edited from Appendix D: Introduction to RAPTOR
Prelude to Programming: Concepts and Design, 5th edition
by Elizabeth Drake and Stewart Venit
Addison-Wesley Pub. 2011

Introduction to RAPTOR: Data Files and OOP Mode

Creating and Displaying Data Files

In RAPTOR we can create data files and read from the files. However, sorting, inserting records, or merging two data files requires some fancy footwork but if you're very ambitious, try it!

The **`Redirect_Output`** Procedure

To create a data file, you use a **`Call`** to the **`Redirect_Output`** procedure. RAPTOR provides two versions of this procedure.

1. A filename is used as an argument to **`Redirect_Output`**, as shown in the following examples:
   - **`Redirect_Output("sample.txt")`**
   - **`Redirect_Output("C:\MyDocuments\John.Doe\sample")`**

   Note that in the first example, only the filename is given. In this case, the specified text file will be created in the same directory as the current RAPTOR program. In the second example, the full path to the file is given. Also, in the second example, no file extension is specified. In this case, the file **`sample`** will be created with no extension.

2. You can either turn on or off **`Redirect_Output`** by including a simple **`yes/true`** or **`no/false`** argument, as follows:
   - **`Redirect_Output(True)`** or **`Redirect_Output(yes)`**
   - **`Redirect_Output(False)`** or **`Redirect_Output(no)`**

Now, the output must be redirected to the data file by using a **`Call`** to the **`Redirect_Output`** procedure. The name of the data file is used as the argument. This filename must be inside quotation marks (as shown in the examples above).

Next, create the code to input data. One variable is required for each field of the records in the data file. The **`Output`** box will **`PUT`** the value of those variables into each record. For example, to create a data file with records that have two fields, **`Name`** and **`Salary`**, two variables are required (probably called **`Name`** and **`Salary`**). As values for different employees are input, each **`Name`** and **`Salary`** will be stored in the data file on a single line.

After the data has been entered, the **`Redirect_Output`** must be turned off. A **`Call`** is used to call the **`Redirect_Output`** procedure again, but this time it's turned off by using either **`no`** or **`false`** as the argument.

Figure 1 (following page) shows a RAPTOR flowchart that will write two records, each with two fields, to a data file named **`sample.txt`**. Figure 2 shows the contents of the file created (and opened in Notepad).

The `Redirect_Input` Procedure

To display the contents of a data file, the **`Redirect_Input`** procedure is used. This works similarly to the **`Redirect_Output`** procedure.

In a **`Call`** to **`Redirect_Input`**, the filename of the file to be read is used as the argument as follows:

```
Redirect_Input("sample.txt")
```

The records are read, normally, within a loop. This is accomplished with **`GET`** statements. **`Input`** boxes are used to **`GET`** each record (in this example, the records consist of the names and salaries). Nothing needs to be entered as a prompt. **`Output`** boxes are used to display the output of each record. The output is displayed in the **`Master Console`**.

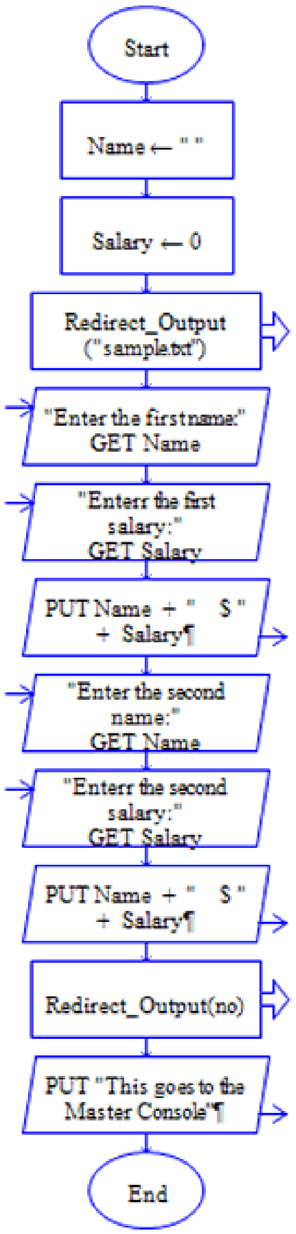
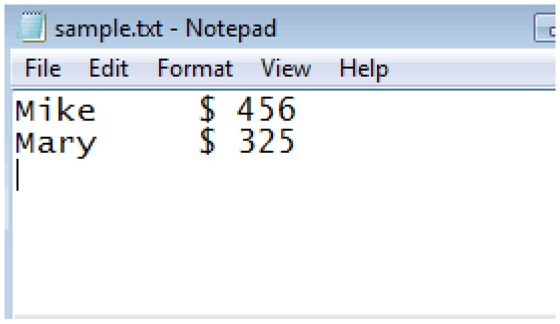

Start
Name ← " "
Salary ← 0
Redirect_Output ("sample.txt")
"Enter the firstname:" GET Name
"Enterr the first salary:" GET Salary
PUT Name + " $ " + Salary¶
"Enter the second name:" GET Name
"Enterr the second salary:" GET Salary
PUT Name + " $ " + Salary¶
Redirect_Output(no)
PUT "This goes to the Master Console"¶
End

Figure 1 Program to write records to a data file



```
Mike      $ 456
Mary      $ 325
```

Figure 2 Text file created

The **`End_Of_Input`** Function

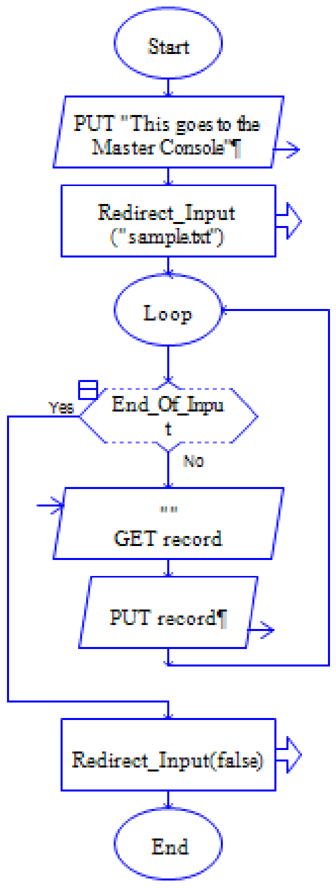
RAPTOR's built-in function, **`End_Of_Input`**, can be used as the test condition of a loop. When reading records in a data file, if this function is used as the test condition, RAPTOR will end the loop when all the records have been read.

