // deposit value is negative
       {
         System.out.println ();
         System.out.println ("Error: Deposit amount
                                  is invalid.");
         System.out.println (acctNumber + "  "
                                + fmt.format(amount));
       }
       else
         balance = balance + amount;
      return balance;
   }
```

balance *is private, but you can get it from outside through a public method called getBalance – see page 52.*

```
public double withdraw (double amount, double fee)   {
    amount += fee;
    if (amount < 0)  // withdraw value is negative
    {
        System.out.println();
        System.out.println("Error: Withdraw amount is invalid.");
        System.out.println("Account: " + acctNumber);
        System.out.println("Requested: " + fmt.format(amount));
    }
    else
        if (amount > balance)  // withdraw value exceeds balance
      {
            System.out.println ();
            System.out.println("Error: Insufficient funds.");
            System.out.println("Account: ” +acctNumber);
            System.out.println("Requested: "+fmt.format(amount));
            System.out.println("Available: "+fmt.format(balance));
      }
        else
            balance = balance - amount;
    return balance;
}
```

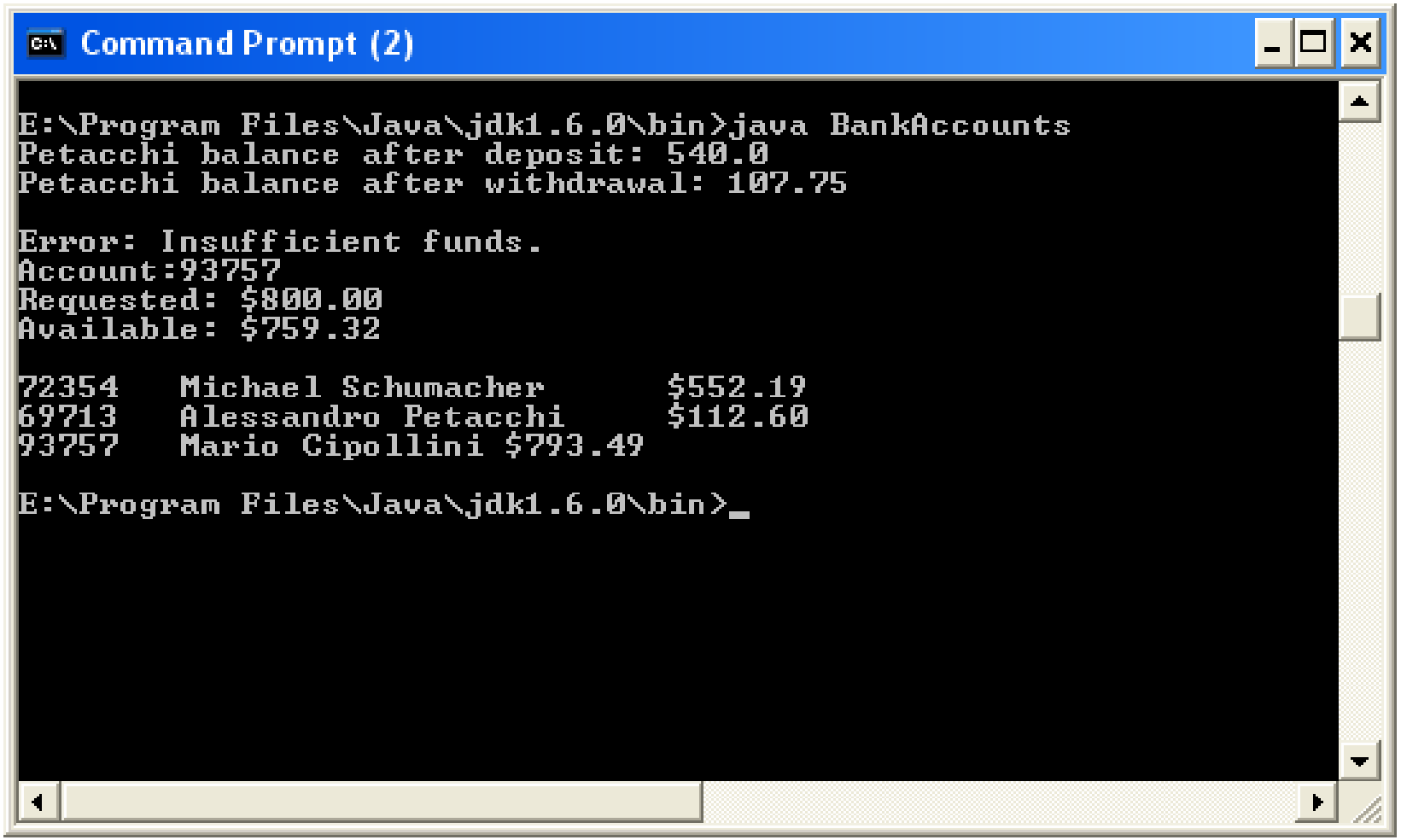

