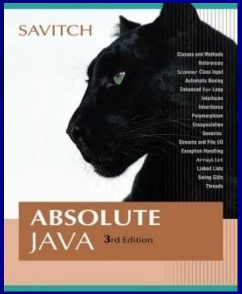
Comp 248
Introduction to Programming
Chapter 4 - *Defining Classes*

Part A

Dr. Aiman Hanna
Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

These slides has been extracted, modified and updated from original slides of Absolute Java 3rd Edition by Savitch; which has originally been prepared by Rose Williams of Binghamton University. Absolute Java is published by Pearson Education / Addison-Wesley.



UNIVERSITÉ Concordia UNIVERSITY

PEARSON Addison Wesley

Class Definitions

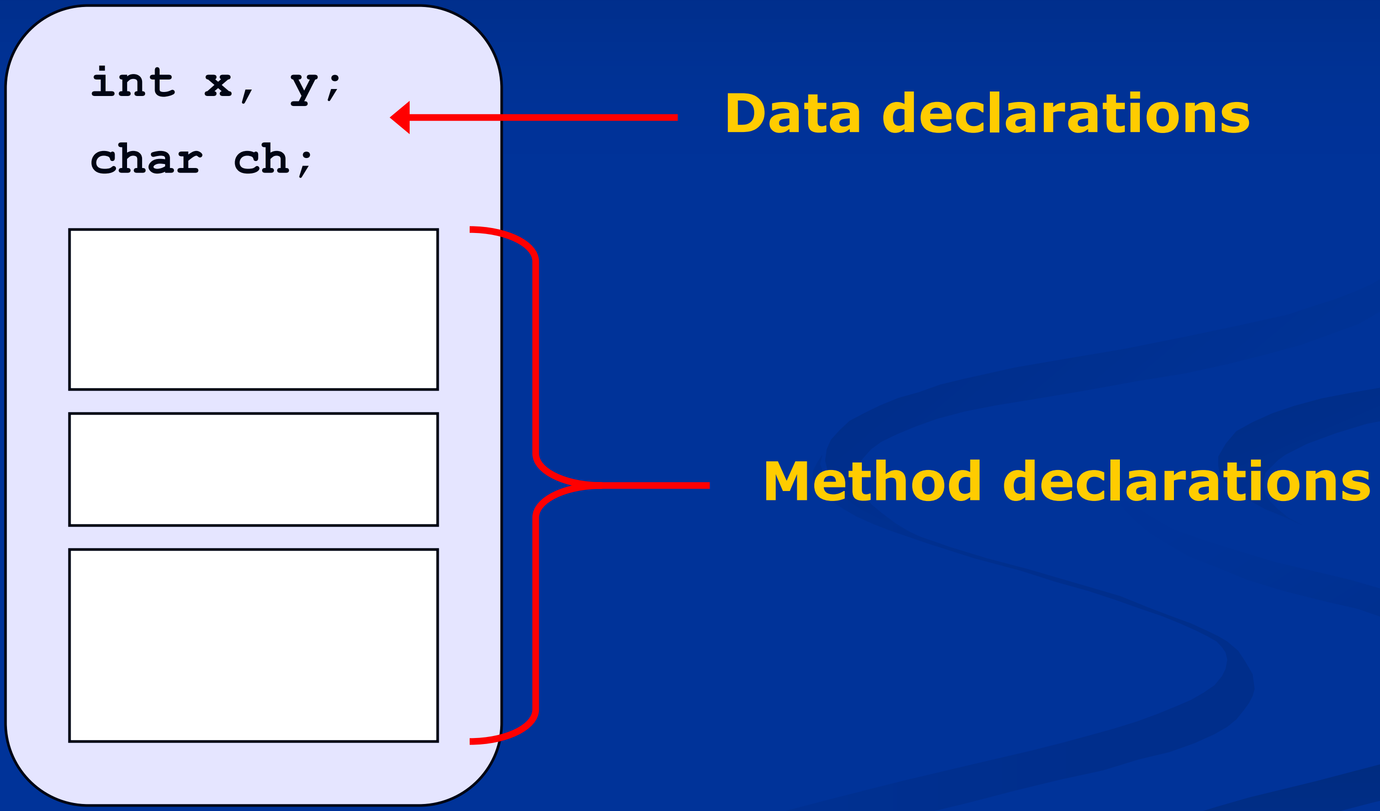
- We have already been using some of the predefined classes, i.e. **`String`** and **`Scanner`** classes

