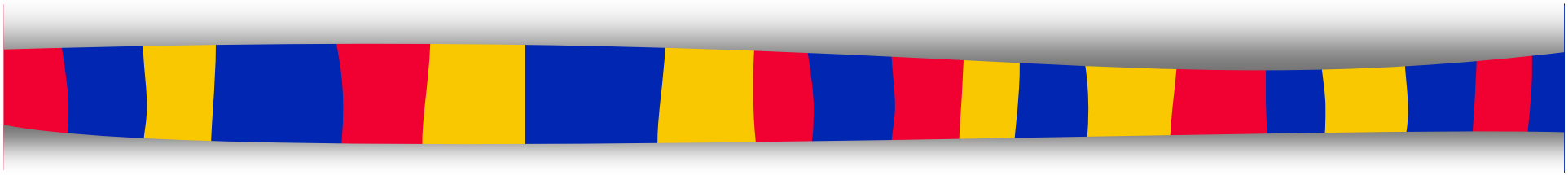
COMP-248
Object Oriented Programming I



Final Review

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

Problem Solving

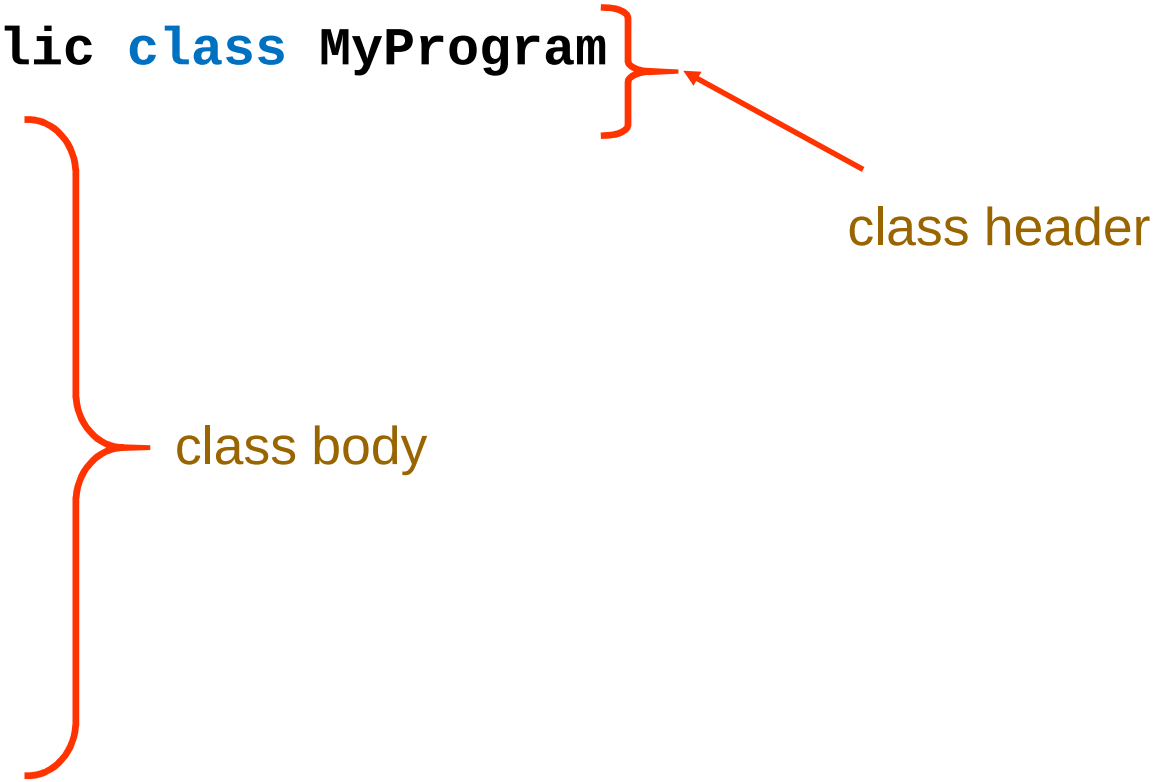
- The purpose of writing a program is to solve a problem
- The general steps in problem solving are:
  - Understand the problem
  - Design a solution (find an algorithm)
  - Implement the solution (write the program)
  - Test the program and fix any problems

Object Oriented Programming

- OPP treats everything in the world as objects (e.g., automobiles, universities, people)
- Each object has the ability to perform actions, and those actions can have an impact on other objects
- **OOP is a methodology that views a program as consisting of objects that interact with each other by means of actions**

Java Program Structure

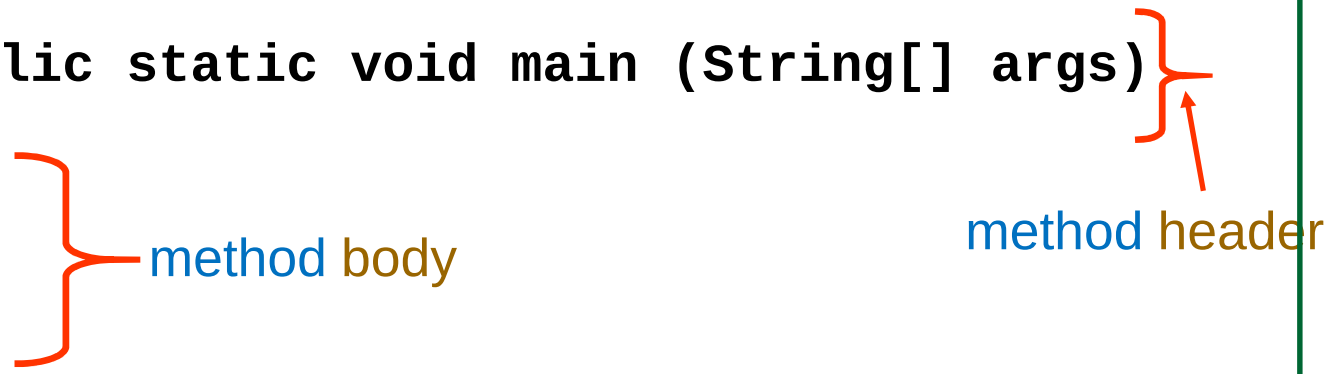
```
//  comments about the class
public class MyProgram
{


}
```