```
    public double addInterest ()   {
         balance += (balance * RATE);
         return balance;
    }

    public double getBalance ()   {
         return balance;
    }
    public long getAccountNumber ()   {
         return acctNumber;
    }

    public String toString ()   {
         return(acctNumber + "\t" + name +
                "\t" + fmt.format(balance));
    }
}// end class Account
```


The BankAccounts Class

```
import Account;

public class BankAccounts {
//Creates some bank accounts and requests various services.
  public static void main (String[] args)   {
       Account acct1=new Account(“Michael Schumacher",
                                          72354, 502.56);
        Account acct2=new Account(“Alessandro Petacchi",
                                          69713, 40.00);
        Account acct3=new Account(“Mario Cipollini",
                                          93757, 759.32);

        acct1.deposit (25.85);
        double petacchiBalance=acct2.deposit(500.00);
        System.out.println (“Petacchi balance after deposit:”
                                    + petacchiBalance);
        System.out.println (“Petacchi balance after withdrawal: ”
                                  + acct2.withdraw (430.75, 1.50));
```


```
        acct3.withdraw (800.00, 0.0);// exceeds balance
        acct1.addInterest();
        acct2.addInterest();
        acct3.addInterest();

        System.out.println ();
        System.out.println (acct1);
        System.out.println (acct2);
        System.out.println (acct3);
    } // end of main
}  // end of class BankAccount
```


Output of the BankAccounts Program



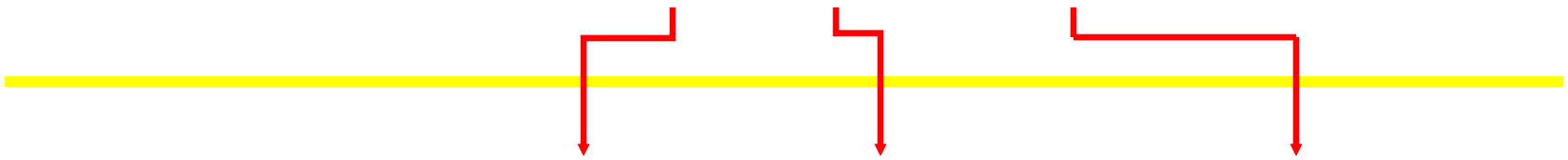
Parameters

- A method accepts 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**.
- The first actual parameter corresponds to the first formal parameter, the second actual parameter to second formal parameter, and so on..
- The type of the actual parameter must be assignment compatible with the corresponding formal parameter.

Parameters (cont.)

- Each time a method is called, the *actual arguments* in the invocation are *copied* into the formal arguments

