Methods: A deeper look

CS102A Lecture 6

James YU

Oct. 19, 2020

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Objectives

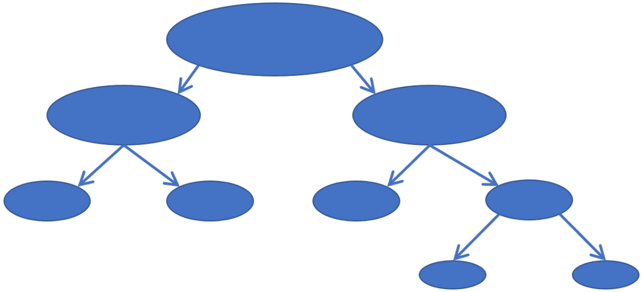
- Modular programming.
- How to use methods.
- Method-call stack (program execution stack).
- Method overloading.

Problem solving

- The programs we have written so far solve simple problems (find the max in an array of numbers).
- They are short and everything fits well in a `main` method.
- What if you are asked to *solve complex problems*, e.g., building a climate model from big data? A giant main method?

Divide and conquer 分治

- Decompose a big/complex task into smaller one and solve each of them.



MEthods

- Methods facilitate the design, implementation, operation and maintenance of large programs.

```java
import java.util.Random;
public class NumberGuessing {
  public static void main(String[] args) {
    Random random = new Random();
    int magicNum = random.nextInt(10);
  }
}
```

- `random.nextInt(10)`: calling a method to generate a random number.
- We don't need to know how random numbers are generated.

Why use methods?

- For reusable code, reducing code duplication:
    - If you need to do the same thing many times, write a method to do it, then call the method each time you have to do that task.
- To parameterize code:
    - You will often use parameters that change the way the method works.
- For top-down programming (divide and conquer):
    - You solve a big problem (the “top”) by breaking it down into small problems. To do this in a program, you write a method for solving your big problem by calling other methods to solve the smaller parts of the problem, which similarly call other methods until you get down to simple methods which solve simple problems.
- To create conceptual units:
    - Create methods to do something that is one action in your mental view of the problem. This will make it much easier for you to program.

Why use methods?

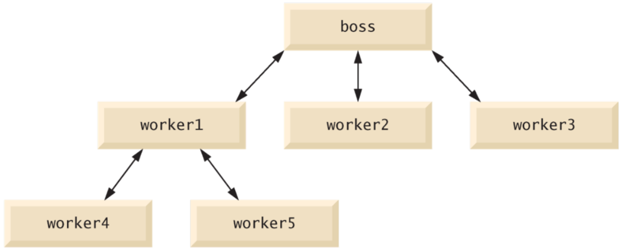
- To simplify:
  - Because local variables and statements of a method can not been seen from outside the method, they (and their complexity) are hidden from other parts of the program, which prevents accidental errors or confusion (e.g., random number generation method).
- To ease debugging and maintenance:
  - You don’t want to debug a `main` method with 100000 lines of code.

Program modules in Java

- Java programs are written by combining *new methods and classes* that you write with *predefined methods and classes* available in the Java Application Programming Interface and in various other libraries.
- Related classes are typically grouped into packages so that they can be imported into programs and reused.
- The Java API provides a rich collection of predefined classes (e.g., `java.util.Scanner`, `java.lang.Math`).

Program modules in Java

- Similar to the hierarchical form of management.
  - A boss (the caller) asks a worker (the callee) to perform a task and report back (return) the results after completing the task.
  - The boss method does not know how the worker method performs its designated tasks (method complexity is hidden).
  - The worker may also call other worker methods, unknown to the boss.