MyProgram.java

Java Program Structure

```
//  comments about the class
public class MyProgram
{

   //  comments about the method
   public static void main (String[] args)
   {

   }

}
```



MyProgram.java

Syntax and Semantics

- **Syntax rules**
  - define how we can put together symbols, reserved words, and identifiers to make a valid program
- **Semantics**
  - define what a statement means
- A program that is syntactically correct is not necessarily logically (semantically) correct

Three types of errors

- **Compile-time (syntax) errors**
  - The compiler will find syntax errors and other basic problems
  - An executable version of the program is not created
- **Run-time errors**
  - A problem can occur during program execution
  - Causes the program to terminate abnormally

Three types of errors …

- **Logical (semantic) errors**
  - A mistake in the algorithm
  - Compiler cannot catch them
  - A program may run, but produce incorrect results

Identifiers

- are the words a programmer uses in a program to **name variables, classes, methods**, …

- Rules to create an identifier:
  - can be made up of: letters, digits, the underscore character (_), and the dollar sign ($)
  - no limit on length
  - **cannot** begin with a digit
  - **cannot** be a *reserved word*

Primitive Types

- 8 primitive data types in Java

Numeric

4 types to represent integers (ex. 3, -5):

```
byte, short, int, long
```

2 types to represent floating point numbers (ex. 3.5):

```
float, double
```

Characters (ex. **'a'**)

```
char
```

Boolean values (true/false)

```
boolean
```

Output & Input

`System.out.print`
Displays what is in parenthesis

`System.out.println`
Displays what is in parenthesis
Advances to the next line

Examples:

```
System.out.print("hello");
System.out.print("you");

System.out.println("hello");
System.out.println("you");

System.out.println();

int price = 50;
System.out.print(price);

char initial = 'L';
System.out.println(initial);
```

```
helloyouhello
you

50L
```

Output

Console Input (p. 76)

Since Java 5.0, use the **`Scanner`** class

The keyboard is represented by the **`System.in`** object

```
import java.util.Scanner;

public class MyProgram
{
   public static void main (String[] args)
   {
      Scanner myKeyboard = new Scanner(System.in);
      …
      String name = myKeyboard.next();
      int age = myKeyboard.nextInt();
      …
```

1. Create an object of class Scanner
2. Reads one **word** from the keyboard
3. Reads an integer from the keyboard

To read from a Scanner

To read *tokens*, use a *nextSomething()* method

- **`nextBoolean()`,**
- **`nextByte`,**
- **`nextInt()`,**
- **`nextFloat()`,**
- **`nextDouble()`,**
- **`next()`,**
- **`nextLine()`**
- **…**

tokens are delimited by whitespaces
(ie blank spaces, tabs, and line breaks)

Note: no **`nextChar()`**

```
import java.util.Scanner;
…
Scanner myKeyboard = new Scanner(System.in);
System.out.println("Your name:");
String name = myKeyboard.next();
System.out.println("Welcome " + name + " Enter your age:");
int age = myKeyboard.nextInt();
```

String

- So far we have seen only primitive types

- A variable can be either:
  - a primitive type (ex: **`int`**, **`float`**, **`boolean`**, …)
  - or a reference to an object (ex: String, Array, …)

- A character string:
  - is an object defined by the **`String`** class
  - delimited by double quotation marks ex: "hello", "a"

```
System.out.print("hello"); // string of characters

System.out.print('a');     // single character
```

Declaring Strings

- Because strings are so common, we don't have to use the **new** operator to create a **String** object

```
String title;
title = new String("content of the string");
```

```
String title = new String("content of the string");
```

```
String title;
title = "content of the string";
```

```
String title = "content of the string";
```

- These special syntax works only for strings

Arithmetic Expressions

- An expression is a combination of one or more operands and their operators
- Arithmetic operators:

Addition	+
Subtraction	-
Multiplication	*
Division	/
Remainder	%

Operator Precedence

- Operators can be combined into complex expressions

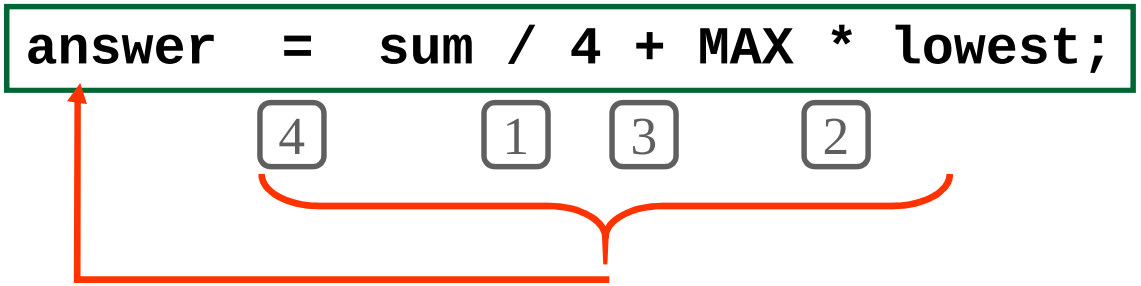
```
result  =  total + count / max - offset;
```