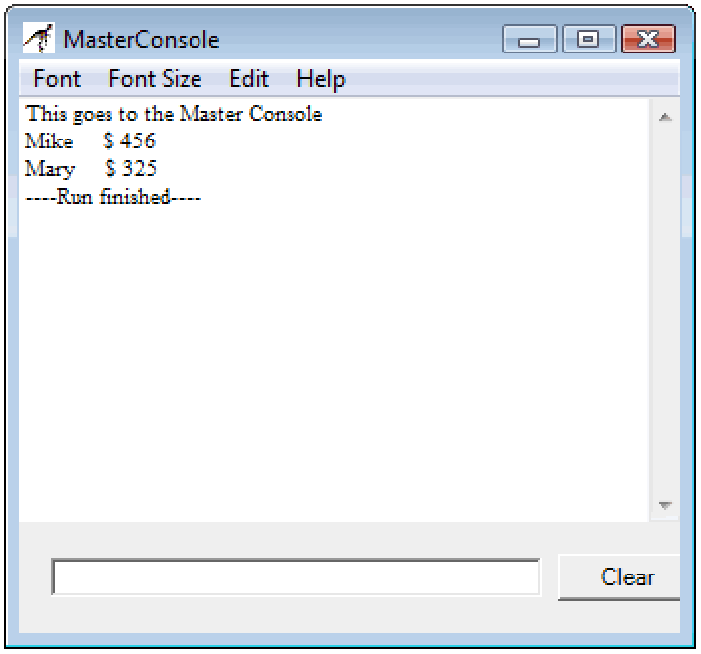
When all the records have been read and written to the **`Master Console`**, the **`Redirect_Input`** procedure must be turned off with a **`Call`** to the procedure using **`False`** or **`no`** for the argument.

How the Contents of the File are Stored

The **`Redirect_Input`** procedure does not separate each field in a record. Rather, each record is stored as one line of string data. Each Input line reads all the fields in one record (or everything that is on a single line in the data file). Therefore, the records can be output to the **`Master Console`** but the fields cannot be manipulated easily to sort, insert, or merge. This can be done, but it requires advanced programming.

Figure 3 shows a sample of the code to read records from a data file (**`sample.txt`**) and the **`Master Console`** display.





```
This goes to the Master Console
Mike    $ 456
Mary    $ 325
----Run finished----
```

Figure 3 Reading records from a data file and displaying them

Object-Oriented Mode

`Object-oriented` mode allows you to create classes with methods and attributes, instantiate objects, and experiment with Object-Oriented Programming (OOP).

To use RAPTOR in OOP, you must select **`Object-oriented`** mode, as shown in Figure 4.

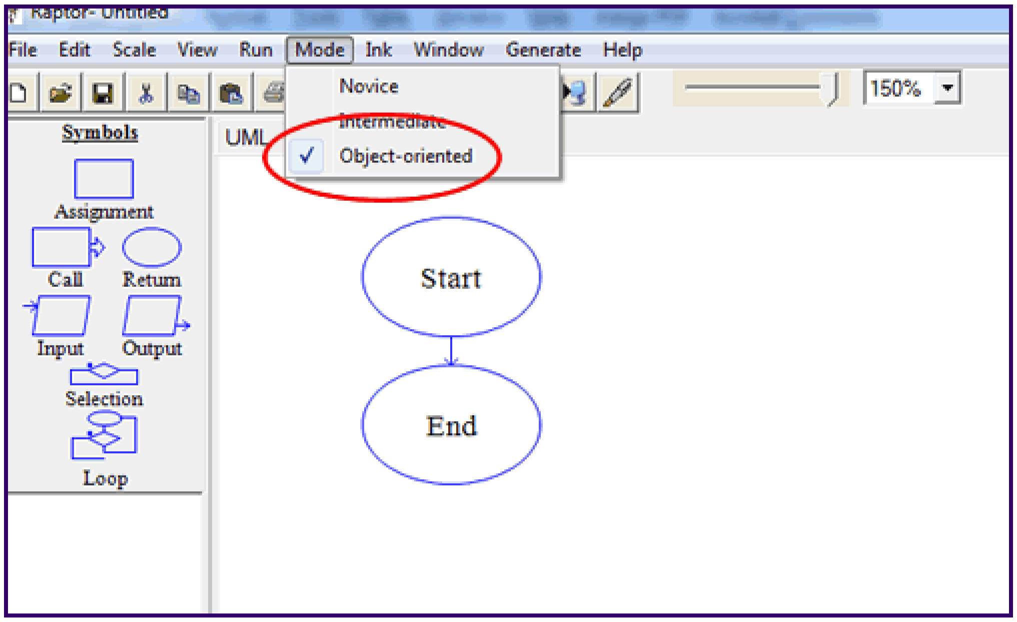


Figure 4 Selecting `Object-oriented` mode

You will see two tabs: **`UML`** and **`main`**. RAPTOR uses a type of UML to create the structure of an object-oriented program. The classes are created in the **`UML`** screen; therefore, click the **`UML`** tab. The button to add a new class is shown in Figure 5. Note that a new **`Return`** symbol has been added to the symbols.

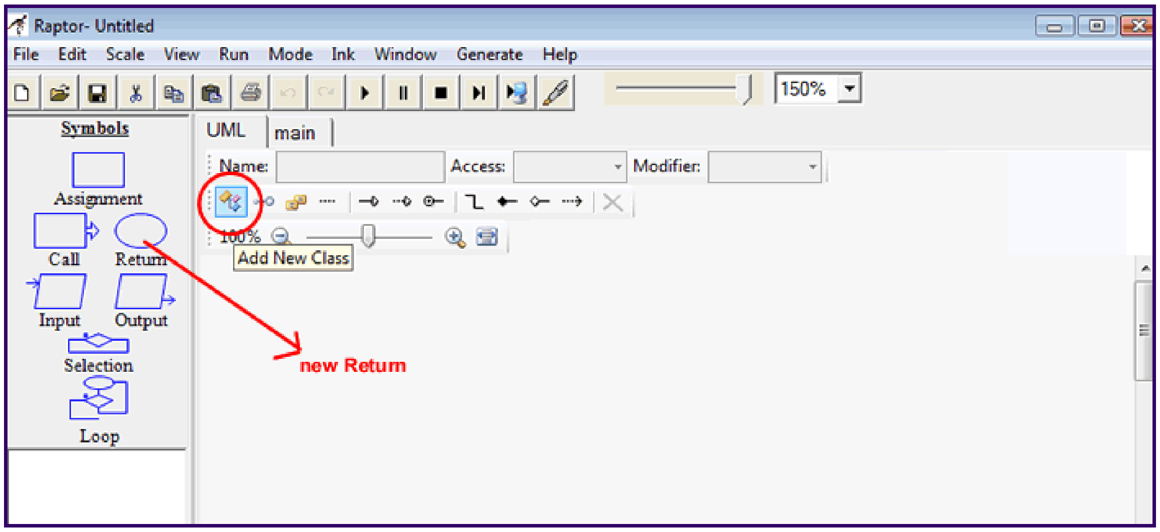
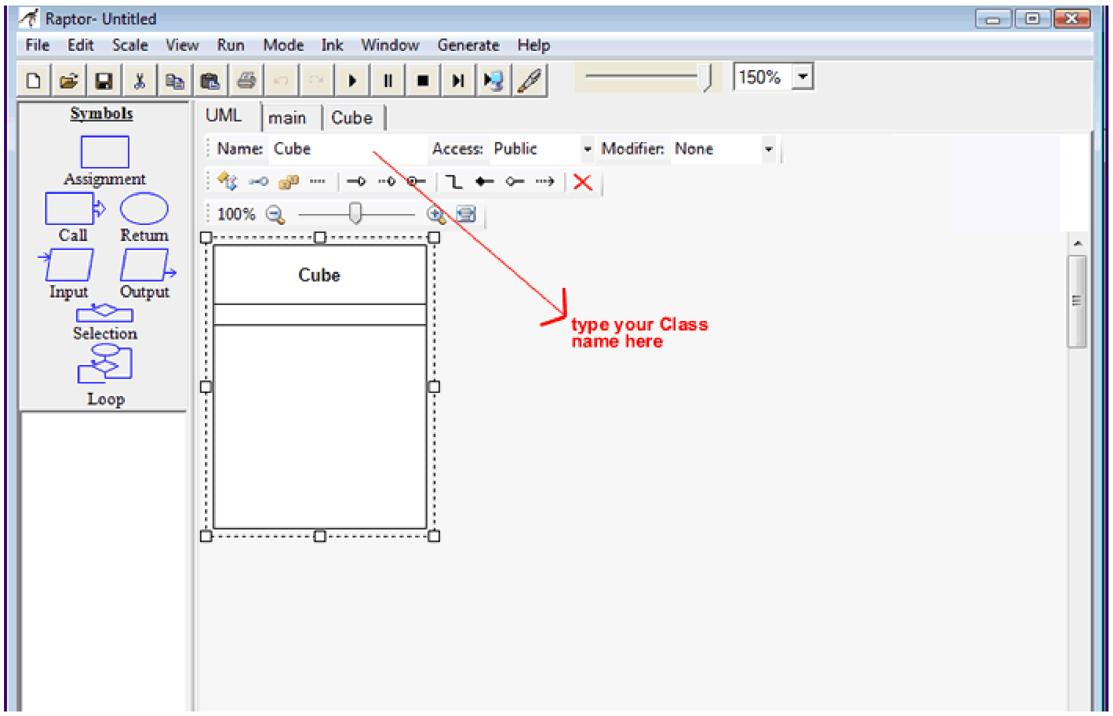


