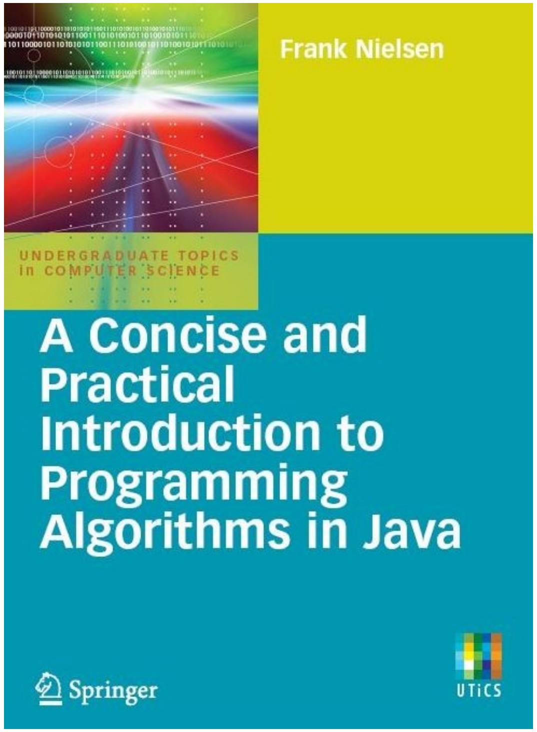
A Concise and Practical Introduction to Programming Algorithms in Java



Frank Nielsen

UNDERGRADUATE TOPICS in COMPUTER SCIENCE

A Concise and Practical Introduction to Programming Algorithms in Java

Springer

UTiCS

Chapter 3: Functions and recursivity

Frank NIELSEN

nielsen@lix.polytechnique.fr

So far... Executive review

Lecture 1: Java=**Typed** **compiled** programming language

Variables: `Type var;` (boolean, int, long, float, double)

Assignment: `var=Expression;` (with type checking)

Expression: `Operand1 Operator Operand2` (+-*/%)

Instruction (;) & *comments // or /* */*

So far... Executive review

Lecture 2: Program **workflow** (blocks/branching/loops)
Determine the set of instructions at runtime

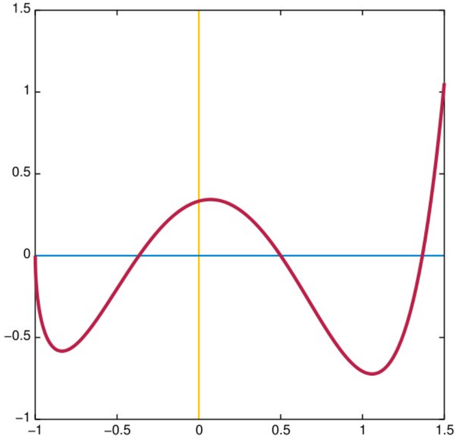
Blocks: sequence of instructions { }

Branching condition: `if predicate B1 else B2`
`(switch case break)`

Loops: `while, do, for` and escaping `break`

Numerical precisions: finite-precision arithmetic
(absurd results, loose of associativity, etc.)

Meaning of a function *in mathematics* ?



- Source (X) and target (Y) domains

- A map that associates to elements of X elements of Y

- An element of X is associated at most once to a member of Y

$$f : [-1,1.5] \rightarrow [-1,1.5]$$

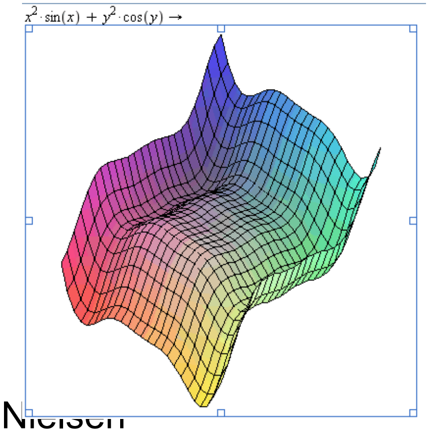
$$x \longmapsto \frac{(4x^3 - 6x^2 + 1)\sqrt{x+1}}{3-x}$$

- The mapping gives always the same result (deterministic/no randomness)

- Functions of several variables may be built blockwise...

  ...using Cartesian product of spaces

$$X_1 \times \cdots \times X_n = \{(x_1, \ldots, x_n) \mid x_1 \in X_1 \text{ and } \cdots \text{ and } x_n \in X_n\}.$$



Meaning of functions for *computing* ?