- precedence determines the order of evaluation
  - 1st: expressions in parenthesis
  - 2nd: unary + and -
  - 3rd: multiplication, division, and remainder
  - 4th: addition, subtraction, and string concatenation
  - 5th: assignment operator

Assignment Revisited

- The assignment operator has a lower precedence than the arithmetic operators

First the expression on the RHS is evaluated



Then the result is stored in the variable on the LHS

Increment and Decrement

- In Java, we often add-one or subtract-one to a variable…
- 2 shortcut operators:
  - The *increment operator* (++) adds one to its operand
  - The *decrement operator* (--) subtracts one from its operand
- The statement: **`count++;`**
  is functionally equivalent to: **`count = count+1;`**
- The statement: **`count--;`**
  is functionally equivalent to: **`count = count-1;`**

Increment and Decrement

- The increment and decrement operators can be used in expressions in two forms:

1. **in prefix form: `++count;`**
   1. the variable is incremented/decremented by 1
   2. the value of the entire expression is the **new** value of the variable (**after** the incrementation/decrementation)
2. **in postfix form: `count++;`**
   1. the variable is incremented/decremented by 1
   2. the value of the entire expression is the **old** value of the variable (**before** the incrementation/decrementation)

Arithmetic promotion

- happens automatically, if the operands of an expression are of different types

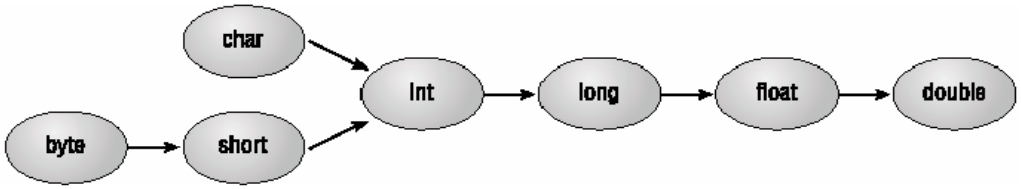
```
aLong + anInt * aDouble
```

- operands are promoted so that they have the same type

- promotion rules:
  - if 1 operand is of type… the others are promoted to…

if 1 operand is of type…	the others are promoted to…
`double`	`double`
`float`	`float`
`long`	`long`

  - **`short`**, **`byte`** and **`char`** are always converted to **`int`**



Flow of Control

1. **Sequence**:

   Unless specified otherwise, the order of statement execution is linear/sequential

   one statement after the other, in sequence

2. **Conditional statements**:

   a statement may or may not be executed depending on some condition

3. **Repetition statements (loops):**

   a statement is executed over and over, repetitively, until some condition becomes true or false

These decisions are based on a **boolean expression** (also called a *condition*) that evaluates to true or false

The order of statement execution is called the **flow of control**

Conditional statements

let us choose which statement will be executed next

sometimes called ***selection statements***

Java has 3 conditional statements:

- the ***if*** statement
- the ***if-else*** statement
- the ***switch*** statement

1- The if statement (p.96)

syntax:

`if` is a Java reserved word

The *condition* is a boolean expression. (evaluates to either true or false)

```
if ( condition )
    statement;
```

If the *condition* is true, the *statement* is executed.
If it is false, the *statement* is skipped.

2- The if-else statement

An *else clause* can be added to an `if` statement to make an *if-else statement*

```
if ( condition )
   statement1;
else
   statement2;
```

A note on comparing characters

We can use the relational operators to compare 2 characters

The results are based on the Unicode character set

```
if ('+' < 'J')
   System.out.println("+ is less than J in Unicode");
```

```
char userAnswer = 'y';
if (userAnswer == 'Y')
   System.out.println("the user said yes");
```

A note on comparing strings 

We **cannot** use the relational operators to compare strings (<, ==, …)

use the **`equals()`** method

to determine if two strings have the same content

ex: **`firstString.equals(secondString)`**

returns a **`boolean`**:

`true` if **`firstString`** has the same content as **`secondString`**

`false` otherwise

Logical operators

To combine multiple boolean expressions into a more complex one

`!`	Logical NOT	(unary operator)
`&&`	Logical AND	(binary operator))
`||`	Logical OR	(binary operator)

They all take boolean operands and produce boolean results

`!a` is **`false`** if `a` is **`true`**
`!a` is **`true`** if `a` is **`false`**

a	!a
true	
false	

Compound statements

```
if ( condition )
   statement;
```

```
if ( condition )
   statement1;
else
   statement2;
```

what if you wanted to execute several statements?

