Programming Fundamentals

Methods Continuation

Week 5| Iresh Bandara

ROBERT GORDON UNIVERSITY ABERDEEN
TEF Gold
INFORMATICS INSTITUTE OF TECHNOLOGY

Learning Outcomes

- On completion of this lecture, students are expected to describe and apply
  - Methods + usage
  - Overloading
  - Proper variable scope usage
  - Recursion

Recap : Methods

- The purpose of using methods is to break up a program into smaller, reusable pieces of software.
- While some methods are predefined - that is written and included as part of the Java environment, most methods will be written by the programmer.

Example

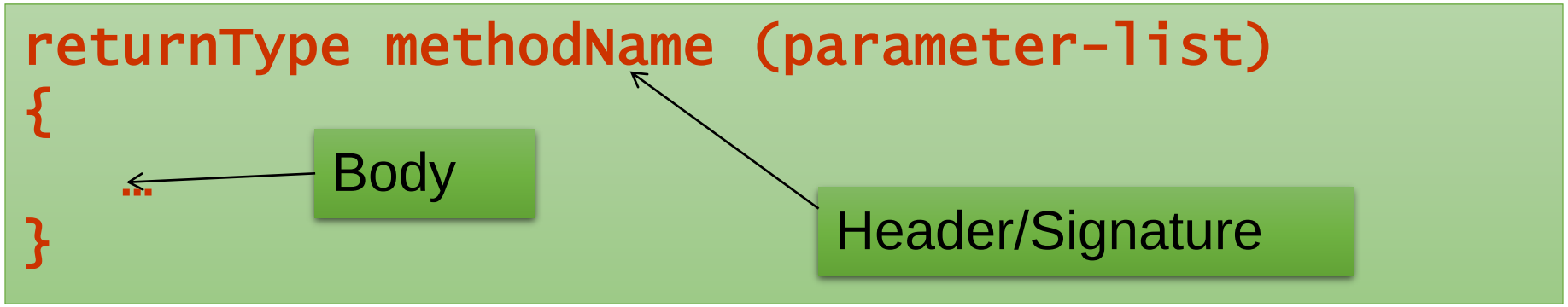
```
for(int i=0;i<10;i++)
{
    System.out.println(i);
}

// repeat for the 2nd time
for(int j=20;j<30;j++)
{
    System.out.println(j);
}

// repeat once afain
for(int k=40;k<50;k++)
{
    System.out.println(k);
}
```

```
public class Main {

    // method to avoid repetition
    public static void printNum(int start, int end)
    {
        for(int i=start;i<end;i++)
        {
            System.out.println(i);
        }
    }

    public static void main(String[] args)
    {
        printNum(0,10);
        printNum(20,30);
        printNum(40,50);
    }

}
```

Recap : How to write a Method?