- We can add/define our own classes to the language

- A class determines:
  - 1) *Attributes, or Instance Variables*: the types of data that an object can contain,
  - 2) *Methods*: the actions it can perform

- Once a new class is defined, *objects*, or *instances*, can be created from this class

Class Definitions



```
int x, y;

char ch;
```

Data declarations

Method declarations

The `new` Operator

- An object of a class is named or declared by a variable of the class type:

  ```
  ClassName  objectName;
  ```

- The **new** operator **must** then be used to create the object and associate it with its variable name (however, some few exceptions do exist):

  ```
  objectName = new ClassName();
  ```

- These can be combined as follows:

  ```
  ClassName objectName = new ClassName();
  ```

Example:

```
Car c1 = new Car(); // Car is the class name and
                    // c1 is the object name
                    // IMPORTANT NOTE: In fact,
                    // c1 is a pointer/reference
                    // to the object
```

Instance Variables and Methods

- Instance variables (attributes) can be defined as in the following two examples
  - Note the **public** modifier (for now):

```
public int  numberOfDoors;
public double  Price;
```

- In order to refer to a particular instance variable, preface it with its object name as follows:

```
c1.price
c2.price
c1.numberOfDoors
```

c1 & c2 are just two objects from the class

Instance Variables and Methods

- Method definitions are divided into two parts: a *heading* and a *method body*:

```
public void myMethod()          ←    // Heading
{
      code to perform some action    // Body
      and/or compute a value
}
```

- Methods are invoked using the name of the calling object and the method name as follows:

```
objName.methodName();
```

Example:

```
C1.getNumberOfDoors();
```

- Invoking a method is equivalent to executing the method body

File Names and Locations

- Reminder: a Java file must be given the same name as the class it contains with an added **`.java`** at the end
    - For example, a class named **`Car`** must be in a file named **`Car.java`**
- For now, your program and all the classes it uses should be in the same directory or folder

More About Methods

- There are two kinds of methods:
  - Methods that compute/perform an action then return a value
  - Methods that compute/perform an action then does not return a value
    - This type of method is called a **`void`** method; in other words, it returns **`void`**
- Notice that in both cases, the function do indeed perform an action

More About Methods

- A method that returns a value must specify the type of that value in its heading:

```
 public typeReturned methodName(paramList)
```

Note: paramList is optional

Examples:

```
  public double getPrice();
  public int getNumOfDoors();
  public void setNumOfDoors(int nd); // nd is just
                                     // a name
```

`main` is a `void` Method

- A program in Java is just a class that has a **`main`** method
- When you give a command to run a Java program, the run-time system invokes the method **`main`**
- Note that **`main`** is a **`void`** method, as indicated by its heading:

```
public static void main(String[] args)
```

`return` Statements

- The body of both types of methods contains a list of declarations and statements enclosed in a pair of braces

```
public <void or typeReturned> myMethod()
{
        declarations                     Body
        statements
}
```

`return` Statements

- The body of a method that returns a value must also contain one or more **`return`** statements
  - A **`return`** statement specifies the value returned and ends the method invocation:

    ```
    return Expression;
    ```

  - **`Expression`** can be any expression that evaluates to something of the type returned listed in the method heading

`return` Statements

- A **`void`** method need not contain a **`return`** statement, unless there is a situation that requires the method to end before all its code is executed

- In this context, since it does not return a value, a **`return`** statement is used without an expression:

```
return;
```

Method Definitions

- An invocation of a method that returns a value can be used as an expression anyplace that a value of the returned type can be used:

```
double pr;
pr = c1.getPrice();
```

- An invocation of a **void** method is simply a statement:

```
objectName.methodName();
```