Several statements can be grouped together into a ***compound statement*** (or block):

```
{
   statement1;
   statement2;
   …
}
```

A **block** can be used wherever a statement is called for by the Java syntax

The switch statement

syntax:

switch, **case**, **break**, **default** are reserved words

break and **default** case are optional

```
switch ( expression )
{
    case value1 :
       statement-list1
       break;
    case value2 :
    statement-list2
       break;
    case value3 :
    statement-list3
       break;
    case  ...
    default:
      default-statement-list
}
```

If *expression* matches *value2*, control jumps to here

The conditional operator

shortcut to an `if` in some cases

ternary operator (needs 3 operands)

syntax : ***`condition`*** `?` ***`expression1`*** `:` ***`expression2`***

semantics:

- if the ***`condition`*** is true, ***`expression1`*** is evaluated;
- if it is false, ***`expression2`*** is evaluated
- the result of the chosen expression is the result of the entire conditional operator

Repetition statements (loops)

allow us to execute a statement several times

like conditional statements, they are controlled by boolean expressions

Java has 3 kinds of loops:

- the **`while`** loop
- the **`do-while`** loop
- the **`for`** loop

The while loop

syntax:

while is a reserved word

```
while ( condition )
    statement;
```

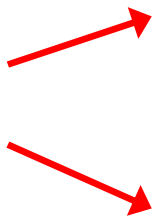
If the *condition* is true, the *statement* is executed.
Then the *condition* is evaluated again.

The *statement* is executed repeatedly until the *condition* becomes false.

The do-while loop

syntax:

do and while are reserved words



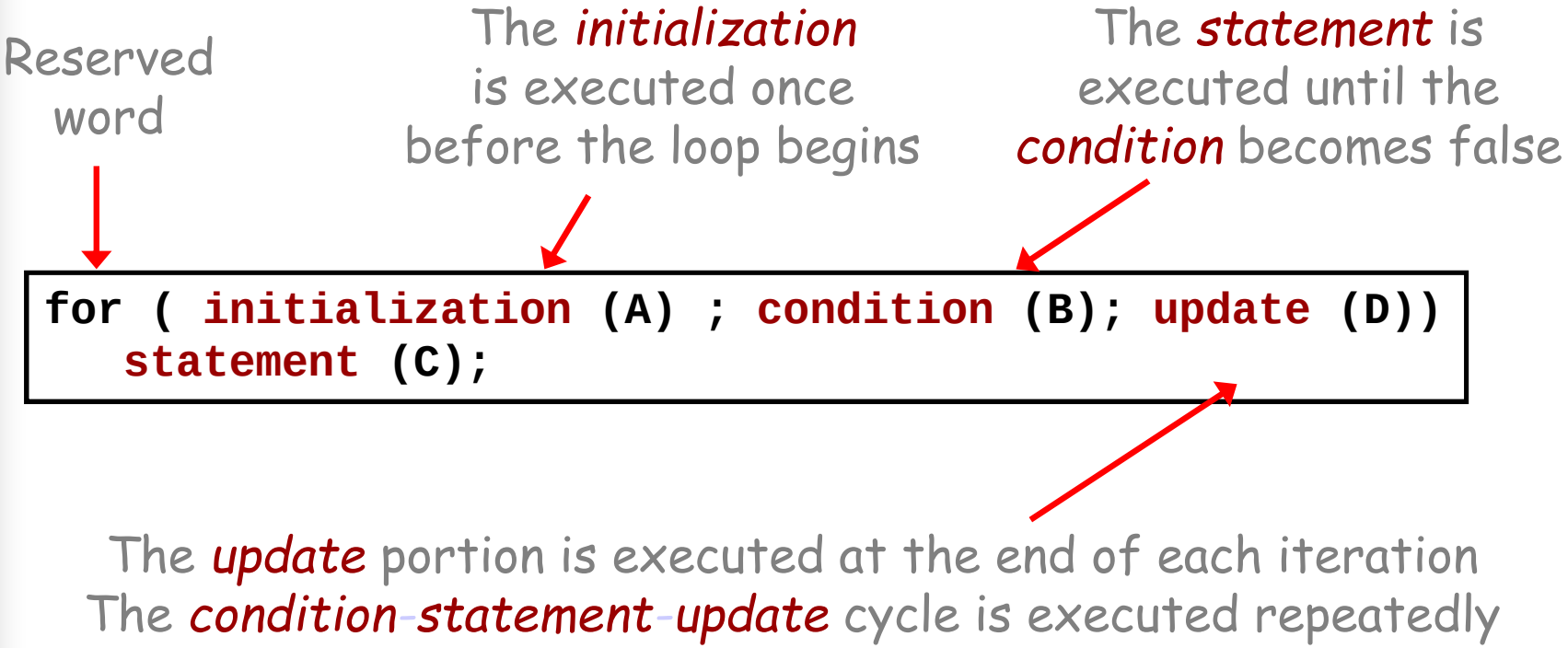
```
do
{
    statement;
}
while ( condition );
```

The *statement* is executed once initially, and then the *condition* is evaluated

The *statement* is executed repeatedly until the *condition* becomes false

The for loop

syntax:



Nested loops

a **`for`** inside a **`for`**, a **`while`** inside a **`for`**, a **`do-while`** inside a **`while`**, …

i.e. the body of a loop can contain another loop

consists of:

- an outer loop
- an inner loop

for 1 iteration of the outer loop, the inner loop goes through its full set of iterations

break and continue

bypasses the normal flow of control of loops

very practical sometimes… but use in moderation…

`break`

will exit the inner-most loop without evaluating the condition

`continue`

will interrupt the current iteration (of the inner-most loop)

and will force a new evaluation of the condition for a possible new iteration

Note: in a for loop, the incrementation is done before the condition is tested…

The exit Statement

A **break** statement will end a loop or switch statement, but will not end the program