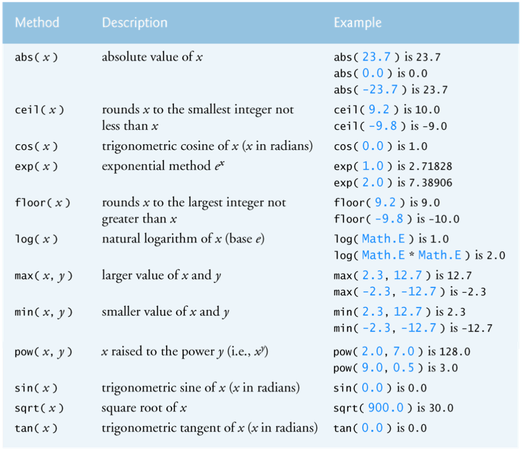


`static` methods

- Sometimes a method performs a task that does not depend on the contents of any object.
  - Known as a static method or a *class method*.
  - Place the keyword `static` before the return type in the declaration.
  - Called via the class name and a dot (`.`) separator.

`static` methods

- Many more useful static methods in `java.lang.Math` class:



Method	Description	Example
`abs( x )`	absolute value of *x*	`abs( 23.7 )` is 23.7 `abs( 0.0 )` is 0.0 `abs( -23.7 )` is 23.7
`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 )` is 10.0 `ceil( -9.8 )` is -9.0
`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is 1.0
`exp( x )`	exponential method e^x	`exp( 1.0 )` is 2.71828 `exp( 2.0 )` is 7.38906
`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is 9.0 `floor( -9.8 )` is -10.0
`log( x )`	natural logarithm of *x* (base *e*)	`log( Math.E )` is 1.0 `log( Math.E * Math.E )` is 2.0
`max( x, y )`	larger value of *x* and *y*	`max( 2.3, 12.7 )` is 12.7 `max( -2.3, -12.7 )` is -2.3
`min( x, y )`	smaller value of *x* and *y*	`min( 2.3, 12.7 )` is 2.3 `min( -2.3, -12.7 )` is -12.7
`pow( x, y )`	*x* raised to the power *y* (i.e., x^y)	`pow( 2.0, 7.0 )` is 128.0 `pow( 9.0, 0.5 )` is 3.0
`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is 0.0
`sqrt( x )`	square root of *x*	`sqrt( 900.0 )` is 30.0
`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is 0.0

`static` fields and class `Math`

- Class Math declares commonly used mathematical constants:
  - `Math.PI` (3.141592653589793).
  - `Math.E` (2.718281828459045) is the base value for natural logarithms.
- These fields are declared in class `Math` with the modifiers `public`, `final` and `static`.
  - `public` allows you to use these fields in your own classes.
  - `final` indicates a constant — value cannot change.
  - `static` makes them accessible via the class name `Math` and a dot (`.`) separator.
  - `static` fields are also known as class variables (in contrast to instance variables).

Why `main` method has to be `static`?

- When you execute the Java Virtual Machine (JVM) with the `java` command, the JVM attempts to invoke the `main` method of the class you specify.
- Declaring `main` as `static` allows the JVM to invoke `main` without creating an object of the class.

Declaring methods

```java
import java.util.Scanner;
public class MaximumFinder {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("enter three floating-point values: ");
    double number1 = input.nextDouble();
    double number2 = input.nextDouble();
    double number3 = input.nextDouble();
    double result = maximum(number1, number2, number3);
    System.out.println("max is " + result);
  }
  public static double maximum(double x, double y, double z) {
    double max = x;
    if(y > max) max = y;
    if(z > max) max = z;
    return max;
  }
}
```

Details of methods

```
public static double maximum(double x, double y, double z) {
  double max = x;
  if(y > max) max = y;
  if(z > max) max = z;
  return max;
}
```

- Find the largest of 3 double values.

```
  double result = maximum(number1, number2, number3);
```

- You need to call it explicitly to tell it to perform its task.
- Method don’t get called automatically after declaration.
- `static` methods in the same class can call each directly.

Details of methods

```java
public static void main(String[] args) {
  Scanner input = new Scanner(System.in);
  System.out.print("enter three floating-point values: ");
  double number1 = input.nextDouble();
  double number2 = input.nextDouble();
  double number3 = input.nextDouble();
  double result = maximum(number1, number2, number3);
  System.out.println("max is " + result);
  double resultCeil = java.lang.Math.ceil(result);
}
```

- Using `static` methods defined in other classes requires a fully qualified method name (including the class name).