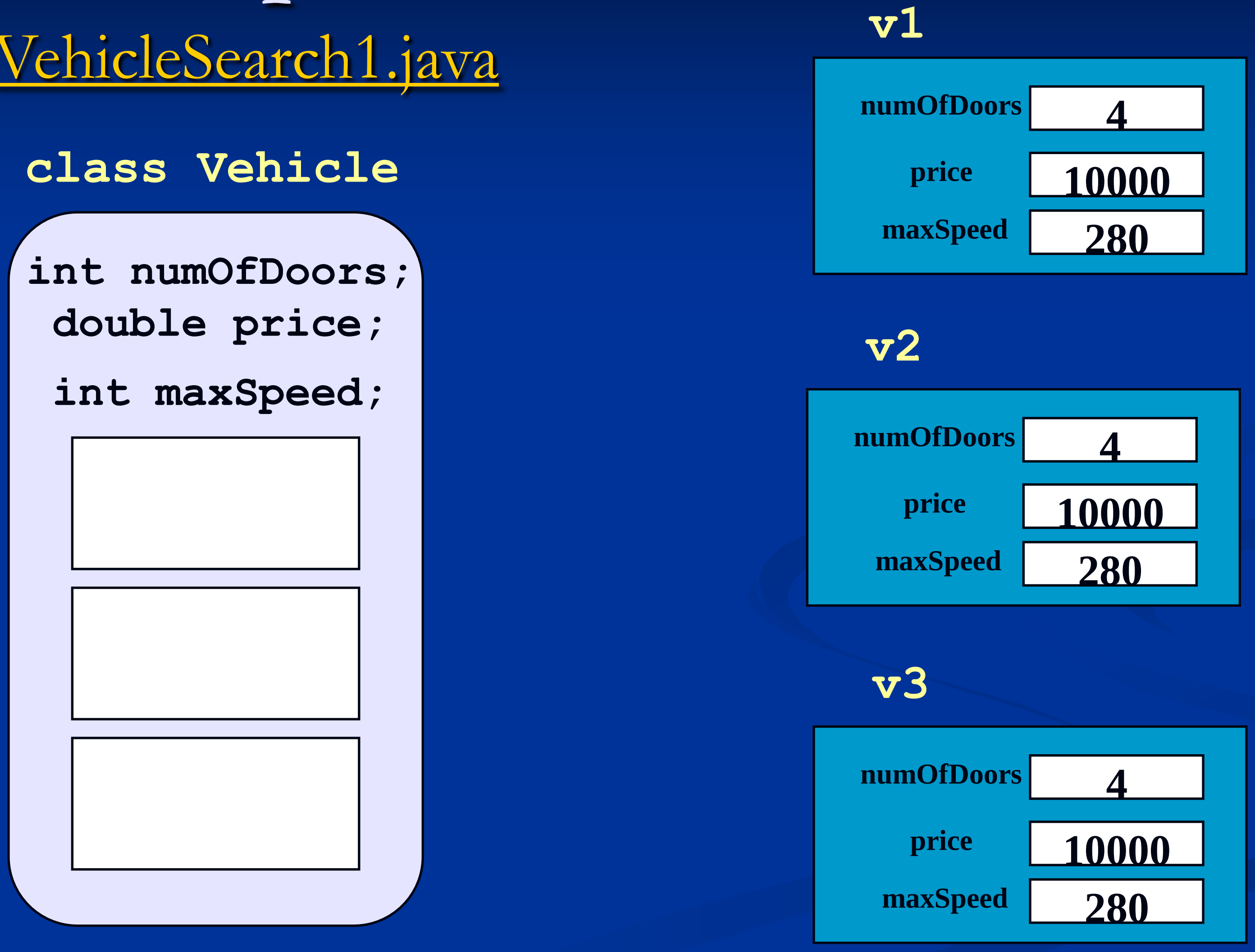
Examples:

```
    c1.setPrice(20000);
    c1.showModel();
```

- VehicleSearch1.java (MS-Word file)

Example: The Vehicle Class

- See VehicleSearch1.java



class Vehicle

```
int numOfDoors;
 double price;

 int maxSpeed;
```

v1

numOfDoors	4
price	10000
maxSpeed	280

v2

numOfDoors	4
price	10000
maxSpeed	280

v3

numOfDoors	4
price	10000
maxSpeed	280

v1, v2 & v3 upon creation

Constructors

- A *constructor* is a special kind of method that is designed to initialize the instance variables for an object:

```
public ClassName(anyParameters){code}
```

  - A constructor must have the same name as the class
  - A constructor has no type returned, not even **`void`**

- VehicleSearch2.java (MS-Word file)

`public` and `private` Modifiers

- The modifier **`public`** means that there are no restrictions on where an instance variable or method can be used

- The modifier **`private`** means that an instance variable or method cannot be accessed by name outside of the class

- VehicleSearch3.java (MS-Word file)

- VehicleSearch4.java (MS-Word file)

Include a No-Argument Constructor

- You should include a *default*, or *no-argument* constructor as part of your program. Default constructors will be discussed later in full details.