The **exit** statement will immediately end the program as soon as it is invoked:

```
System.exit(0);
```

The **exit** statement takes one integer argument

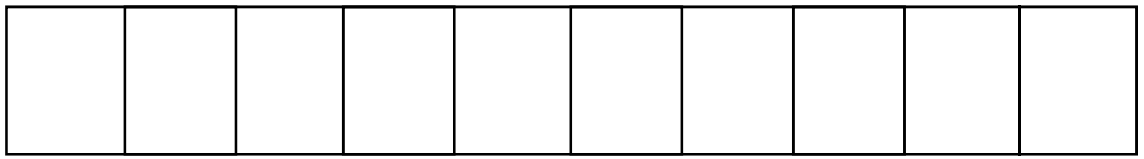
- By tradition, a zero argument is used to indicate a normal ending of the program

Arrays

An *array* is an ordered list of elements of the same type

The entire array has a single name

Each value has a numeric *index*



This array holds 10 values that are indexed from 0 to 9

An array of size n is indexed from 0 to n-1

Declaring and creating arrays

declaring the reference:

syntax: `type_of_elements[] name_of_array;`

```
int[] scores;        double[] marks;
char[] vowels;    String[] sentence;
```

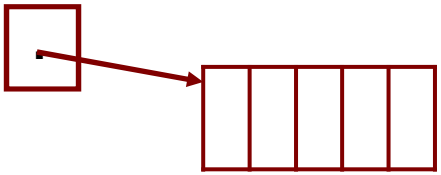
creating the elements:

syntax: `name_of_array = new type_of_elements[size];`

```
scores = new int[5];
```

declaration + creation:

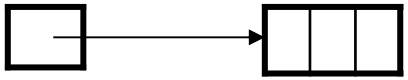
```
int[] scores = new int[5];
```



Multidimensional arrays

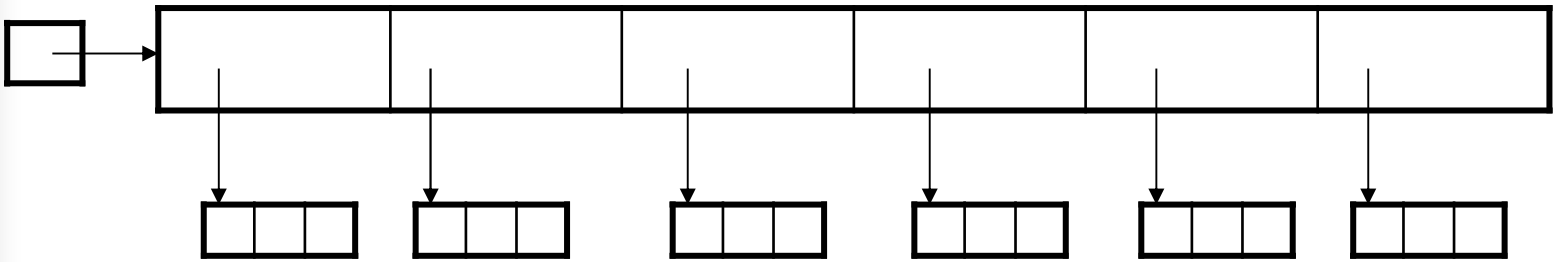
A one-dimensional array

stores a list of elements of simple type (primitive or reference)



A two-dimensional array

stores a list of elements, where each element is a 1-D array of simple type



Two-dimensional arrays

Declaration:

```
double[] student = new double[5]; // 1-D 5 tests for 1 student
double[][] section = new double[5][80]; // 2-D 80 students per section
double[][][] course = new double[3][5][80]; // 3-D 5 sections per course
```

Access to an element

```
value = section[3][5]
```

Length with multidimensional arrays

```
char[][] page = new char[30][100];
```

`length` does not give the total number of indexed variables

- `page.length` is equal to 30
- `page[0].length` is equal to 100

```
int row, col;
for (row = 0; row < page.length; row++)
  for (col = 0; col < page[row].length; col++)
    page[row][col] = 'Z';
```

Classes and Objects

- A class is a type and you can declare variables of a class type.

- A value of a class is called an objects. An object has 2 components:

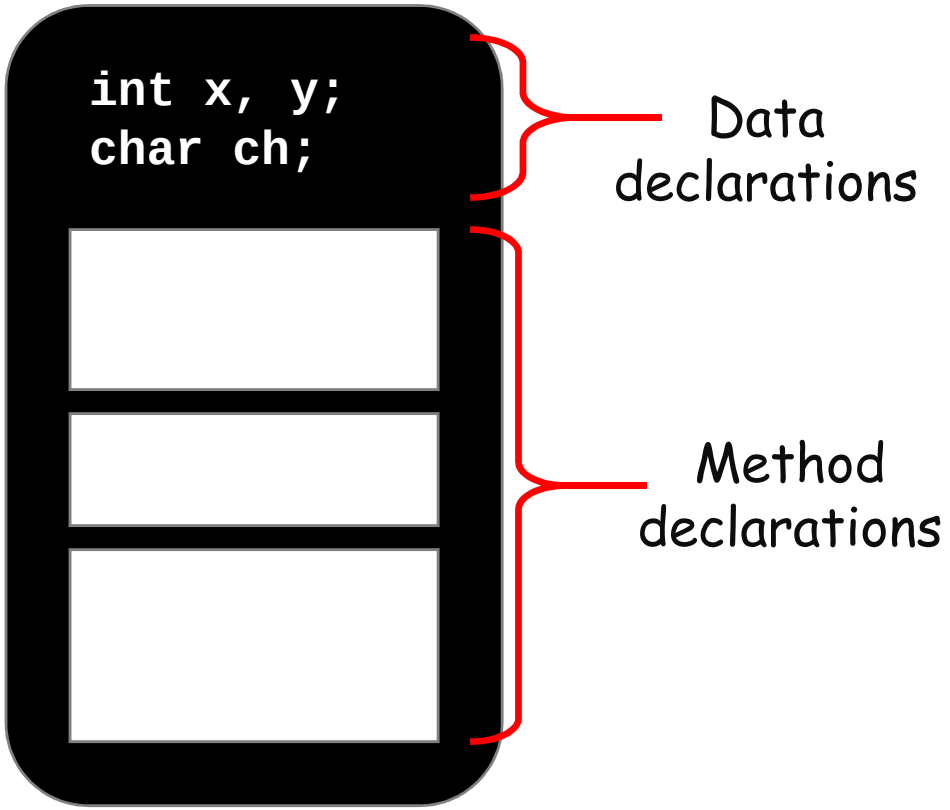
  - *Data (instance variables)* - descriptive characteristics

