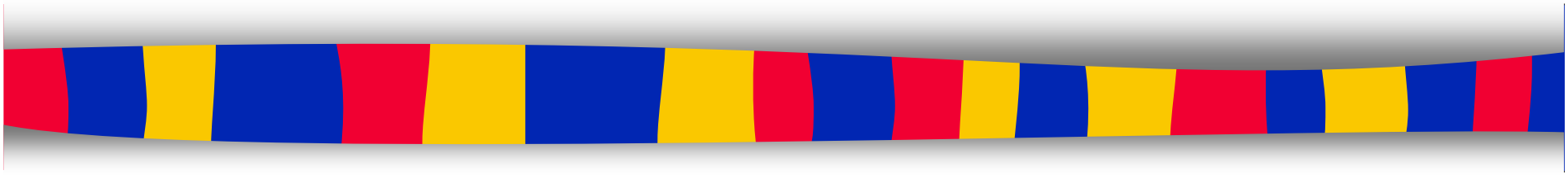
COMP-248
Object Oriented Programming I



Classes and Objects

By Emad Shihab, PhD, Fall 2015,
Parts of the slides are taken from Prof. L. Kosseim
Adapted for Section EE by S. Ghaderpanah, Fall 2015

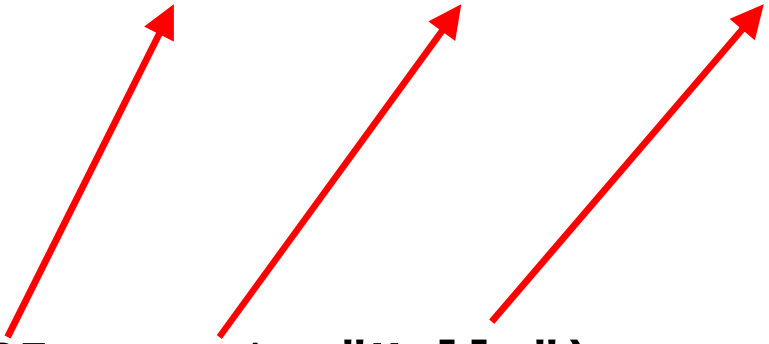
Next:

1. Writing our own classes
   - 1.1 Classes and Objects
   - 1.2 Instance Variables
   - 1.3 **Methods (More)**
2. Some notions of OOP
3. Passing and returning objects
4. Recap

Call by value

- When a method is called:
  - the actual parameters (arguments) are **copied** into the formal parameters
  - the method works with a **copy** of the actual parameters
  - when we return from the function, the actual parameters are **unchanged**

```
public void someMethod(int num1, int num2, String message)
{
    …
}
```



```
someMethod(25, count, "Hello");
```

Example

```
public class AClass
{
 private double anAttribute;

 public void aMethod(int aParam)
 {
   System.out.println(aParam);
   if (aParam < 0)
      aParam = 0;

   System.out.println(aParam);
 }
}
```

class def.

```
public class SomeDriver
{
  public static void main(String[] args)
  {
    AClass anObject=new AClass();
    int aVar = -100;

    System.out.println(aVar);
    anObject.aMethod(aVar);
    System.out.println(aVar);
  }
}
```

driver

```
-100
-100
0
-100
```

output

Formal and Actual Parameters

- When a method is called:
  - the order of the actual param. must be == to the order of the formal param.
  - the nb of actual param. must be == to the nb of formal param.
  - the types of the actual param. must be compatible with the types of the formal param.

```
public class Test
{
   …
   public void aMethod(String a, long b, double c, char d, boolean e)
   {…}
…
```

class def.

```
public class Driver {
    Test myTest = new Test();
    myTest.aMethod("hello", 10, 15.5, 'a', 'a' == 'b');  // 1. OK?
    myTest.aMethod("hello", 10.5, 15, 'a', true);  // 2. OK?
    myTest.aMethod("hello", 10, 15);  // 3. OK?
    myTest.aMethod("hello", 10, 15, 67, true);  // 4. OK?
    myTest.aMethod("hello", 10, 15, (char)67, true);  // 5. OK?
…
```