```
ch = obj.calc (25, count, "Hello");
```



```
char calc (int num1, int num2, String message)
{
   int sum = num1 + num2;
   char result = message.charAt (sum);

   return result;
}
```


Call-by-Value

- When an actual parameter is passed into a method, its value is saved in the corresponding formal parameter.

- When the type of the formal parameter is a primitive data type, the value of the actual parameter is passed into the method and saved in the corresponding formal parameter (**CALL-BY-VALUE**).

Call-by-Reference

- When the type of the formal parameter is an object data type, the reference to an object is passed into the method and this reference is saved in the corresponding formal parameter (**CALL-BY-REFERENCE**).
- In call-by-value, there is no way to change the value of the corresponding actual parameter in the method.
- But in call-by-reference, we may change the value of the corresponding actual parameter by changing the content of the passed object.

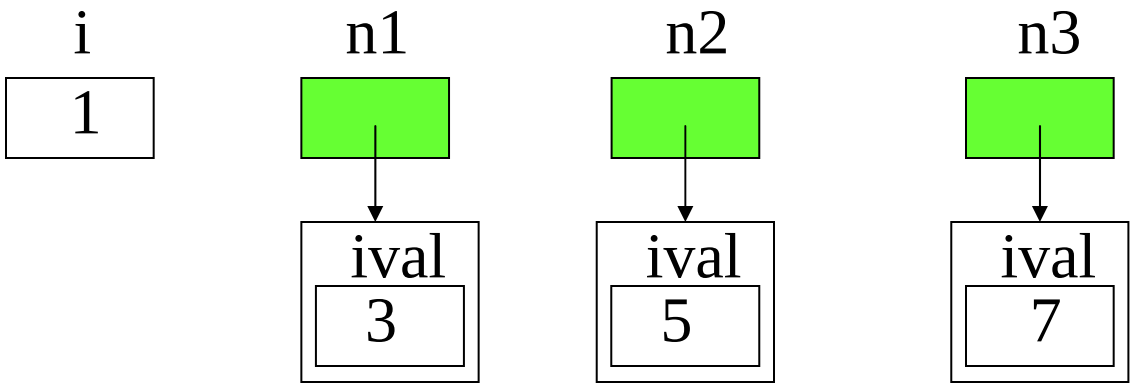
Call-by-Value and Call-by-Reference - Example