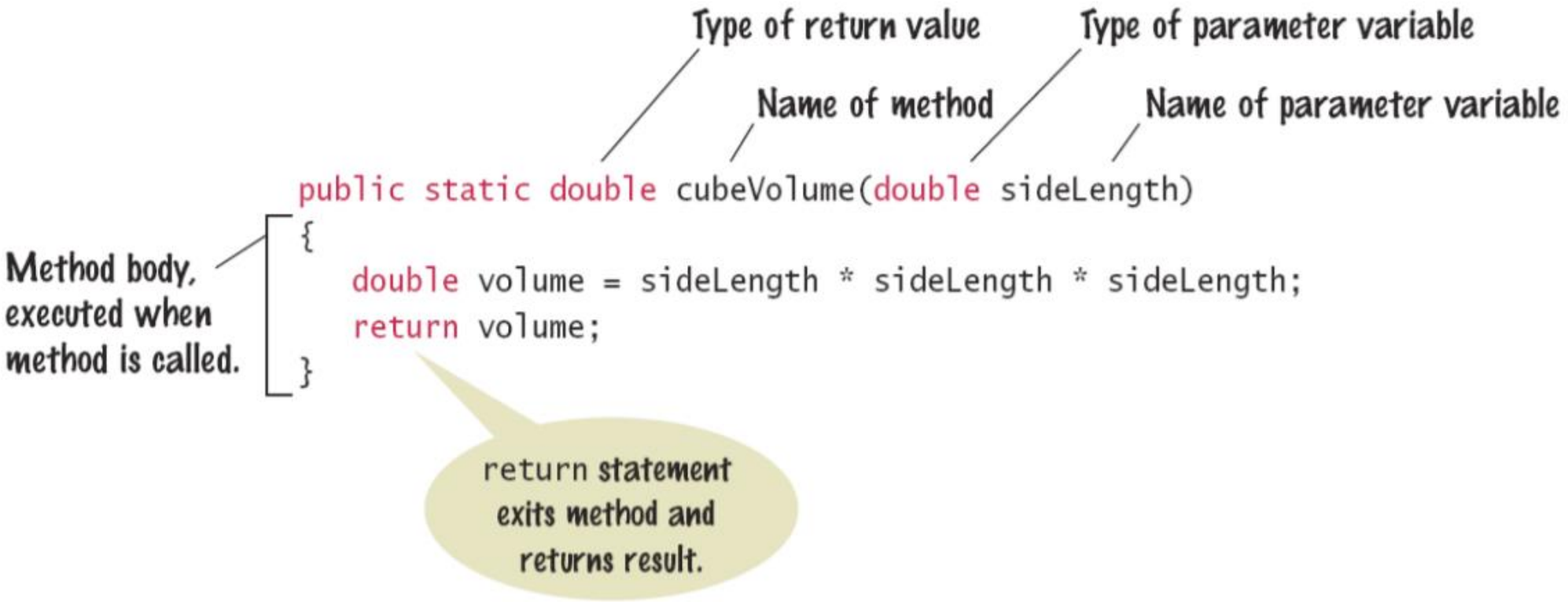
- We have so far used methods such as main() and will now look at how we can create methods of our own.
- To define a method:
  - give it a name
  - specify the method's return type or choose **void**
  - specify the types of parameters and give them names or keep the parenthesis empty.
  - write the method body
  - test the method

Recap : How to write a Method?



- A method is always defined inside a class.
- A method returns a value of the specified type unless it is declared void; the return type can be any primitive data type or a class type.
- A method's parameters can be of any primitive data types or class types.

Example


Type of return value
Type of parameter variable
Name of method
Name of parameter variable
public static double cubeVolume(double sideLength)
{
double volume = sideLength * sideLength * sideLength;
return volume;
}
Method body, executed when method is called.
return statement exits method and returns result.

Recap : Invoking a Method

- We invoke (or 'call') a method by stating:
  - Its name (identifier)
  - The values to be taken by its parameters

- Example:

```
cubeVolume(12.3,12.3);// calling multi-arg method
double result=cubeVolume(12.3);// calling a one argument method
cubeVolume(); // calling no argument method
```

Recap : Passing Parameters

- So the values that are supplied to the method as parameters can be:
  - *constant* values, such as `12.3`
  - *expressions*, such as `7.5+5.6`
  - *variables*, such as in `sideLength=12.3`
    - not a named parameter. Only initialize when passing

- Where an expression is used, it is evaluated first and then the result is copied to the method.
- Where a variable is used, its value is copied to the method and the variable remains unchanged -> pass by value.

Recap : Formal & Actual Parameters

- The formal parameters are:
  - The identifiers used when writing the method signature.
  - Their use is local to the method
- The actual parameters are:
  - the parameters in the method call (those being passed to the method).
- Actual parameters must match the formal parameters in number and type.

Recap : Returning Information

- The rules of Java only allow us to pass information into a method through the parameters.
- To get results out of a method, we turn it into an expression and return a value of a particular type.
- Storing a returned value after the call

```
double result = cubeVolume(12.3);
```

- The methods were of type void which means that they do not return any value.