Yes
No
No
No
Yes

driver

Type Conversion of Parameters

- The types of the actual param. must be compatible with the types of the corresponding formal param.

```
public double myMethod(int p1, int p2, double p3) {…}
…
int a=1,b=2,c=3;
double result = myMethod(a,b,c);
```

- If no exact type match --> automatic type conversion

  c is type casted to a **double**

  remember:

  byte→short→int→long→float→double

The this Reference

All instance variables are understood to have **<the calling object>.** in front of them

Sometime it is handy, and **even necessary**, to have an explicit name for the calling object

Inside a method you can use the keyword **this** as a name of the calling object

```
public void deposit(int amount)
{
   balance += amount;
  this.balance += amount;
}
```

The this Reference

this must be used:

if a **parameter or other local variable has the same name** as an **instance variable**

```
public class Account {
  private double balance;

  public void initialize(double balance)
  {
     this.balance = balance;
  }       ↑               ↑
}      instance          local
```

Example

```
public class Account {

    private int account;

    public void setAcc(int x)
    {
        account = x;
    }

    public int aMethod(int account)
    {
        this.account += account;
        return account;
    }
}
```

```
public class AccountTest {

public static void main (String[] args)
    {
        int x = 10;
        Account myTest=new Account();

        myTest.setAcc(50);
        int y = myTest.aMethod(x);
        System.out.println(y);

    }
}
```

```
Output:
10
```

Example 2

```
public class Account {

    private int account;

    public void setAcc(int x)
    {
        account = x;
    }

    public int aMethod(int account)
    {
        account += account;
        return account;
    }
}
```

```
public class AccountTest {

public static void main (String[] args)
    {
        int x = 10;
        Account myTest=new Account();

        myTest.setAcc(50);
        int y = myTest.aMethod(x);
        System.out.println(y);

    }
}
```

```
Output:
20
```

Example 3

```
public class Account {

    private int account;

    public void setAcc(int x)
    {
        account = x;
    }

    public int aMethod(int account)
    {
        this.account += account;
        return this.account;
    }
}
```

```
public class AccountTest {

public static void main (String[] args)
    {
        int x = 10;
        Account myTest=new Account();

        myTest.setAcc(50);
        int y = myTest.aMethod(x);
        System.out.println(y);

    }
}
```

```
Output:
60
```

A special method: toString()

Printing and object

```
public class Banking
{
   public static void main (String[] args)
   {
      Account acct1 = new Account("Ted", 123, 100.0);
      acct1.deposit (25.55);
      System.out.println(acct1);
   }
}
```

driver

the **toString** method is automatically called

if you have not defined a **toString** method, then

```
Account@182f0db
```

otherwise, define your own version of **toString** for your object

Defining your own toString()

`toString` must:

- takes no parameter
- return a String value that represents the data in the object

```
public class Account
{
   private long acctNumber;
   private double balance;
   private String name;
   public Account(String owner, long account, double initial)
   { … }
   //  Returns a one-line description of the account as a string.
   public String toString(              ) {
      return (acctNumber + "\t" + name + "\t" + balance);
   }
…
```

class def.

```
123  Ted  125.55
```

Another special method: equals()

To compare two objects, you can use the equals() method

`public boolean equals (Class_Name P_Name)`

When defining the equals(), a common way to define equals() is to say equals() returns true **if all instance variables of one object equals the instance variables of another object**.

```
public class Account
{
   private long acctNumber;
   private double balance;
   private String name;
   public Account(String owner, long account, double initial)
   { … }
   //  Returns a one-line description of the account as a string.
   public String toString() {
      return (acctNumber + "\t" + name + "\t" + balance);
   }

    public boolean equals(Account obj){
          return((this.acctNumber == obj.acctNumber)
                    && ((this.name).equals(obj.name)));
    }
```