- If you do not include any constructors in your class, Java will automatically create a *default* or *no-argument* constructor that takes no arguments, performs no initializations, but allows the object to be created

- If you include even one constructor (possibly non-default) in your class, Java will not provide this default constructor

Local Variables

- A variable declared within a method definition is called a *local variable*
  - All variables declared in the **`main`** method are local variables
  - All method parameters are local variables

- If two methods each have a local variable of the same name, they are still two entirely different variables

Global Variables

- Some programming languages include another kind of variable called a *global* variable
- The Java language does **not** have global variables

Blocks

- A *block* is another name for a compound statement, that is, a set of Java statements enclosed in braces, **{ }**
- A variable declared within a block is local to that block, and cannot be used outside the block
- Once a variable has been declared within a block, its name cannot be used for anything else within the same method definition

Declaring Variables in a `for` Statement

- You can declare one or more variables within the initialization portion of a **`for`** statement

- A variable so declared will be local to the **`for`** loop, and cannot be used outside of the loop

- If you need to use such a variable outside of a loop, then declare it outside the loop

- Statements14.java (MS-Word file)
- Statements15.java (MS-Word file)

Parameters of a Primitive Type

- A method can accept no parameters, one parameter, or few of them (parameter list)
  - These parameter(s) are referred to as *formal parameters*

```
public void setVehicleInfo(int nd, double pr, int mxsp)
```

- When a method is invoked, the appropriate values must be passed to the method in the form of *arguments*, and must be in the right order
  - These arguments are called *actual parameters*

```
c1.setVehicleInfo(4, 12500.99, 280);
```

Parameters of a Primitive Type

- The type of each argument must be compatible with the type of the corresponding parameter. The following two statements use the method correctly

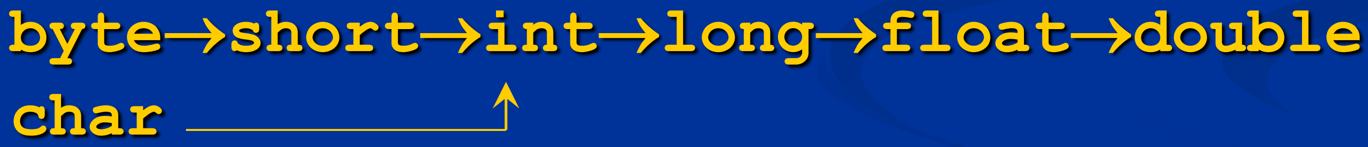
```
c1.setVehicleInfo(4, 12500.99, 280);

int n = 5, m = 260;
double p = 19700.95;
c1.setVehicleInfo(n, p, m);
```

- NOTE: In both examples, the value of each argument (not the variable name) is the one plugged into the corresponding method parameter
  - This method of plugging in arguments for formal parameters is known as the ***call-by-value mechanism***

- MethodParameters1.java (MS-Word file)

Parameters of a Primitive Type

- If argument and parameter types do not match exactly, Java will attempt to make an automatic type conversion
  - A primitive argument can be automatically type cast from any of the following types, to any of the types that appear to its right:



`byte→short→int→long→float→double`

`char` → **`int`**

Methods That Return a Boolean Value

- An invocation of a method that returns a value of type **`boolean`** returns either **`true`** or **`false`**

- Therefore, it is common practice to use an invocation of such a method to control statements and loops where a boolean expression is expected
  - i.e. within **`if-else`** statements, **`while`** loops, etc.

Comparing Objects of the Same Class for Equality

- You cannot use **`==`** to compare objects
  - VehicleCompare1.java (MS-Word file)
- Instead use methods such as user-defined **`equals`**, or **`toString`** to compare the objects
  - VehicleCompare2.java (MS-Word file)
  - VehicleCompare3.java (MS-Word file)
  - VehicleCompare4.java (MS-Word file)