Method Comments

- Whenever you write a method, you should comment its behavior
- Method comments explain:
  - The purpose of the method
  - The meaning of the parameter variables
  - The return value
  - Any special requirements

```
/**
   Computes the volume of a cube.
   @param sideLength the side length of the cube
   @return the volume
*/
public static double cubeVolume(double sideLength)
```

Variable Scope

- Variables can be declared
  - Inside a method
    - Known as "local variable"
    - Availability inside the method
    - Parameters are local variables
- Inside a block of code {}
  - If variable declared inside {}
- Outside method
  - Sometimes called Global scope
  - Use and change inside any method
- Instance/member variables
  - Declare inside a class

Example of Scopes

- sum is a local variable in main
- square is only visible inside the for loop block
- i is only visible inside the for loop

```
public static void main(String[] args)
{
  int sum = 0;
  for (int i = 1; i <= 10; i++)
  {
    int square = i * i;
    sum = sum + square;
  }
  System.out.println(sum);
}
```

sum

i

square

The scope of a variable is the part of the program in which it is visible.

Local Variables of Methods

- Variables declared inside one method are not visible to other methods
- sideLength is local to main
- Using it outside main will cause a compiler error

```
public static void main(String[] args)
{
  double sideLength = 10;
  int result = cubeVolume();
  System.out.println(result);
}

public static double cubeVolume()
{
  return sideLength * sideLength * sideLength; // ERROR
}
```

Reusing names for local variables

- Variables declared inside one method are not visible to other methods
  - result is local to square and result is local to main
  - They are two different variables and do not overlap

```
public static int square(int n)
{
  int result = n * n;
  return result;
}

public static void main(String[] args)
{
  int result = square(3) + square(4);
  System.out.println(result);
}
```

result

result

Re-using names for block variables

- Variables declared inside one block are not visible to other methods
  - i is inside the first for block and i is inside the second
  - They are two different variables and do not overlap

```
public static void main(String[] args)
{
  int sum = 0;
  for (int i = 1; i <= 10; i++)
  {
    sum = sum + i;
  }
  for (int i = 1; i <= 10; i++)
  {
    sum = sum + i * i;
  }
  System.out.println(sum);
}
```

i

i

Overlapping Scope

- Variables (including parameter variables) must have unique names within their scope
  - n has local scope and n is in a block inside that scope
  - The compiler will complain when the block scope n is declared

```
public static int sumOfSquares(int n)
{
  int sum = 0;
  for (int i = 1; i <= n; i++)
  {
    int n = i * i; // ERROR
    sum = sum + n;
  }
  return sum;
}
```

Local n

block scope n

Global and Local Overlapping

- Global and Local (method) variables can overlap
  - The local same will be used when it is in scope
  - No access to global same when local same is in scope

```
public class Scoper
{
  public static int same;    // 'global'
  public static void main(String[] args)
  {
    int same = 0;      // local
    for (int i = 1; i <= 10; i++)
    {
      int square = i * i;
      same = same + square;
    }
    System.out.println(same);
  }
}
```

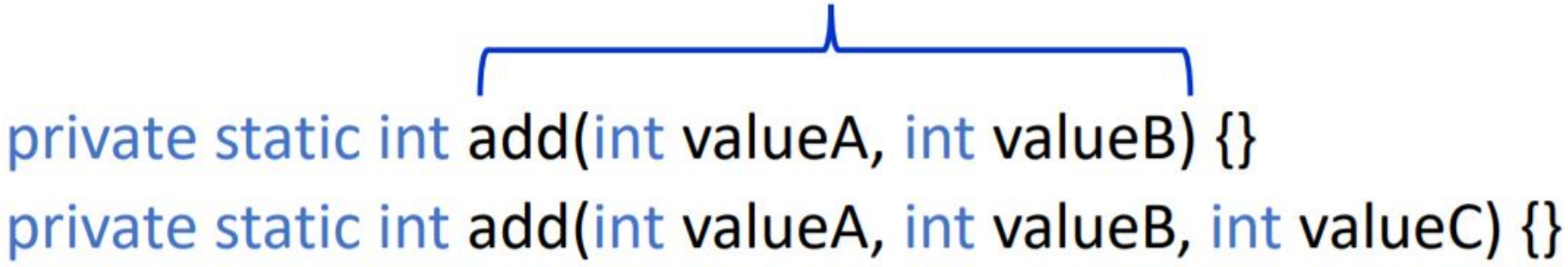
same

same

Variables in different scopes with the same name will compile, but it is not a good idea