Notes on equals() and toString()

- Java expects certain methods to be in (almost) all classes
- Some standard Java libraries assumes the existence of these methods
- equals() and toString() are two such methods
- You should include/define these methods in your classes and make sure to keep the exact same spelling

Constructors

Assume:

```
public class BankAccount
{
   private long acctNumber;
   private double balance;
   private String name;

   public void initialize(String theOwner, long theAccount, double theInitial)
   {
      name = theOwner;
      acctNumber = theAccount;
      balance = theInitial;
   }
…
```

BankAccount.java

to initialize an object, we can do:

```
BankAccount account123 = new BankAccount();
account123.initialize("ted", 123, 100);

BankAccount account555 = new BankAccount();
account555.initialize("mary", 555, 300);
```

driver

Constructors

A constructor is a special method that:

- is called automatically called when an object of the class is declared
- is usually used to initialize the data of an object
- **must have the same name as the class name**
- has no return type (not even **`void`**)
- must be **`public`**
- can have parameters

```
BankAccount account123 = new BankAccount("ted", 123, 100);
BankAccount account555 = new BankAccount("mary", 555, 300);
```

If you do not define any constructor in your class, a default constructor that initializes instance variables to 0

Example 1

```
public class BankAccount
{
   private long acctNumber;
   private double balance;
   private String name;

   // a constructor:
   public BankAccount (String theOwner, long theAccount, double theInitial)
   {
      name = theOwner;
      acctNumber = theAccount;
      balance = theInitial;
   }

   public void deposit(double amount)
   { … }
   public void withdraw(double amount)
   { … }
   …
}
```


BankAccount.java

Example 2

```
public class Coin
{
   private final int HEADS = 0;
   private final int TAILS = 1;
   private int face;

   public Coin()
   {
      flip();
   }

   public void flip()
   {
      face = (int)(Math.random()*2);
   }

   public boolean isHeads()
   {
      return (face == HEADS);
   }
```

```
public class CountFlips
{
   public static void main (String[]
   args)
   {
      Coin myCoin = new Coin();
      Coin anotherCoin = new Coin();
      Coin athirdCoin = new Coin();

      …
   }
}
```

driver

Default Constructor

If you do not include any constructors in your class,

- Java will automatically create a *default* or *no-argument* constructor

The default constructor:

- takes no arguments
- initializes all instance variables to zero (null or false)
- but allows the object to be created

If you include even one constructor in your class

- Java will not provide this default constructor

Overloading methods

Overloading = **same name is used to refer to different things**

"chair" (person or furniture)

/ (integer or real division)

Overloaded methods:

several methods that have the **same name** but **different definitions**

the ***signature*** of each overloaded method must be unique

signature = name of method + parameter list

the compiler determines which version of the method is being invoked by analyzing the signature