Details of methods

```
    double result = maximum(number1, number2, number3);
```

- A method call supplies arguments for each of the method’s parameters.
  - One to one correspondence and the type must be consistent.
- Before any method can be called, its arguments must be evaluated to determine their values.
- If an argument is a method call, the method call must be performed to determine its return value.

```
  Math.pow( Math.pow(x2 - x1, 2) + Math.pow(y2 - y1, 2), 0.5 );
```

Details of methods

```java
public static double maximum(double x, double y, double z) {
  double max = x;
  if(y > max) max = y;
  if(z > max) max = z;
  return max;
}
```

- *Return type*: the type of data the method returns to its caller. `void` means returning nothing.
- The *method name* follows the return type. Naming convention: `lowerCamelCase`.
- A comma-separated *list of parameters* mean that the method requires additional information from the caller to perform its task.
  - Each parameter must specify a *type* and an *identifier*.
  - A method's parameters are considered to be *local variables* of that method and can be used only in that method's body.

Details of methods

```java
public static double maximum(double x, double y, double z) {
  double max = x;
  if(y > max) max = y;
  if(z > max) max = z;
  return max;
}
```

- *Method header* = *modifiers* + *return type* + *method name* + *parameters*.
- *Method body* contains one or more statements that perform the method's task.
- The return statement returns a value (or just control) to the point in the program from which the method is called.
  - It is good to have `return;` for a method with a return type *void*. This means that the method terminates without returning data.

Returning results

- If the method does not return a result, control returns when the program flow reaches the method-ending right brace.
- Or when the statement `return;` executes.
- If the method returns a result, the statement
  - `return expression;` evaluates the expression, then returns the result to the caller.

Stack

- Stack data structure: analogous to a pile of dishes.
    - When a dish is placed on the pile, it's normally placed at the top (referred to as pushing onto the stack).
    - Similarly, when a dish is removed from the pile, it's always removed from the top (referred to as popping off the stack).
- *Last-in, first-out (LIFO)* — the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.

Method-call stack

- When a program calls a method, the called method must know how to return to its caller, so the return address of the calling method is pushed onto the method-call stack (also known as program execution stack).
- If a series of method calls occurs, the successive return addresses are pushed onto the stack in last-in, first-out order.
- The program-execution stack also contains the memory for the local variables used in each invocation of a method.
    - Stored in the activation record (or stack frame) of the method call.
    - When a method call is made, the activation record for that method call is pushed onto the method-call stack.
- When a method returns to its caller, the activation record for the method call is popped off the stack and those local variables are no longer known to the program.

Queue

- Queue data structure: analogous to a box of badminton.
    - The badminton is usually placed on one side and taken out on the other side.
- *First-in, first-out (FIFO)* — the first item pushed (inserted) to the queue is the first item popped (removed) from.

Passing arguments in method calls

- Typically two ways: `pass-by-value` and `pass-by-reference`.
- When an argument is passed by value, a copy of the argument's value is passed to the called method.
    - The called method works exclusively with the copy.
    - Changes to the copy do not affect the original variable's value in the caller.
- When an argument is passed by reference, the called method can directly access the argument's value in the caller and modify that data, if necessary.
- Improves performance by avoiding copying possibly large amounts of data.

Pass-by-value in Java

- In Java, all arguments are passed by value.
- A method call can pass two types of values to the called method: copies of primitive values and copies of references to objects.
- Although an object's reference is passed by value, a method can still interact with the referenced object using the copy of the object's reference (arrays are also objects).
    - The parameter in the called method and the argument in the calling method refer to the same object in memory.

Example

```java
public static void main(String[] args) {
  int a = 3;
  System.out.println("Before: " + a);
  triple(a);
  System.out.println("After: " + a);
}

public static void triple(int x) {
  x *= 3;
}
```

Example

```java
public static void main(String[] args) {
  int[] a = {1, 2, 3};
  System.out.print("Before: ");
  for(int value : a) System.out.printf("%d ", value);
  triple(a);
  System.out.print("\nAfter: ");
  for(int value : a) System.out.printf("%d ", value);
}

public static void triple(int[] x) {
  for(int i = 0; i < x.length; i++)
    x[i] *= 3;
}
```

Passing arrays to methods

- To pass an array argument to a method, specify the name of the array without any brackets.

```java
int[] numbers = {1, 2, 3};
modifyArray(numbers);
```

- When we pass an array object’s reference into a method, we need not pass the array length as an additional argument because every array knows its own length.
- For a method to receive an array reference through a method call, the method’s parameter list must specify an array parameter.