Figure 5 Adding a new class

Creating a Class

When you click the **`Add New Class`** button to add a new class, a **`Name`** box will appear. Enter a name for the **`Class`**, as shown in Figure 6.



Name: Cube   Access: Public   Modifier: None

Figure 6 Entering a `Class` name

In Figure 6, a **`Class`** named **`Cube`** has been created. Double-click inside the class (**`Cube`**) to add members (methods and attributes). In RAPTOR, note that attributes are called **`Fields`**. A new window opens to allow you to enter the members (see Figure 7).

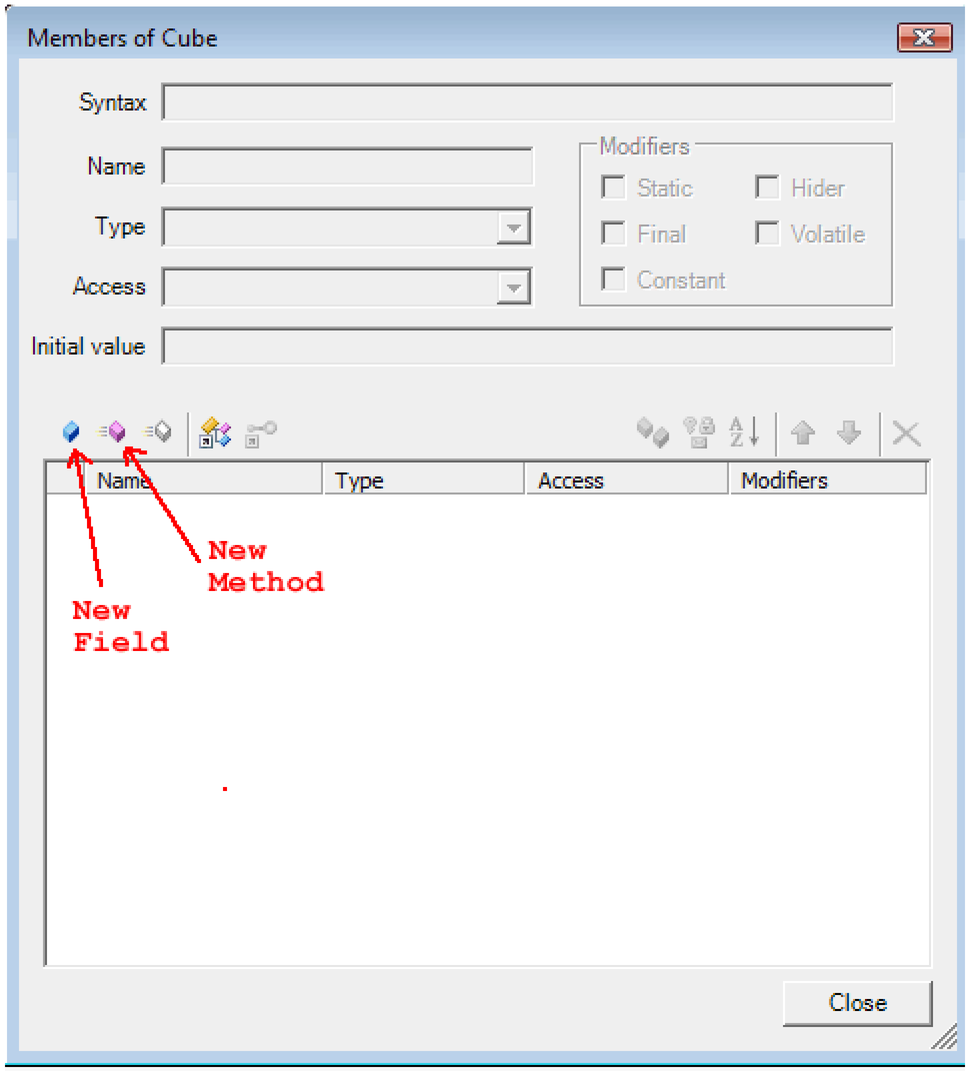


Figure 7 Adding members to a `Class`

From this point, examples will be used to demonstrate the features of OOP mode and indicate how to use them in a program.

Example: Using the Cube `Class` to Find the Volume of a Cube

We will use a class named **`Cube`** that takes the value of a side of a cube and computes the cube's volume. So we need the following:

attributes: **`Side`** (a number) and **`Volume`** (a number)

methods: **`SetSide()`**, **`GetSide()`**, **`ComputeVolume()`**, and **`GetVolume()`**

Figure 8 (following page) shows the **`Class Cube`** and its members.