- A ***portion of a program*** processing data and returning a **result**

- A **function** not returning a result is also called a **procedure**

- A function has **typed parameters** as **arguments**

- A function usually yields the **same result** for a given set of arguments (except for side-effects or use of pseudo-randomness)

- A function needs to be **declared** first before calling it elsewhere

```
TypeF F(Type1 arg1, Type2 arg2, ..., TypeN argN)
{
TypeF result;
block of instructions;
return result;
}
```

Declaring functions in Java

```
class INF311{

public static  typeF F(type1 arg1, ..., typeN argN)
            {
            // Description
            Block of instructions;
            }
...
        }
```

- This kind of function is also called a **static method**
- Functions must be defined **inside classes**
- A function not returning a result has type **void** (also known as a procedure)

Defining the body of a function in Java

```
Class INF311{

public static  typeF F(type1 arg1, ..., typeN argN)
                              {
                              // Description
                              Block of instructions;
                              }
                      }
```

Body of a function

- Body should contain an instruction return to indicate the result

- If branching structures are used (if or switch) , a return should be written for all different branches. Otherwise we get acompiler error!

Defining the body of a function in Java

```
class INF311{

public static  typeF F(type1 arg1, ..., typeN argN)
           {
           // Description
           Block of instructions;
           }
       }
```

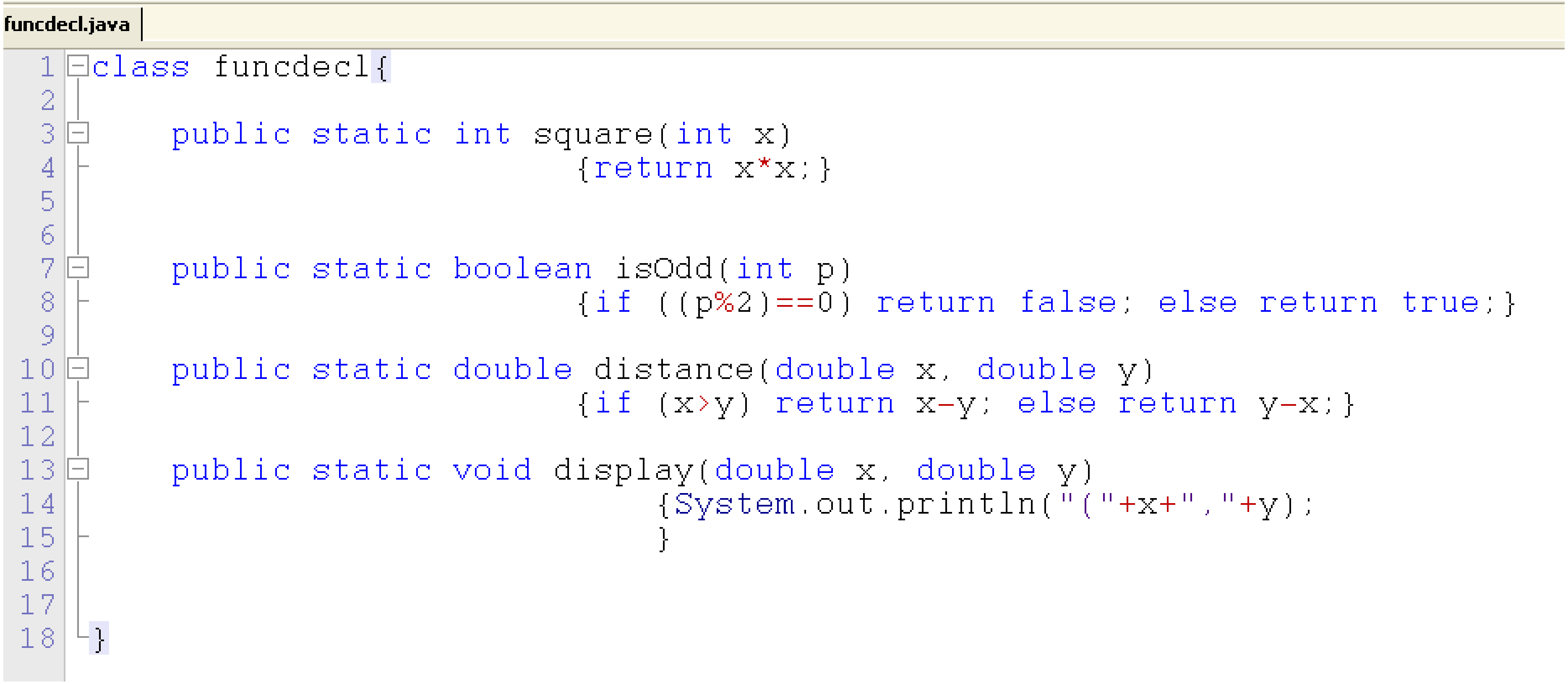
Body of a function

Body should contain an instruction `return` to indicate the result

If branching structures are used (if or switch) ,
then a `return` should be written for all different branches.