Passing arrays to methods

- When a method argument is an entire array or an array element of a reference type, the called method receives a copy of the reference.

```java
int[] numbers = {1, 2, 3};
modifyArray(numbers);
```

- When a method argument is a primitive-type array element, the called method receives a copy of the element's value. Such primitive values are called scalars or scalar quantities.

```java
int[] numbers = {1, 2, 3};
modifyElement(numbers[1]); // receive a scalar of value 2
```

Using command-line arguments

- It's possible to pass arguments from the command line to an application by including a parameter of type `String[]` in the parameter list of main.

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

- By convention, this parameter is named `args`.
- When an application is executed using the `java` command, Java passes the command-line arguments that appear after the class name in the java command to the application's `main` method as `String`s in the array `args`.

Variable-length argument lists

- With *variable-length argument lists*, you can create methods that receive an unspecified number of arguments.
- A type followed by an ellipsis (`...`) in a method's parameter list indicates that the method receives a variable number of arguments of that particular type.

```java
public static double average(double... numbers)
```

  - Can occur only once in a parameter list, and the ellipsis, together with its type, must be placed *at the end of the parameter list*.
- Java treats the variable-length argument list as an array of the specified type.

Example

```java
public static double average(double... numbers) {
  double total = 0.0;
  for(double d : numbers) total += d;
  return total / numbers.length;
}

public static void main(String[] args) {
  double d1 = 10.0, d2 = 20.0, d3 = 30.0;
  System.out.printf("average of d1 and d2: %f\n", average(d1, d2));
  System.out.printf("average of d1 ~ d3: %f\n", average(d1, d2, d3));
}
```

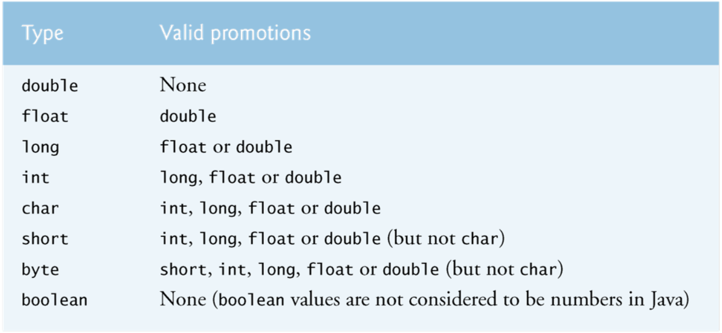
```
average of d1 and d2: 15.000000
average of d1 ~ d3: 20.000000
```

Argument promotion

- *Argument promotion* — converting an argument's value, if possible, to the type that the method expects to receive in its corresponding parameter.
  - `Math.sqrt()` expects to receives a `double` argument, but it is ok to write `Math.sqrt(4)`: Java converts the `int` value `4` to the double value `4.0`.

Promotion rules

- Specify which conversions are allowed (which conversions can be performed without losing data):



Type	Valid promotions
`double`	None
`float`	`double`
`long`	`float` or `double`
`int`	`long`, `float` or `double`
`char`	`int`, `long`, `float` or `double`
`short`	`int`, `long`, `float` or `double` (but not `char`)
`byte`	`short`, `int`, `long`, `float` or `double` (but not `char`)
`boolean`	None (`boolean` values are not considered to be numbers in Java)

Promotion rules

- Besides arguments passed to methods, the rules also apply to expressions containing values of two or more primitive types.
    - `2 * 2.0` becomes `4.0`.

```java
int x = 2;
double y = x * 2.0;
// is x 2.0 or 2 now?
```

Case study: Random number generation

- The element of chance can be introduced in a program via an object of class `Random` (package `java.util`) or via the `static` method `random` of class `Math`.
- Objects of class `Random` can produce random `boolean`, `byte`, `float`, `double`, `int`, `long` and Gaussian values.
- `Math` method `random` can produce only `double` values in the range $0.0 \ < x \ < 1.0$, where x is the value returned by method `random`
- `Random` class documentation: urlhttp://java.sun.com/javase/6/docs/api/java/util/Random.html
- `Random` method `nextInt` generates a random `int` value in the range –2147483648 to $+2147483647$, inclusive.