Method Overloading

- Method signature: name, number, and type of parameters



- You can use the same name of the method for :
  - Different types of parameters
  - Different number of parameters

- Also called “static polymorphism”

Return Value and Signature

- The return value is not included in the signature

```
private static int add(int valueA, int valueB) {}
 private static int add(int valueA, int valueB, int valueC) {}  //OK

private static float add(int valueA, int valueB, int valueC) {}  //Not OK
```

- Third method is not possible to implement.

Method Overloading : Example

```
public class Main {

  1  public void test(double a,double b){}


   2 public void test(int a,int b){}

   3 public void test(int c, double d){}

  4  public void test(double e, int f){}

    public static void main(String[] args)
    {

        Main m=new Main();
        m.test(12.3,12.2);
        m.test(12,12);
        m.test(12,12.2);
        m.test(12.3,12);

    }

}
```

Recursive Methods

- A recursive method is a method that calls itself
- A recursive computation solves a problem by using the solution of the same problem with simpler inputs
- For a recursion to terminate, there must be special cases for the simplest inputs
- Every recursive call must simplify the task in some way.
- There must be special cases to handle the simplest tasks directly.

PowerOf Example

```
public class Main {

    static double myPower(double number, int powerOf)
    {
        if(powerOf ==0){ //special case
            return 1;
        }
        else
        {
            return  number * myPower(number,powerOf -1);
        }
    }

     public static void main(String[] args)
     {

         double result=myPower(2,3);
         System.out.println(result);

     }
}
```

Recursive Calls and Returning

- **Assume the developer calls the method** `myPower(2,4)`
  - `The call myPower(2,4) calls myPower(2,3)`
    - `The call myPower(2,3) calls myPower(2,2)`
      - `The call myPower(2,2) calls myPower(2,1)`
        - `The call myPower(2,1) calls myPower(2,0)`
          - `myPower(2,0) returns 1`
        - `myPower(2,1) returns 2*1`
      - `myPower(2,2) returns 2*2`
    - `Mypower(2,3) returns 2*4`
  - `myPower(2,4) finally returns 2*8`

Recursive Triangle Example

```
public static void printTriangle(int sideLength)
{
  if (sideLength < 1) { return; }

  printTriangle(sideLength - 1);
  for (int i = 0; i < sideLength; i++)
  {
    System.out.print("[]");
  }
  System.out.println();
}
```

Special Case

Recursive Call

```
[]
[][]
[][][]
[][][][]
```

Print the triangle with side length 3.
Print a line with four [].

- The method will call itself (and not output anything) until `sideLength` becomes < 1
- It will then use the return statement and each of the previous iterations will print their results
  - 1, 2, 3 then 4

Recursive Calls and Returns

- The call `printTriangle(4)` calls `printTriangle(3)`.
  - The call `printTriangle(3)` calls `printTriangle(2)`.
    - The call `printTriangle(2)` calls `printTriangle(1)`.
      - The call `printTriangle(1)` calls `printTriangle(0)`.
        - The call `printTriangle(0)` returns, doing nothing.
      - The call `printTriangle(1)` prints `[]`.
    - The call `printTriangle(2)` prints `[][]`.
  - The call `printTriangle(3)` prints `[][][]`.
- The call `printTriangle(4)` prints `[][][][]`.

Thank you

ROBERT GORDON UNIVERSITY ABERDEEN
TEF Gold
INFORMATICS INSTITUTE OF TECHNOLOGY