  - *Actions (methods)* - what it can do (or what can be done to it)

Declaration of Classes

A class contains:

1. data declarations (instance variables)
2. action declarations (methods)



Declaring Objects

- The new operator is used to create an object of a class and associate it with a variable

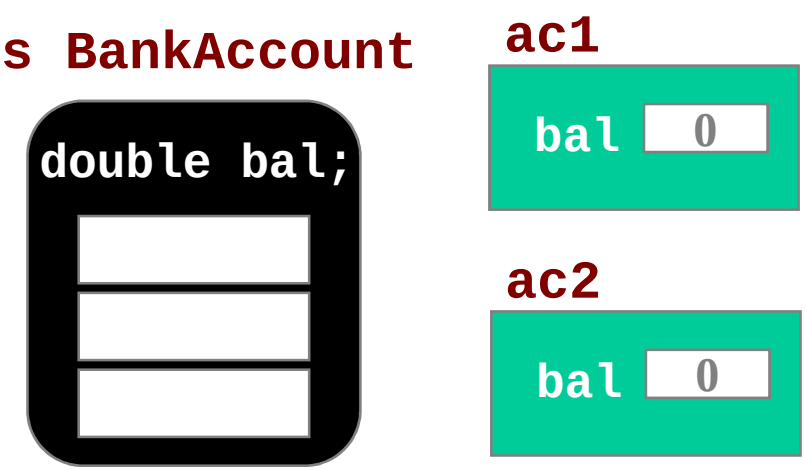
- Example:
  - `nameOfClass nameOfObject = new nameOfClass();`
  - `nameOfClass nameOfObject ;`
    `nameOfObject = new nameOfClass();`

Instance Variables

- variables/constants declared inside the class **but not inside a specific method**
- also called attributes
- can be used by any method of the class
- initialized to 0 (false for booleans)
- each object (instance) of the class has its own instance data

Example:

```
BankAccount ac1=new BankAccount();
BankAccount ac2=new BankAccount();
```



Methods

- Implement the behavior of all objects of a particular class
- All objects of a class share the method definitions
- Some methods are a bit special…
  (ex: constructor)
- Group of statements that are given a name
- A method is defined once, but can be used (called/invoked) several times

Methods cont'd…

There are two kinds of methods:

- Methods that compute and return a value
- Methods that perform an action
  does not return a value
  is called a **`void`** method

Methods that return a result

- Must specify the type of that value in its heading:
  `public typeReturned methodName(paramList)`

- Examples:

  description: determines if the coin is a tail (1==tail, 0==heads)
  name: **isTail**
  result: **boolean**

```
public boolean isTail()
{
  if (face == 1)
     return true;
  else
     return false;
}
```

  description: returns the value of the face
  name: **getFace**
  result: **int**

```
public int getFace()
{
   return (face);
}
```

Accessing Class Members

To access any members (data or method)

within the class

```
nameOfData
nameOfMethod(actualParameters)
```

from outside the class

non-static:

```
nameOfObject.nameOfData
nameOfObject.nameOfMethod(actualParameters)
```

Static:

```
We will cover this later…
```

Calling Methods

Methods that return a result are expressions that have:

a type and a value

Methods that do not return a result are **statements**

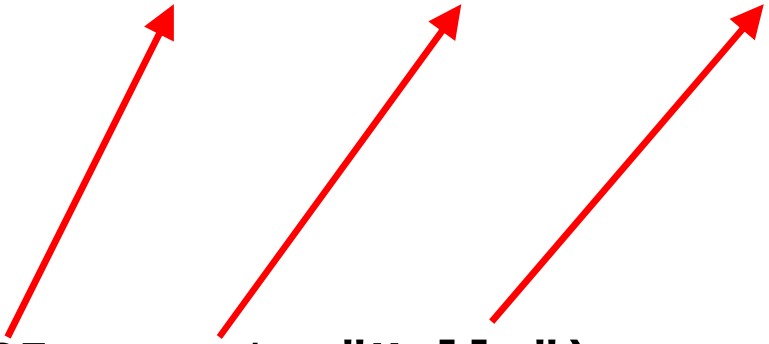
you call them with the statement: `objectName.methodName();`

```
public class Driver {
   public static void main (String[] args)   {
      Coin coin1 = new Coin();
      boolean result;

      coin1.flip();           // 1. OK?   Yes
      result = coin1.flip(); // 2. OK?   No
      if (coin1.flip())       // 3. OK?   No
         System.out.print("whatever");
      System.out.print(coin1.flip()); //4. OK?  No

   }
}
```