Case study: Random number generation

- If truly random, then every value in the range should have an equal chance (or probability) of being chosen each time `nextInt` is called.
- The numbers are actually *pseudorandom numbers* — a sequence of values produced by a complex mathematical calculation.
- The calculation uses the current time of day to seed the random-number generator such that each execution of a program **yields a different sequence of random values**.
- Class `Random` provides another version of method `nextInt` that receives an `int` argument and returns a value from `0` up to the integer, but not including it.

Case study: Random number generation

```java
// Shifted and scaled random integers.
import java.security.SecureRandom; // program uses class SecureRandom
public class RandomIntegers {
  public static void main(String[] args) {
    // randomNumbers object will produce secure random numbers
    SecureRandom randomNumbers = new SecureRandom();
    // loop 20 times
    for (int counter = 1; counter <= 20; counter++) {
      // pick random integer from 1 to 6
      int face = 1 + randomNumbers.nextInt(6);
      System.out.printf("%d ", face); // display generated value
      // if counter is divisible by 5, start a new line of output
      if (counter % 5 == 0)
      System.out.println();
    }
  }
} // end class RandomIntegers
```

Case study: Random number generation

```
1 5 3 6 2
5 2 6 5 2
4 4 4 2 6
3 1 6 2 2
```

```
6 5 4 2 6
1 2 5 1 3
6 3 2 2 1
6 4 2 6 4
```

Case study: Rolling a six-sided die

```java
// Roll a six-sided die 6,000,000 times.
import java.security.SecureRandom;
public class RollDie {
  public static void main(String[] args) {
    // randomNumbers object will produce secure random numbers
    SecureRandom randomNumbers = new SecureRandom();

    int frequency1 = 0; // count of 1s rolled
    int frequency2 = 0; // count of 2s rolled
    int frequency3 = 0; // count of 3s rolled
    int frequency4 = 0; // count of 4s rolled
    int frequency5 = 0; // count of 5s rolled
    int frequency6 = 0; // count of 6s rolled

    // tally counts for 6,000,000 rolls of a die
    for (int roll = 1; roll <= 6000000; roll++) {
      int face = 1 + randomNumbers.nextInt(6); // number from 1 to 6
```

Case study: Rolling a six-sided die

```java
   // use face value 1-6 to determine which counter to increment
   switch ( face ) {
    case 1: ++frequency1; break; // increment the 1s counter
    case 2: ++frequency2; break; // increment the 2s counter
    case 3: ++frequency3; break; // increment the 3s counter
    case 4: ++frequency4; break; // increment the 4s counter
    case 5: ++frequency5; break; // increment the 5s counter
    case 6: ++frequency6; break; // increment the 6s counter
   }
  }
  System.out.println("Face\tFrequency"); // output headers
  System.out.printf("1\t%d%n2\t%d%n3\t%d%n4\t%d%n5\t%d%n6\t%d%n",
  frequency1, frequency2, frequency3, frequency4,
  frequency5, frequency6);
 }
} // end class RollDie
```

Case study: Rolling a six-sided die

```
Face Frequency
1    999501
2    1000412
3    998262
4    1000820
5    1002245
6    998760
```

```
Face Frequency
1    999647
2    999557
3    999571
4    1000376
5    1000701
6    1000148
```

Random-number repeatability for testing

- The calculation that produces random numbers uses the time of day as a seed value to change the sequence's starting point.
- Each new `Random` object seeds itself with a value based on the computer system's clock at the time the object is created.
- Sometimes useful to repeat the exact same sequence of pseudorandom numbers during each execution.
  - Enables you to prove that your application is working for a specific sequence of random numbers before you test the program with different sequences of random numbers.
- When repeatability is important, you can create a `Random` object with a seed value as an argument to the constructor.