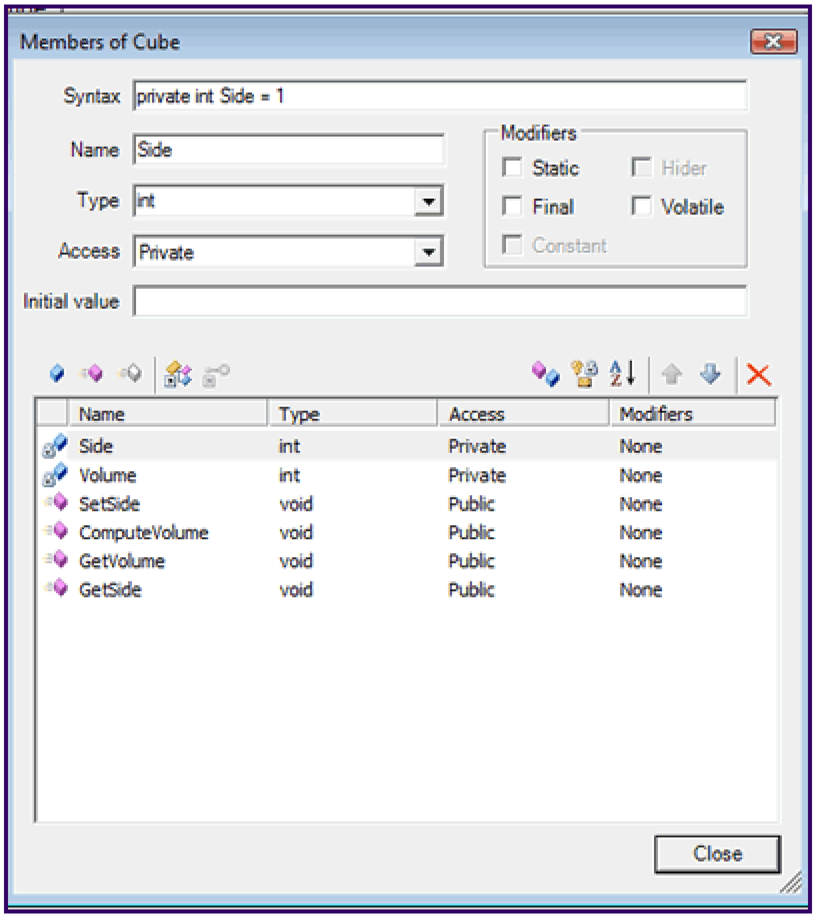
- Note the syntax for a **`Field`**: A **`Field`** must be given a data type. The type of **`Side`** and **`Volume`** is **`int`** and in this case, each field has been given an initial value of 1.
- Note the syntax for a **`Method`**. If the **`Method`** receives a value passed from **`main`**, you must include that parameter. For example,
    - The **`Method SetSide()`** is passed a value for the length of a side so the syntax for this **`Method`** is

      ```
      public void SetSide(int NewSide)
      ```
    - The **`Method ComputeVolume()`** uses the value of the side of a cube to do its calculations so it needs one parameter, the integer variable **`Side`**. The syntax is

      ```
      public void ComputeVolume(int Side)
      ```
    - The **`Method GetVolume()`** retrieves the value of the volume of the cube from **`ComputeVolume()`** so the syntax for this **`Method`** is

      ```
      public void GetVolume(int Volume)
      ```
    - The **`Method GetSide()`** does not need a parameter so the syntax is

      ```
      public void GetSide()
      ```

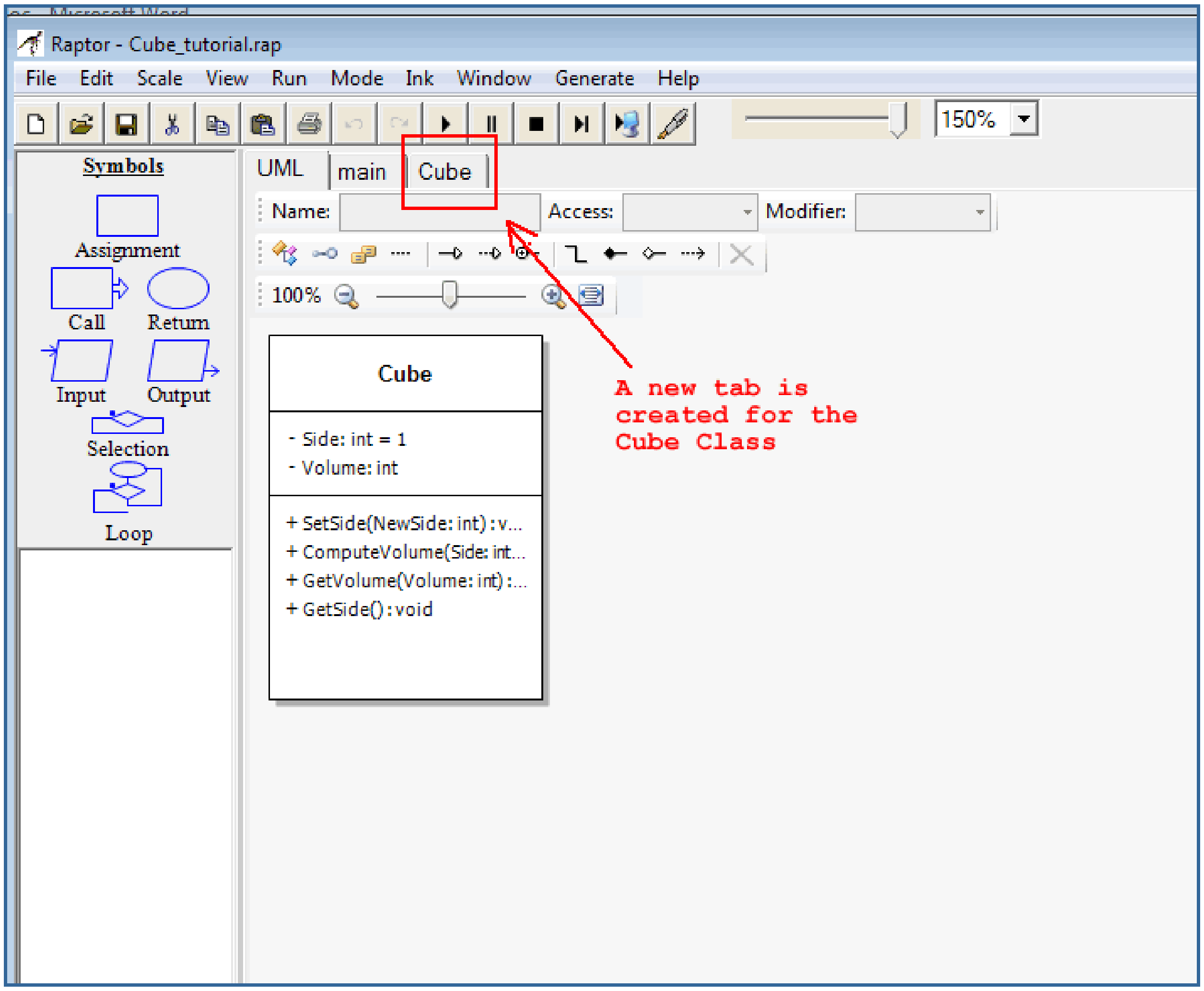


Syntax: private int Side = 1
Name: Side
Type: int
Access: Private
Initial value:

Name	Type	Access	Modifiers
Side	int	Private	None
Volume	int	Private	None
SetSide	void	Public	None
ComputeVolume	void	Public	None
GetVolume	void	Public	None
GetSide	void	Public	None

Figure 8 The `Class` `Cube` and its members

Once the **`Class`** has been created, a new tab is automatically added, with the name of the **`Class`** (see Figure 9). Now the code for each of the **`Class`’s** methods must be created. Click the **`Cube`** tab to see four new tabs—one for each **`Method`**, as shown in Figure 10.