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");
```

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;
}
```

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)));
    }
```

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

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

Visibility modifiers

- **public** members
  - can be directly accessed from anywhere (inside & outside the object)
  - violate encapsulation
- **private** members
  - can only be accessed from inside the class definition
- by default…
  - members can be accessed by any class in the same package
  - i.e. access is more open than private, but more strict than public

Visibility modifiers

- **Data**
  - should be **private**
  - public data violate encapsulation
  - constants are OK (but not encouraged) to be public because they cannot be modified anyways
- Methods:
  - should mostly be **public**
    - if the method provides the object's services so that it can be invoked by clients also called ***service method***
  - should be **private**
    - if the method is created simply to assist a service method

    also called a ***support method***

Visibility modifiers

	`public`	`private`
Variables	NO! Violates encapsulation	YES! Enforces encapsulation
Methods	Yes, if method provides services to clients	Yes, if method supports other methods in the class

Accessor and Mutator Methods

data members are usually private

so to access them, we usually have *set* and *get* methods

mutator (setX): sets the data X and makes sure it stays in a coherent state

accessor (getX): returns the value of data X

```
public class Time {
  private int hour;
  private int minutes;
  private int seconds;

  public void setHour(_____________________) {
    ____________________________________________;
  }

  public void setMinute(_____________________) {
    ____________________________________________;
  }

  public void setSecond(_____________________) {
    ____________________________________________;
  }
```

```
  public int getHour(_____________________){
     ____________________________________________;
  }
  public int getMinute(_____________________){
     ____________________________________________;
  }
  public int getSecond(_____________________){
     ____________________________________________;
  }
  // constructor
  public Time(int h, int m, int s) {
     ____________________________________________;
     ____________________________________________;
     ____________________________________________;
   }
}
```

why should the *set* and *get* methods be public?

Static Members

Members can be:

- public / private / protected / package
- static (class members) / non-static (instance members)

Instance members (non **`static`**)

- associates a variable or method with **each object**
- invoked through the name of a **specific object**

```
myAccount.deposit(10);
system.out.print(yourCoin.face);
```

Class members (**`static`**)

- associates a variable or method with **each class** (shared by all objects of the class)
- invoked through the **name of the class**

```
System.out.print(Math.sqrt(25));
if (Character.isUpperCase('a'))
   …
```

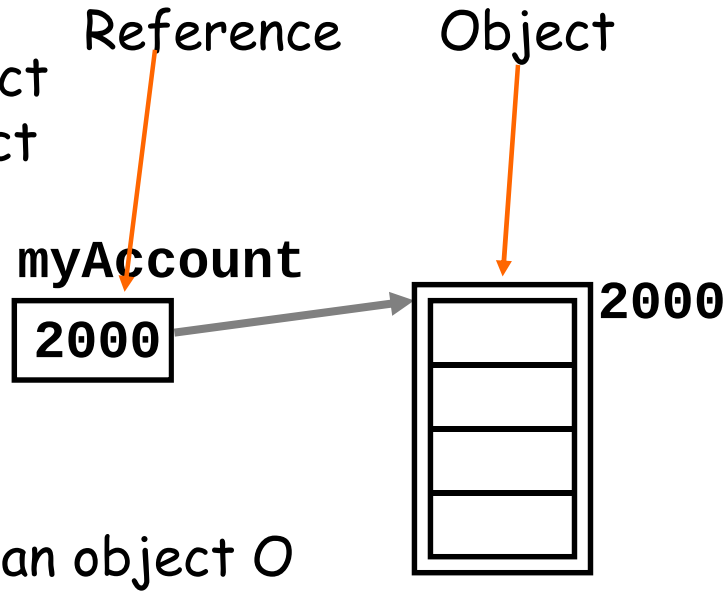
References

an object ≠ a reference to an object

reference=

holds the memory address of an object
can be seen as a "pointer" to an object