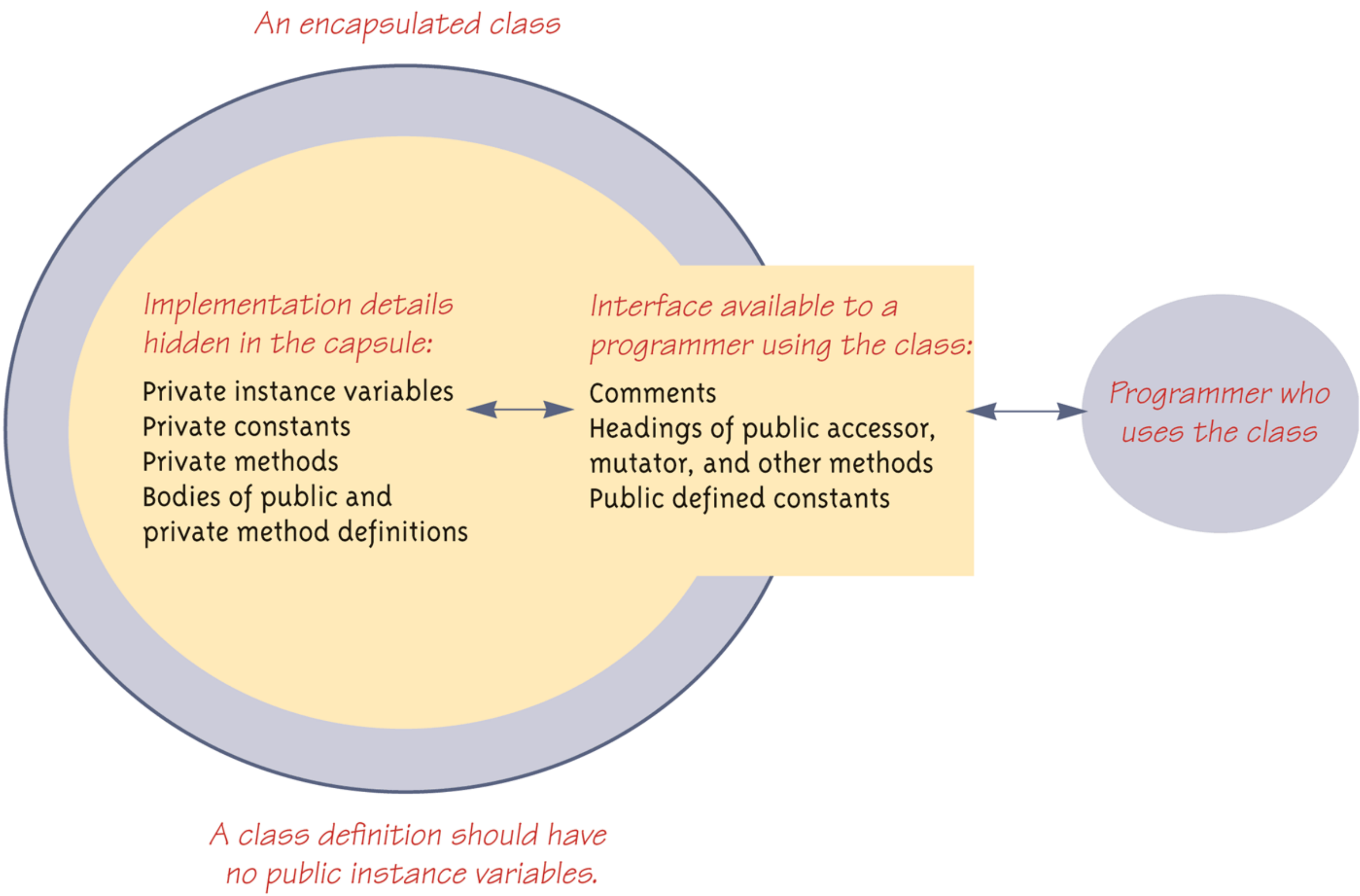
Accessor and Mutator Methods

- *Accessor* methods allow the programmer to obtain the value of an object's instance variables
  - The data can be accessed but not changed
  - The name of an accessor method typically starts with the word **`get`**

- *Mutator* methods allow the programmer to change the value of an object's instance variables in a controlled manner
  - Incoming data is typically tested and/or filtered
  - The name of a mutator method typically starts with the word **`set`**

Encapsulation

Display 4.10 **Encapsulation**



An encapsulated class

Implementation details hidden in the capsule:

Private instance variables
Private constants
Private methods
Bodies of public and private method definitions

Interface available to a programmer using the class:

Comments
Headings of public accessor, mutator, and other methods
Public defined constants

Programmer who uses the class

A class definition should have no public instance variables.

A Class Has Access to Private Members of All Objects of the Class

- Within the definition of a class, private members of **any** object of the class can be accessed, not just private members of the calling object

  - For example, see the equals function in VehicleCompare2.java (MS-Word file)

The function has access to the private date of the passed object, *vec*