Raptor - Cube_tutorial.rap
File Edit Scale View Run Mode Ink Window Generate Help
150%
Symbols
Assignment
Call
Return
Input
Output
Selection
Loop
UML
main
Cube
Name:
Access:
Modifier:
100%
A new tab is created for the Cube Class

Cube

- Side: int = 1
- Volume: int

+ SetSide(NewSide: int) : v...
+ ComputeVolume(Side: int...
+ GetVolume(Volume: int) :...
+ GetSide() : void

Figure 9 New tab for the `Class Cube`

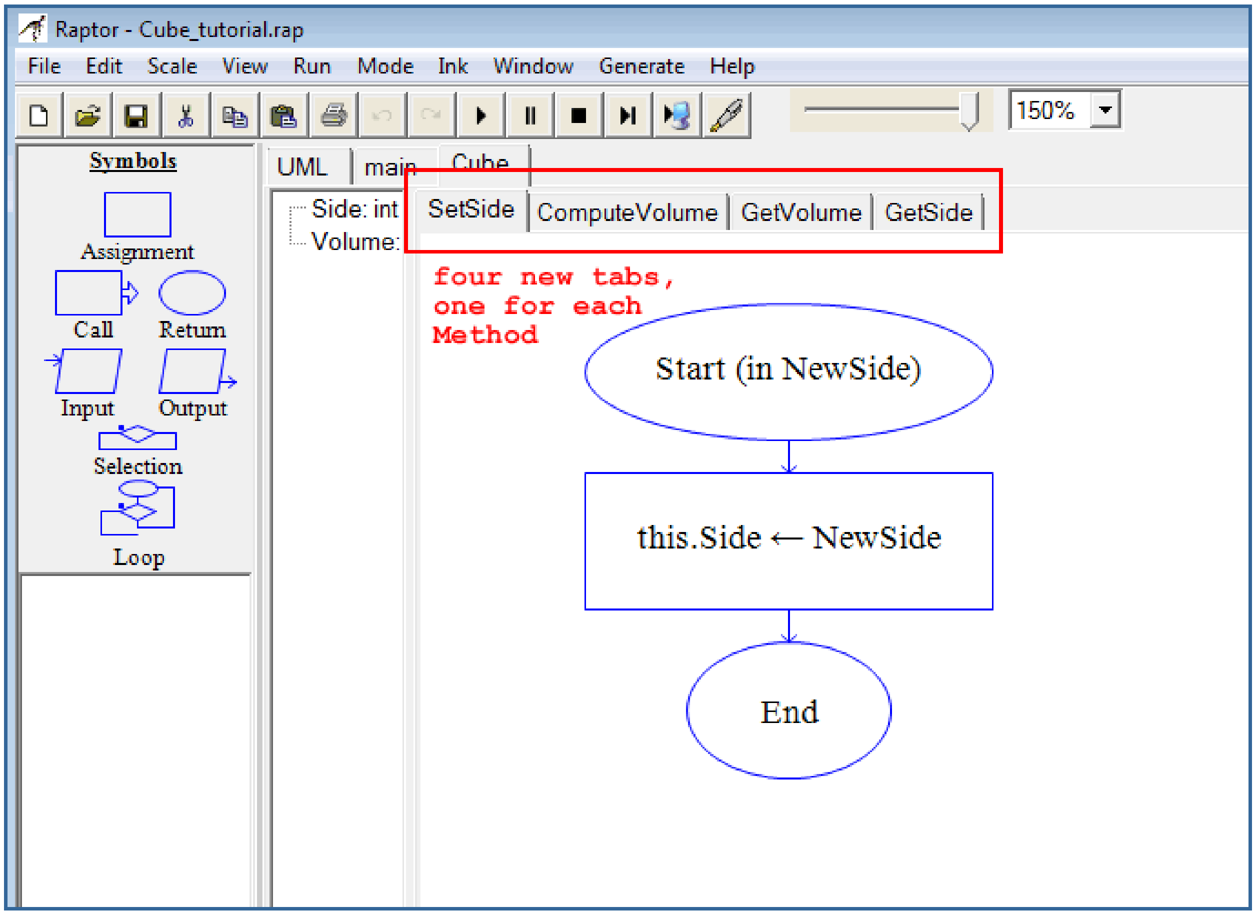


Figure 10 New tabs for each new `Method`

Code the Methods

The **`Methods`** for this program are as follows: **`SetSide(NewSide)`**, **`ComputeVolume(Side)`**, **`GetVolume(Volume)`**, and **`GetSide()`**.

`SetSide() Method:`

The **`SetSide()Method`** does one thing only. It sets the value of the side of a cube, as passed to it from the main program, to the variable **`NewSide`**. This assignment is done using the **`this`** keyword. The code for this method is shown in Figure 11.