```
public class Test {
  public static void main(String[] args) throws IOException {
    int i=1;   MyInt n1,n2,n3;
    n1=new MyInt(3);  n2=new MyInt(5);  n3=new MyInt(7);
        ➔ values before chvalues
    chvalues(i,n1.ival,n2,n3);
        ➔ values after chvalues
        System.out.println(i+"-"+n1.ival+"-"+n2.ival+"-"+n3.ival);
  }
  static void chvalues(int x, int y, MyInt w, MyInt z) {
    x=x-1;  y=y+1;
    w = new MyInt(8);
    z.ival = 9;
  }
}
class MyInt {
  public int ival;
  public MyInt(int x) { ival=x; }
}
```

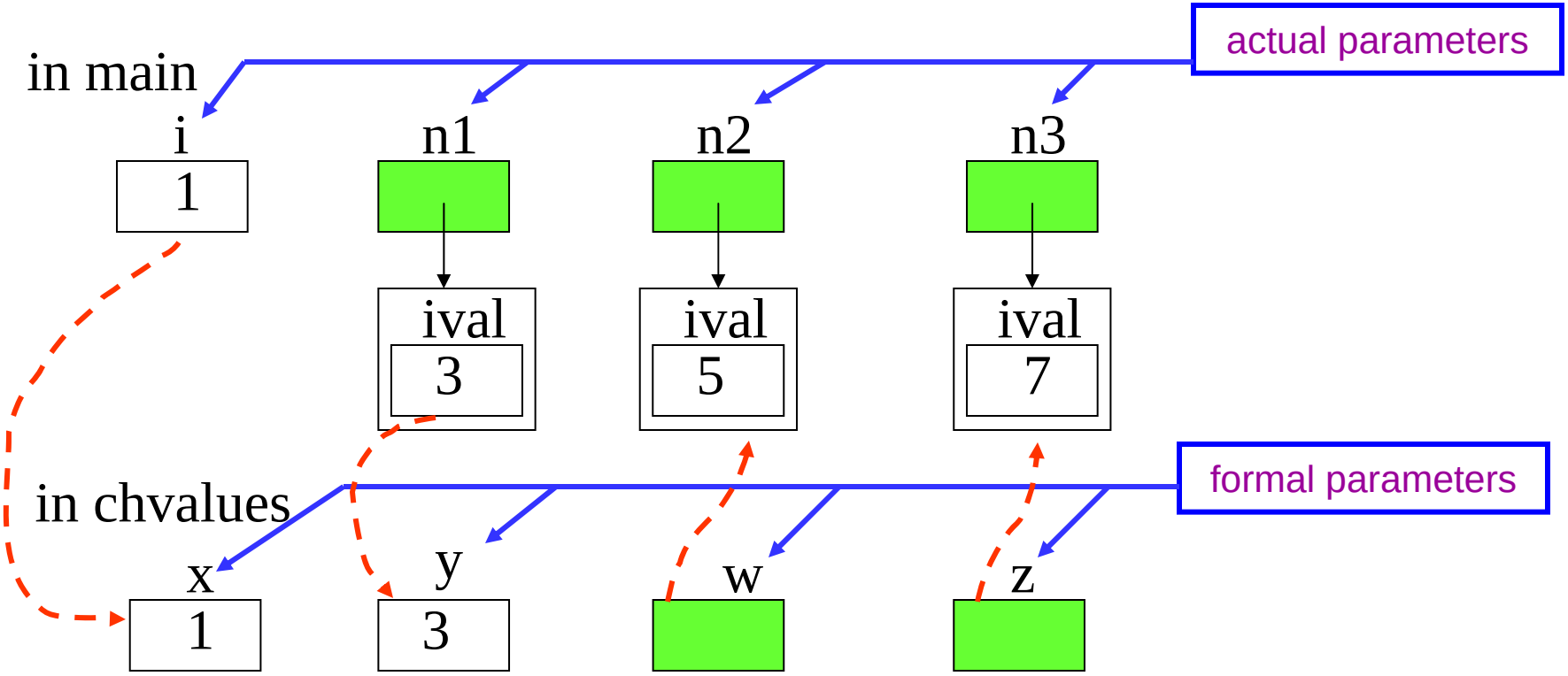

Call-by-Value and Call-by-Reference – Example

Step 1 - Execution begins in Main

in main before invocation of chvalues