```
Account myAccount;
myAccount = new Account();
```

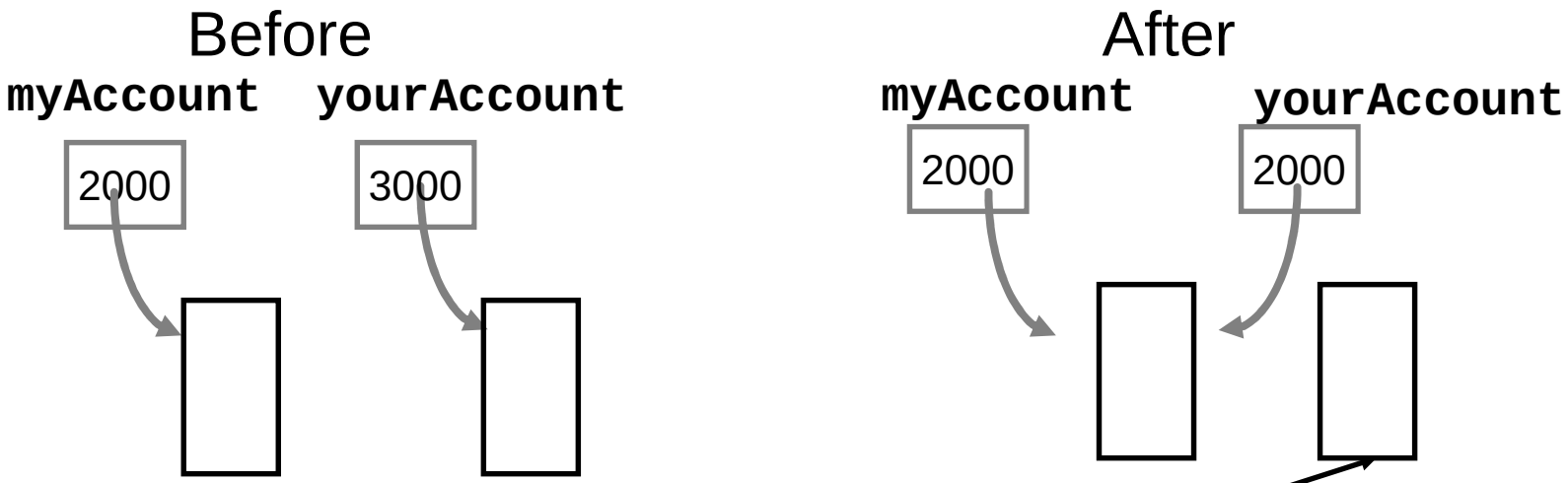


if a reference R contains the address of an object O
then we say that **R points to O**

Reference Assignment

For references, assignment copies the memory location

```
Account myAccount = new Account();
Account yourAccount = new Account();
yourAccount = myAccount;
```



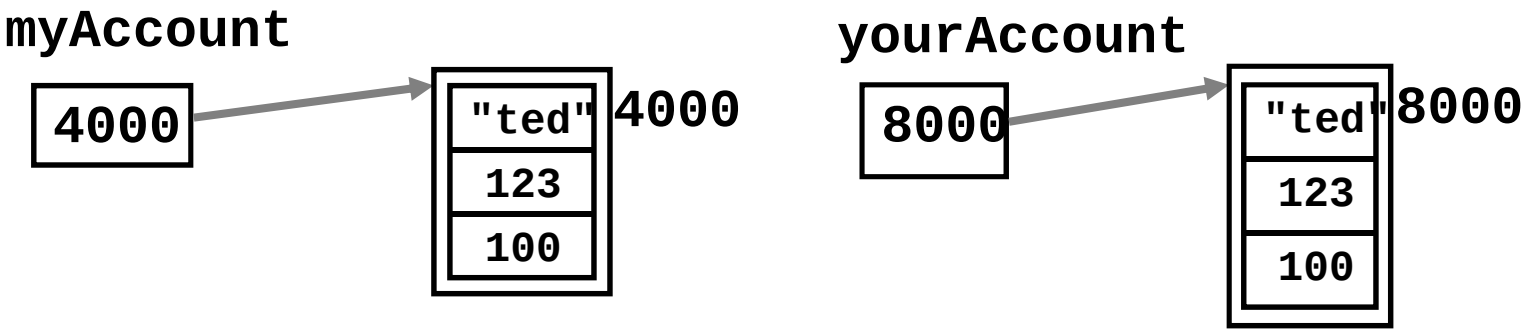
- and guess what… you have just lost access to
- if 2 references "point" to the same objects, they are aliases of each other

Equality of References

The == operator:

- compares equality of references
- returns **true** if the **references are aliases** of each other
  - ie if they **point** to the same object
  - **NOT if the objects pointed to have the same content**

```
Account myAccount = new Account("ted", 123, 100);
Account yourAccount = new Account("ted", 123, 100);
```




```
if (myAccount == yourAccount)   // true or false?
   System.out.print("the same");
```

Equality of Objects

The method **equals** :

- is defined for all objects
- but unless we redefine it in our class, it has the same semantics as the == operator
- we can redefine it to return **true** under whatever conditions we think are appropriate (ex. equality of content, not address)

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

  boolean equals(Account anotherAcc) {

     return (this.name.equals(anotherAcc.name) && (this.balance ==
     anotherAcc.balanace) && (this.acctNumber == anotherAcc.acctNumber))

  }
…
```

```
if (myAccount.equals(yourAccount))
   System.out.print("same content");
if (myAccount == yourAccount)
   System.out.print("same object");
```

Pass by Value vs Pass by Reference

- Pass by value
  - all primitive types are always passed by value
  - the formal parameter is a copy of the actual parameter
  - the method modifies the copy
  - but the actual parameter is never modified
- Pass by reference
  - object parameters are always passed by reference
  - the actual parameter and the formal parameter become aliases of each other
  - the method can modify the actual parameters
  - we copy the reference; not the object

Conclusion

if argument is a primitive type:

A method cannot change the value of the argument

if argument is a reference:

A method can change the value of an instance variable of an objet passed as argument

Copy Constructors

A copy constructor:

- A constructor with only one argument of the same type as the class
- Creates a separate, independent object that is copy of the argument object

Mutable vs. Immutable Classes

- *Immutable class*
  - A class that contains no methods (other than constructors) that change the data in an object of the class
  - It is safe to return a reference to an immutable object because the object cannot be changed

    ex: the `String` class and `Wrapper` classes

Mutable vs. Immutable Classes

- *Mutable class*
  - A class that contains public methods that can change the data in its objects
  - **Never write a method that returns a mutable object**
  - Instead, use a copy constructor, and return a copy of the object

The End