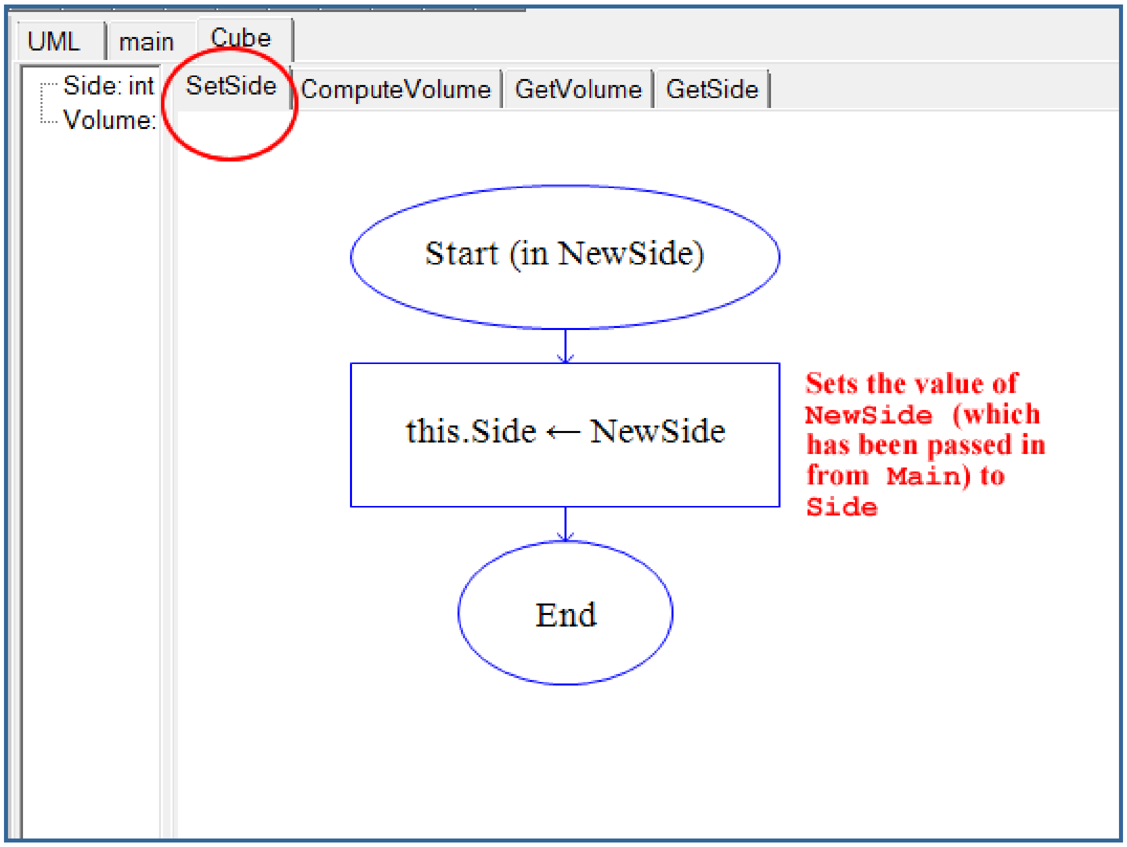


Figure 11 Code for the `SetSide()` method

`ComputeVolume(Side) Method:`

The **`ComputeVolume(Side)Method`** computes the volume of the cube. First, it must receive the value needed for the computation (**`Side`**). Then, it must do the computation by cubing the value. Finally, it needs to export this result when requested. Figure 12 shows the code.

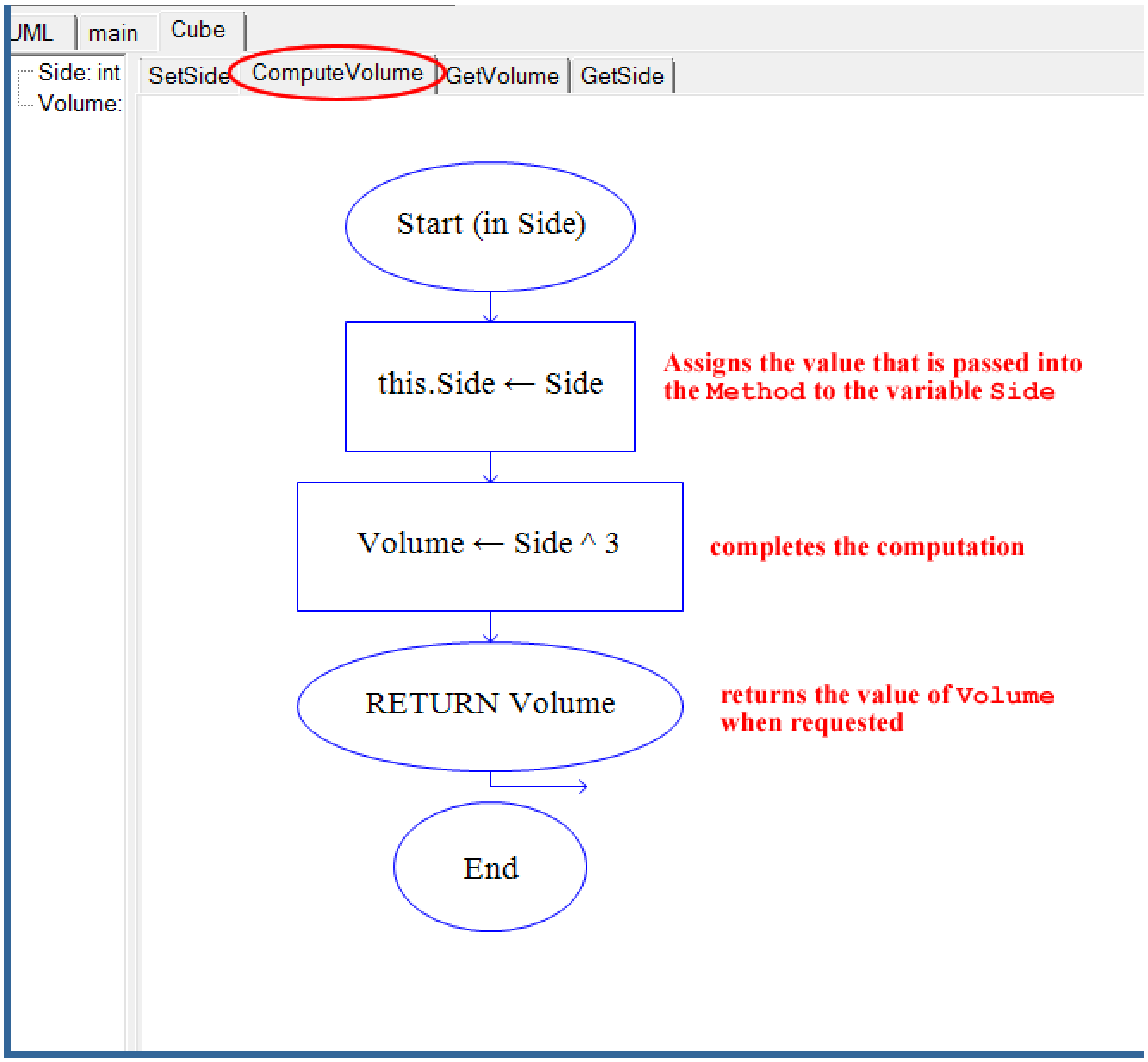

UML
main
Cube
Side: int
Volume:
SetSide
ComputeVolume
GetVolume
GetSide
Start (in Side)
this.Side ← Side
Volume ← Side ^ 3
RETURN Volume
End
Assigns the value that is passed into the Method to the variable Side
completes the computation
returns the value of Volume when requested

Figure 12 Code for the `ComputeVolume()` method

`GetVolume(Volume) Method:`

The **`GetVolume(Volume)Method`** retrieves the value of **`Volume`** when it is accessed and then returns it, as shown in Figure 13.