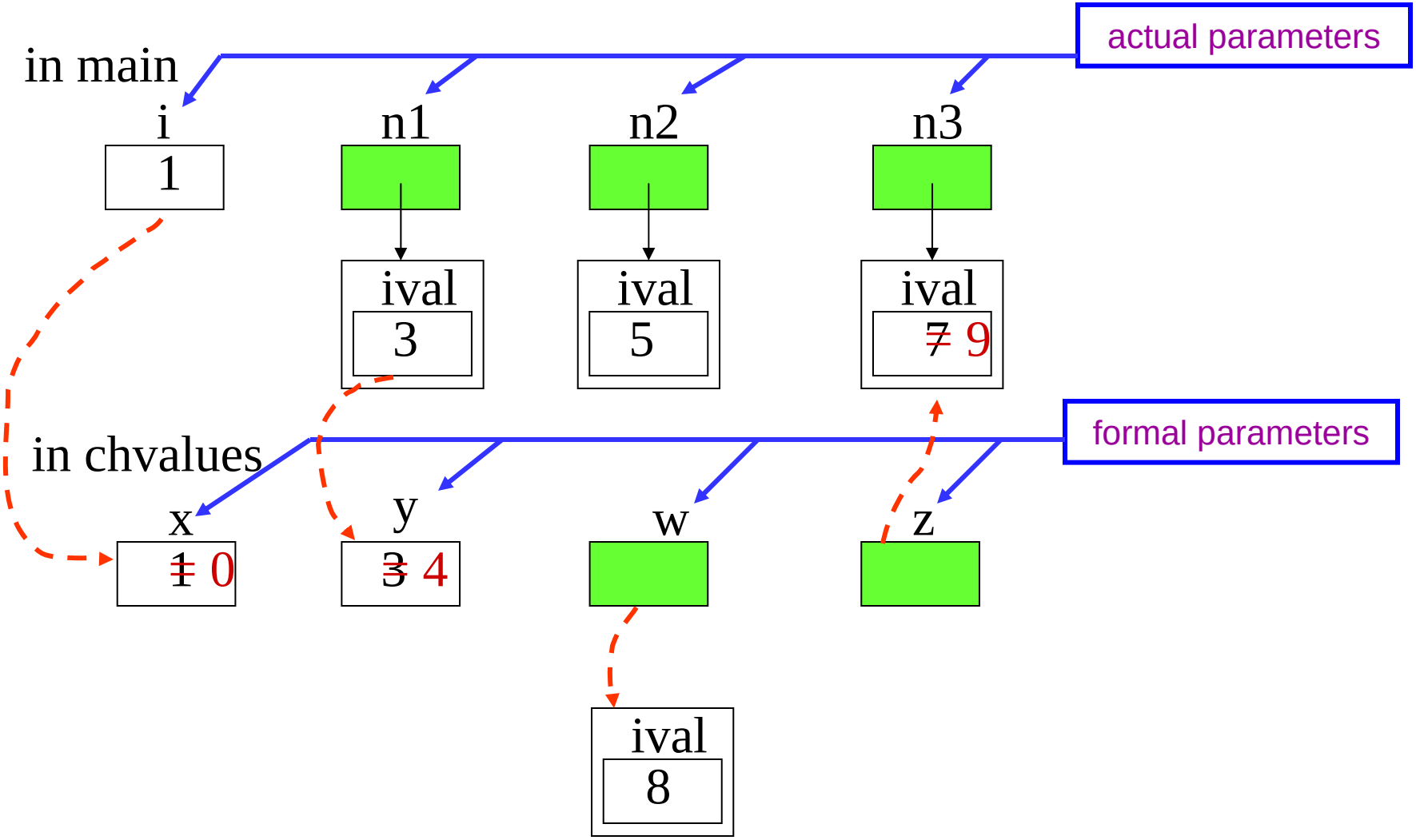
Step 2 – Invoke Method chvalues(i,n1.ival,n2,n3)



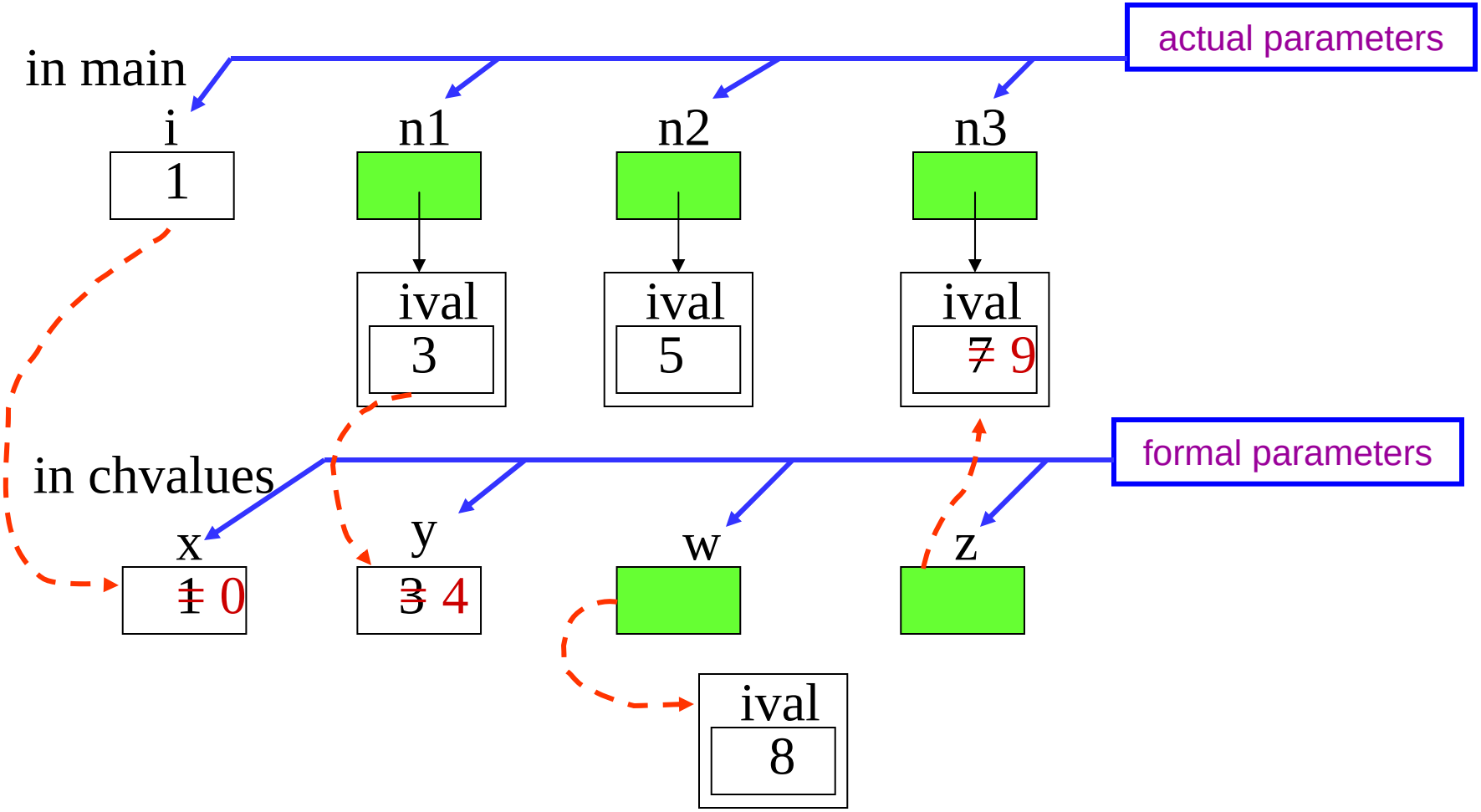
Red dotted lines indicate the passing of parameter values.

Notice that formal parameters x and y are primitive types and thus copies of the actual parameters are passed (pass by value) while w and z are objects and thus references to the objects are passed (pass by reference).

Step 3 – Method chvalues Executes



Step 4 – Return to Main Method – Print Results



Output: 1 – 3 – 5 – 9

Object Reference `this`

- The keyword `this` can be used inside instance methods to refer to the *receiving* object of the method.
- The receiving object is the object through which the method is invoked.