Scope of declarations

- Declarations introduce names that can be used to refer to classes, methods, variables and parameters.
- *The scope of a declaration* is the portion of the program that can refer to the declared entity by its name.
- Such an entity is said to be “in scope” for that portion of the program.
- Scope information — Java Language Specification, Section 6.3: Scope of a Declaration `http://java.sun.com/docs/books/jls/third_edition/html/names.html#103228`

Scope of declarations

- Basic scope rules:
  - The scope of a parameter declaration is **the body of the method** in which the declaration appears.
  - The scope of a local-variable declaration is **from the point at which the declaration appears to the end of that block**.
  - The scope of a local-variable declaration that appears in the initialization section of **a for statement's header is the body of the for statement and the other expressions in the header**.
  - A method or field's scope is the entire body of the class.

Scope of declarations

- Any block may contain variable declarations.
- If a local variable or parameter in a method has the same name as a field of the class, the field is “hidden” until the block terminates execution: *shadowing*.
- The application in the next slides demonstrates scoping issues with fields and local variables.

Scope of declarations

```java
// Scope class demonstrates field and local variable scopes.
public class Scope {
  // field that is accessible to all methods of this class
  private static int x = 1;

  // create and initialize local variable x during each call
  public static void useLocalVariable() {
    int x = 25; // initialized each time useLocalVariable is called

    System.out.printf("%nlocal x on entering method useLocalVariable is %
        d%n", x);
    ++x; // modifies this method's local variable x
    System.out.printf("local x before exiting method useLocalVariable is
        %d%n", x);
  }
```

Scope of declarations

```java
   // modify class Scope's field x during each call
   public static void useField() {
      System.out.printf("%nfield x on entering method useField is %d%n", x);
      x *= 10; // modifies class Scope's field x
      System.out.printf("field x before exiting method useField is %d%n", x);
   }

   // method main creates and initializes local variable x
   // and calls methods useLocalVariable and useField
   public static void main(String[] args) {
      int x = 5; // method's local variable x shadows field x

      System.out.printf("local x in main is %d%n", x);
```

Scope of declarations

```
      useLocalVariable(); // useLocalVariable has local x
      useField(); // useField uses class Scope's field x
      useLocalVariable(); // useLocalVariable reinitializes local x
      useField(); // class Scope's field x retains its value

      System.out.printf("%nlocal x in main is %d%n", x);
   }
} // end class Scope
```

Scope of declaration

```
local x in main is 5

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 1
field x before exiting method useField is 10

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 10
field x before exiting method useField is 100

local x in main is 5
```

Method overloading

- Methods of the same name can be declared in the same class, as long as they have different sets of parameters.
- Used to create several methods that perform *the same/similar tasks* on *different types or different numbers* of arguments.
    - Java compiler selects the appropriate method to call by examining the number, types and order of the arguments in the call.
- Compiler distinguishes overloaded methods by their *signature*.
    - A combination of the method's *name* and the *number*, *types* and *order of its parameters*.

```java
public double calculateAnswer(double wingSpan, int numberOfEngines,
        double length, double grossTons) {
  //do the calculation here
  return 0.0;
}
```

- *Signature*: `calculateAnswer(double, int, double, double)`.

Method overloading

- Method calls cannot be distinguished by return type. If you have overloaded methods only with different return types:
  - `int square(int a)`
  - `double square(int a)`

  and you called the method by `square(2);`, the compiler will be confused (since return value ignored).

Method overloading

```java
// Overloaded method declarations.
public class MethodOverload {
  // square method with int argument
  public static int square(int intValue) {
    System.out.printf("%nCalled square with int argument: %d%n", intValue
        );
    return intValue * intValue;
  }

  // square method with double argument
  public static double square(double doubleValue) {
    System.out.printf("%nCalled square with double argument: %f%n",
         doubleValue);
    return doubleValue * doubleValue;
  }
```

Method overloading

```java
  // test overloaded square methods
  public static void main(String[] args) {
    System.out.printf("Square of integer 7 is %d%n", square(7));
    System.out.printf("Square of double 7.5 is %f%n", square(7.5));
  }
} // end class MethodOverload
```

```
Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```