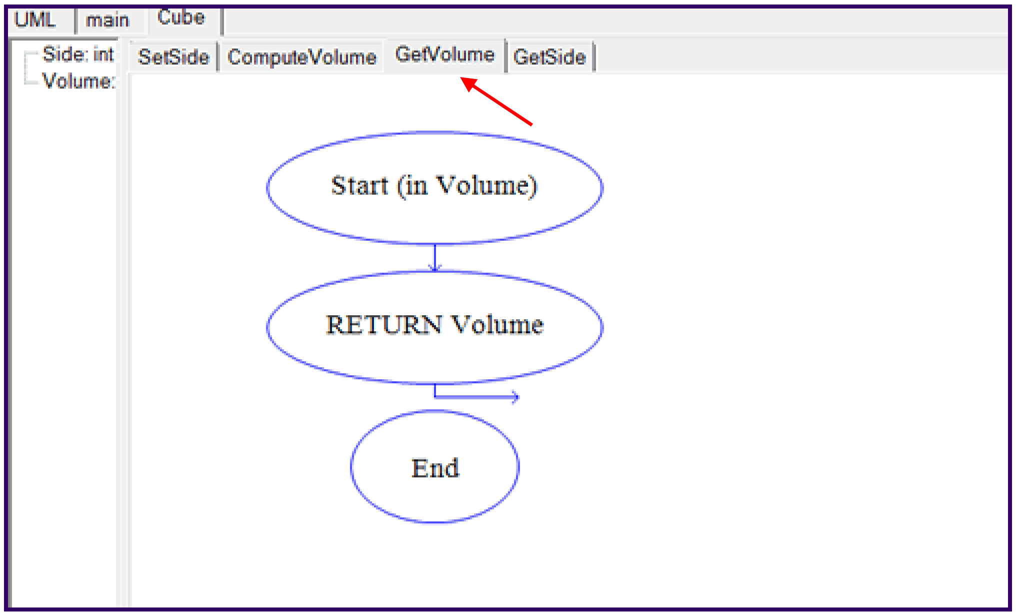


Figure 13 Code for the `GetVolume()` method

`GetSide()` Method:

The **`GetSide() Method`** retrieves the value of **`Side`** when accessed, as shown in Figure 14.

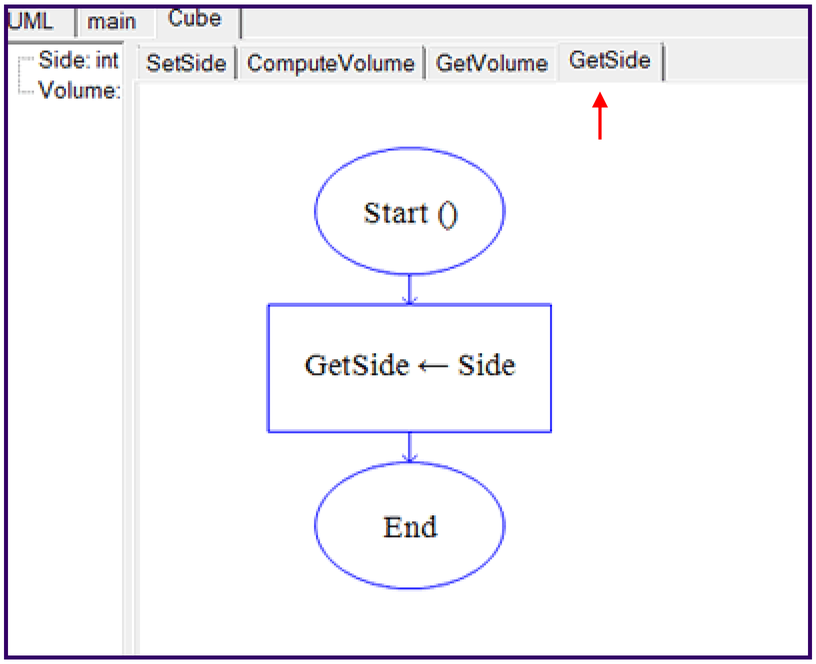
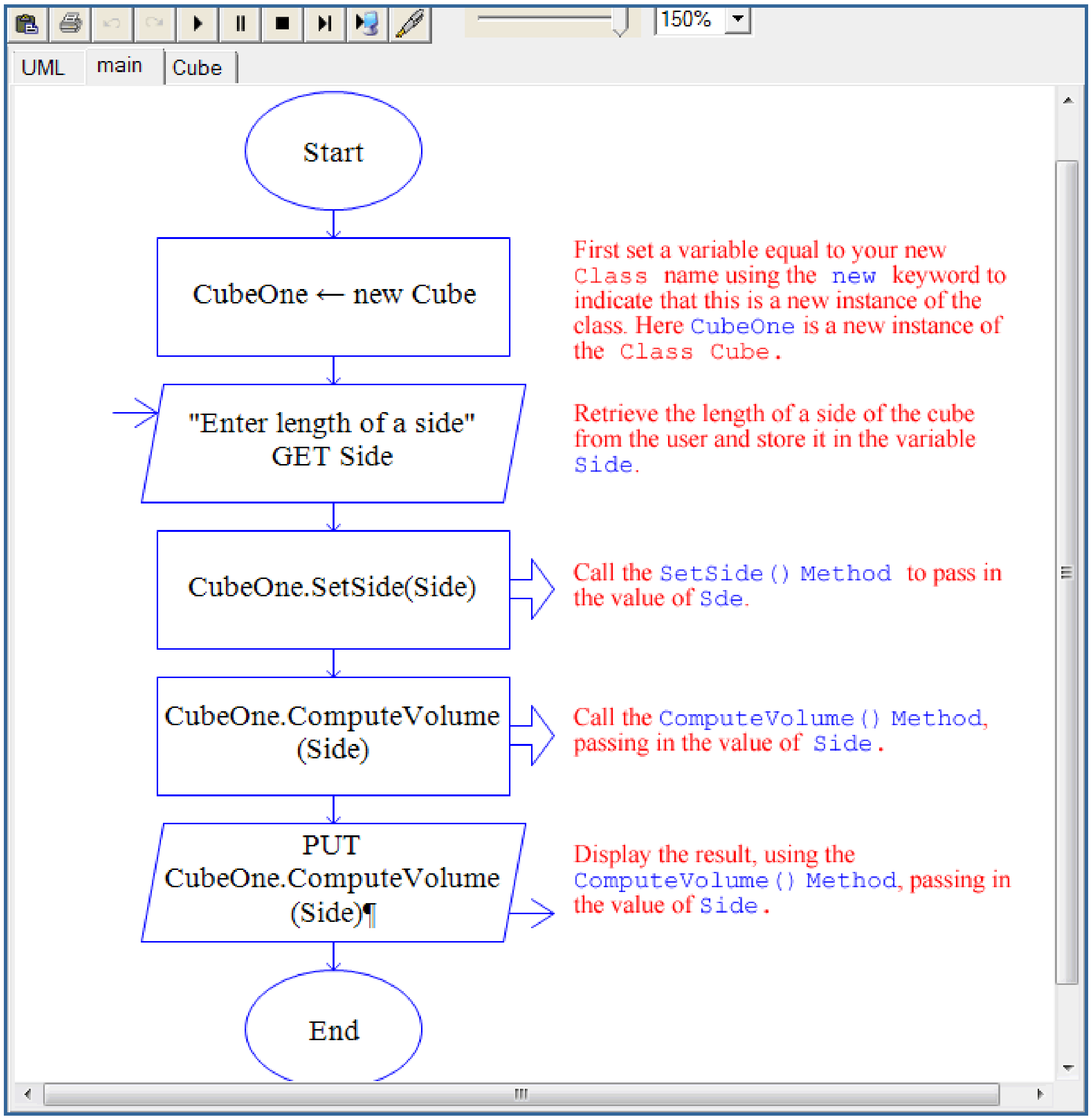