the return type of the method is not part of the signature

Example

Version 1

```
public int increment(int x)
{
   return x+1;
}
```

Version 2

```
public int increment(int x, int value)
{
   return x+value;
}
```

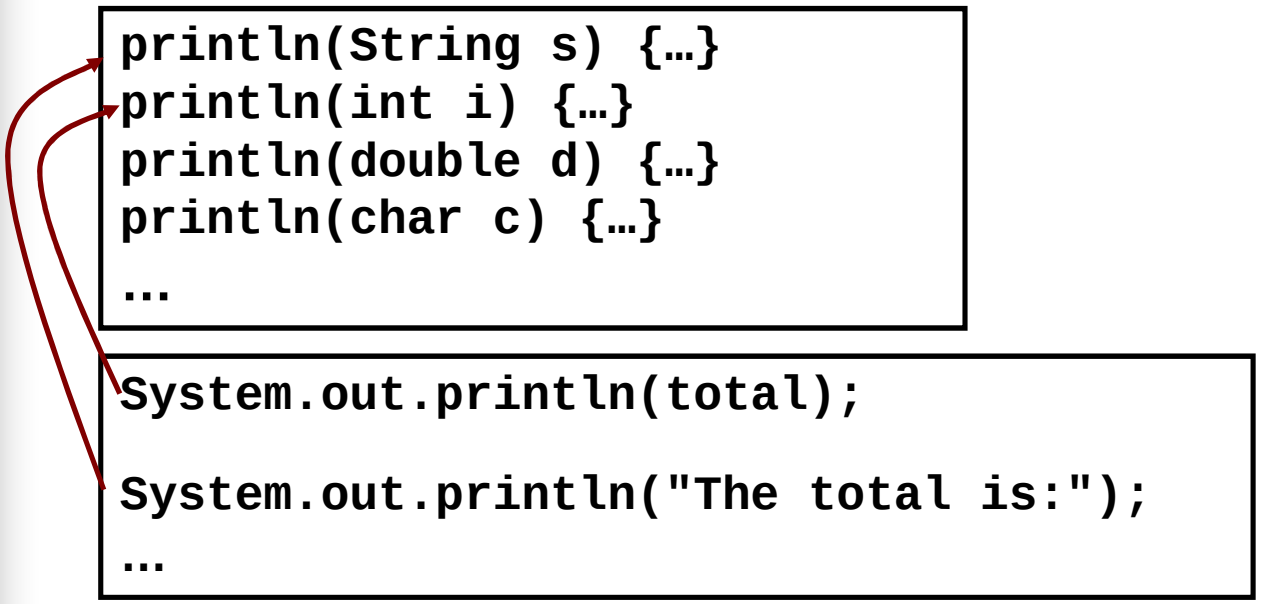
Invocations

```
int result = 1;

result = increment(20, 4);      // 1. OK?
result = increment(10);         // 2. OK?
result = increment(result);     // 3. OK?
result = increment(result, result);  // 4. OK?
result = increment(result, result, result); // 5. OK?
result = increment();           // 6. OK?
```

Overloaded methods

guess what… the **println** method is overloaded!



Overloaded methods

practical when the same operation must be done on different types or different numbers of parameters

```
public class Calculator
{
   public int addem(int op1, int op2) {
      return op1+op2;
   }
   public int addem(int op1, int op2, int op3) {
      return op1+op2+op3;
   }

   public int opposite(int op) {
      return -op;
   }
   public boolean opposite(boolean op) {
      return !op;
   }
   public char opposite(char op) {
      if (Character.isUpperCase(op))
         return Character.toLowerCase(op);
      return Character.toUpperCase(op);
   }
}
```

Calculator.java

```
Calculator myCalc = new Calculator();

System.out.print(myCalc.addem(10,20));
System.out.print(myCalc.addem(10,2,5));

System.out.print(myCalc.opposite(10));
System.out.print(myCalc.opposite('a'=='b'));
System.out.print(myCalc.opposite('a'));
```

driver

Overloading constructors

Constructors are methods… so they can be overloaded

```
public class Account
{
  private long acctNumber;
  private double balance;
  private String name;

  public Account(String owner, long nb, double init)  {
     name = owner;
     acctNumber = nb;
     balance = init;
  }
  public Account(String owner, long nb)  {
     name = owner;
     acctNumber = nb;
     balance = 0;
  }
  public Account(long nb)  {
     name = "";
     acctNumber = nb;
     balance = 0;
  }
…
```

Account.java

```
public class Banking
{
  public static void main (String[] args)
  {
    Account acct1 = new Account("Jane", 456, 400); // 1. OK?
    Account acct2 = new Account("Ted", 123); // 2. OK?
    Account acct3 = new Account("Ted"); // 3. OK?
    Account acct4 = new Account();  // 4. OK?
  }
}
```

Banking.java

Car example…

Next:

1. Writing our own classes
   1.1 Objects vs classes
   1.2 Instance Variables
   1.3 Methods
2. **Some notions of OOP**
3. Passing and returning objects
4. Recap