- The object reference `this` cannot occur inside static methods.
- Two common usage of this:
  - to pass the receiving object as a parameter
  - to access fields shadowed by local variables.

- Each instance method runs under an object, and this object can be accessible using `this` keyword.

Passing `this` as a Parameter

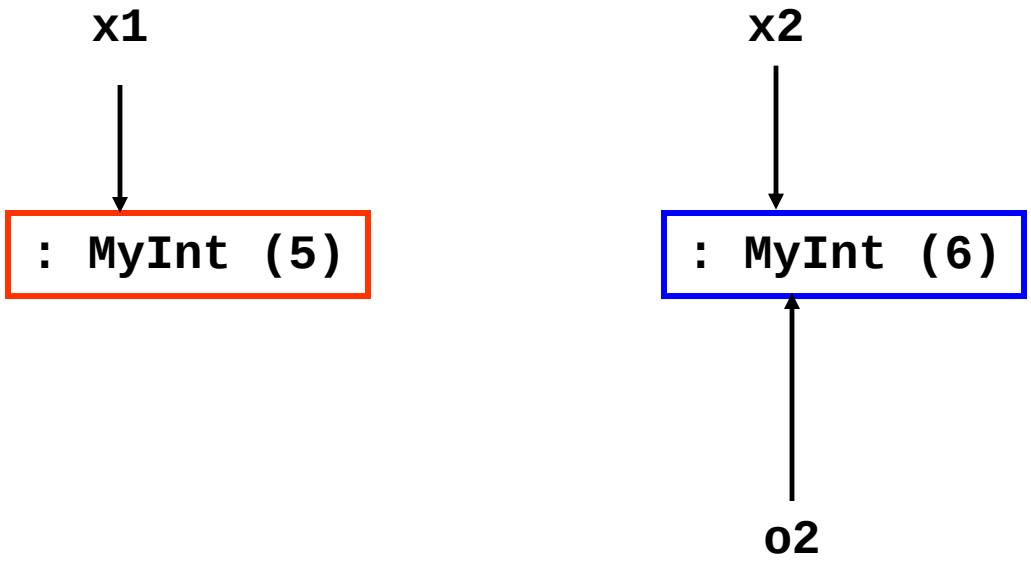
```
public class MyInt {
   private int ival;
   public MyInt(int val) { ival=val; }
   public boolean isGreaterThan(MyInt o2) {
       return (ival > o2.ival);
   }
   public boolean isLessThan(MyInt o2) {
       return (o2.isGreaterThan(this));
   }
}
```

Usage in some other place (method)