Figure 14 Code for the `GetSide()` method

The `Main` Program

Now the **`Main`** program can be created. The program for this example is extremely simple; it will allow the user to enter a value for the side of a cube, compute the volume of that cube, and display the result. This is accomplished by instantiating an object of type **`Cube`**, which we will call **`CubeOne`**, and using the methods and attributes of **`Cube`**. Figure 15 shows how this is done the RAPTOR OOP way.



```
CubeOne ← new Cube
"Enter length of a side"
GET Side
CubeOne.SetSide(Side)
CubeOne.ComputeVolume
(Side)
PUT
CubeOne.ComputeVolume
(Side)¶
```

First set a variable equal to your new `Class` name using the `new` keyword to indicate that this is a new instance of the class. Here `CubeOne` is a new instance of the `Class Cube`.

Retrieve the length of a side of the cube from the user and store it in the variable `Side`.

Call the `SetSide() Method` to pass in the value of `Sde`.

Call the `ComputeVolume() Method`, passing in the value of `Side`.

Display the result, using the `ComputeVolume() Method`, passing in the value of `Side`.

Figure 15 Code to input a side of a cube and output its volume

Inheritance and Polymorphism

Once you have mastered the basics: creating **`Classes`**, **`Fields`**, and **`Methods`**, and using **`dot notation`** in your program, you can use the OOP mode in RAPTOR to create and run more complicated programs.

You create child classes that inherit from a parent class in the UML screen. Figure 16 (following page) shows the association between a parent **`Class`** named **`Animal`** and two child **`Classes`** (subclasses) named **`Frog`** and **`Snake`**. Use the **`New Association`** button to set the inheritance between the parent and child, as indicated in Figure 16.

In this example, **`Frog`** and **`Snake`** inherit the **`showAnimalStuff() Method`** from **`Animal`** but each child class has its own **`Method`** for **`makeSound()`** and **`showAnimalType()`**. The OOP characteristics of both polymorphism and inheritance are demonstrated by this example. **[Special thanks** to George L. Marshall, Jr. from Calhoun Community College at the Research Park Campus in Huntsville, Alabama for the **`Animal`** example.]

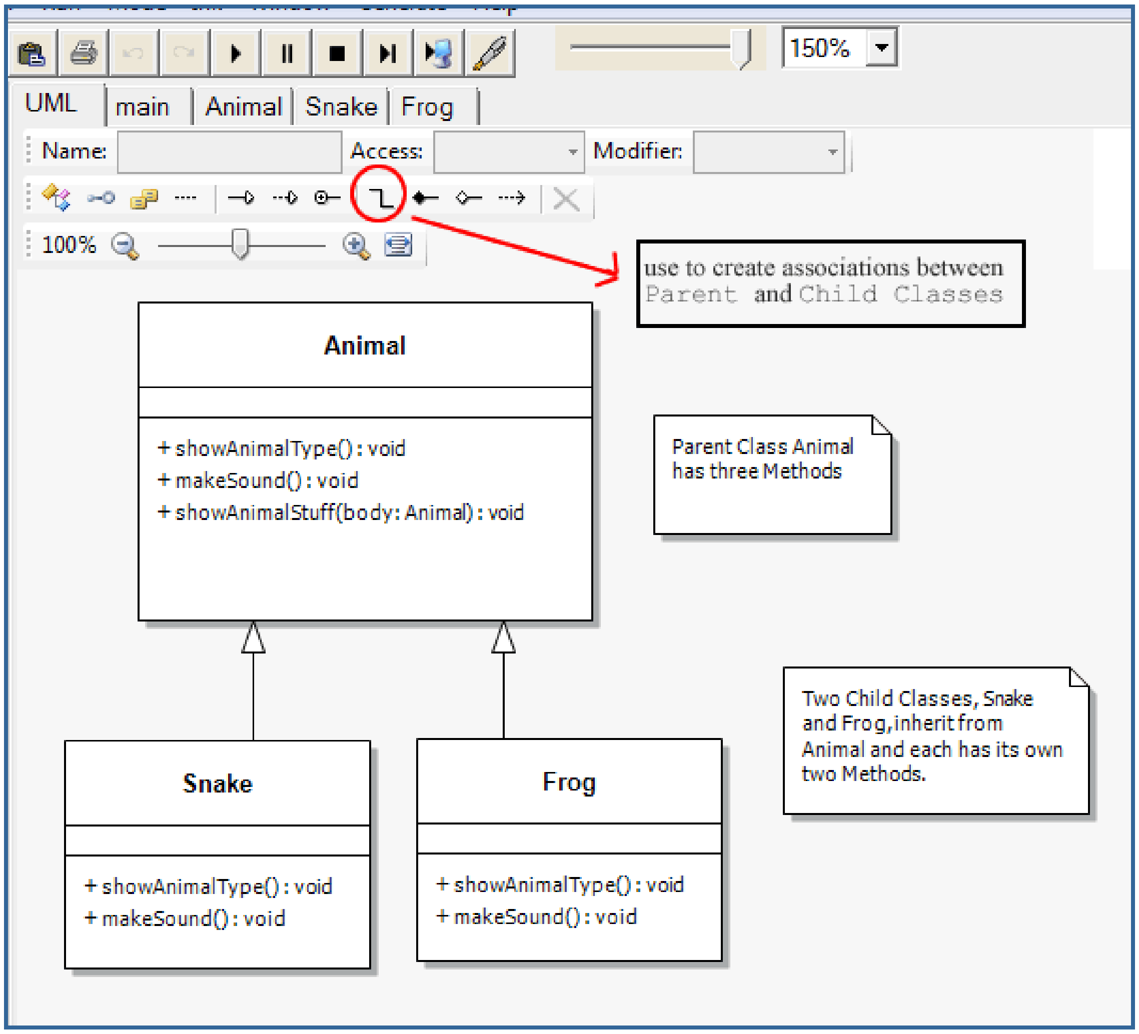

150%
UML
main
Animal
Snake
Frog
Name:
Access:
Modifier:
100%
use to create associations between Parent and Child Classes
Animal
+ showAnimalType() : void
+ makeSound() : void
+ showAnimalStuff(body : Animal) : void
Parent Class Animal has three Methods
Two Child Classes, Snake and Frog, inherit from Animal and each has its own two Methods.
Snake
+ showAnimalType() : void
+ makeSound() : void
Frog
+ showAnimalType() : void
+ makeSound() : void

Figure 16 Child Classes inherit from the Parent Class

By combining all the features of RAPTOR's OOP mode and all that you have learned in this text about object-oriented programming, it is possible to create some interesting and sophisticated programs.