```
   MyInt x1=new MyInt(5), x2=new MyInt(6);
   x1.isGreaterThan(x2);  //output false
   x1.isLessThan(x2);     //output true
```


Passing `this` as a Parameter (cont.)

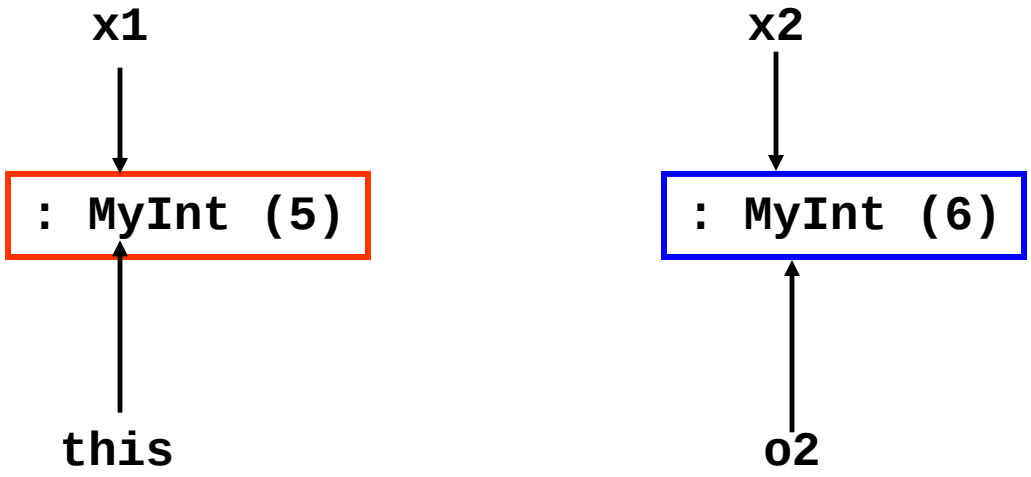
Variable bindings just after entering isGreaterThan method of x1



x1.ival is compared to x2.ival, since 5 is not greater than 6, false is returned

Passing `this` as a Parameter (cont.)

Variable bindings just after entering isLessThan method of x1



x1.isLessThan(x2) sets o2 = x2 and then calls o2.isGreaterThan(this)

`this` refers to the object on which the method invocation occurred which is x1. So the invocation is: o2.isGreaterThan(x1). On this invocation the formal parameter o2 becomes x1and what is returned is the result of the comparison of x2.ival > x1.ival (6 > 5) which is true.