... Otherwise we get a compiler error! *(why? => not type safe!)*

Using functions in Java



```
class funcdecl{

    public static int square(int x)
                        {return x*x;}


    public static boolean isOdd(int p)
                        {if ((p%2)==0) return false; else return true;}

    public static double distance(double x, double y)
                        {if (x>y) return x-y; else return y-x;}

    public static void display(double x, double y)
                            {System.out.println("("+x+","+y);
                            }



}
```

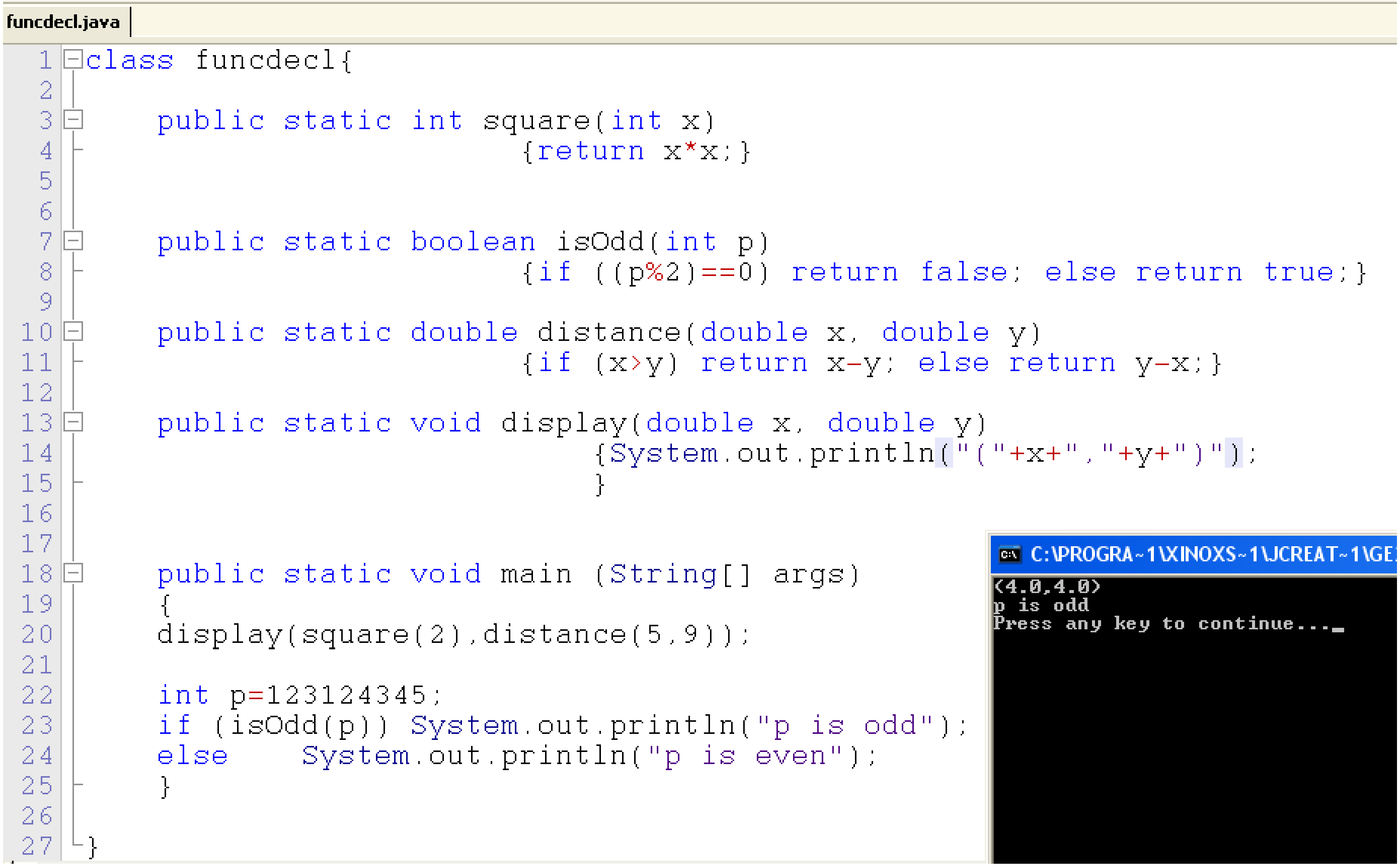
A few examples of basic functions

```
class FuncDecl{

   public static int square(int x)
                    {return x*x;}


   public static boolean isOdd(int p)
                    {if ((p%2)==0) return false;
                                     else return true;}

   public static double distance(double x, double y)
                    {if (x>y) return x-y;
                           else return y-x;}

   public static void display(double x, double y)
                       {System.out.println("("+x+","+y+")");
                        return; // return void
                       }

   public static void main (String[] args)
   {
   ...
   }
}
```

A few examples of basic functions

```
class FuncDecl{
   public static int square(int x){...}


   public static boolean isOdd(int p) {...}


   public static double distance(double x, double y) {...}


   public static void display(double x, double y) {...}


   public static void main (String[] args)
   {
   display(3,2);
   display(square(2),distance(5,9));

   int p=123124345;
   if (isOdd(p))
          System.out.println("p is odd");
             else   System.out.println("p is even");
   }
}
```

Functions... JCreator IDE



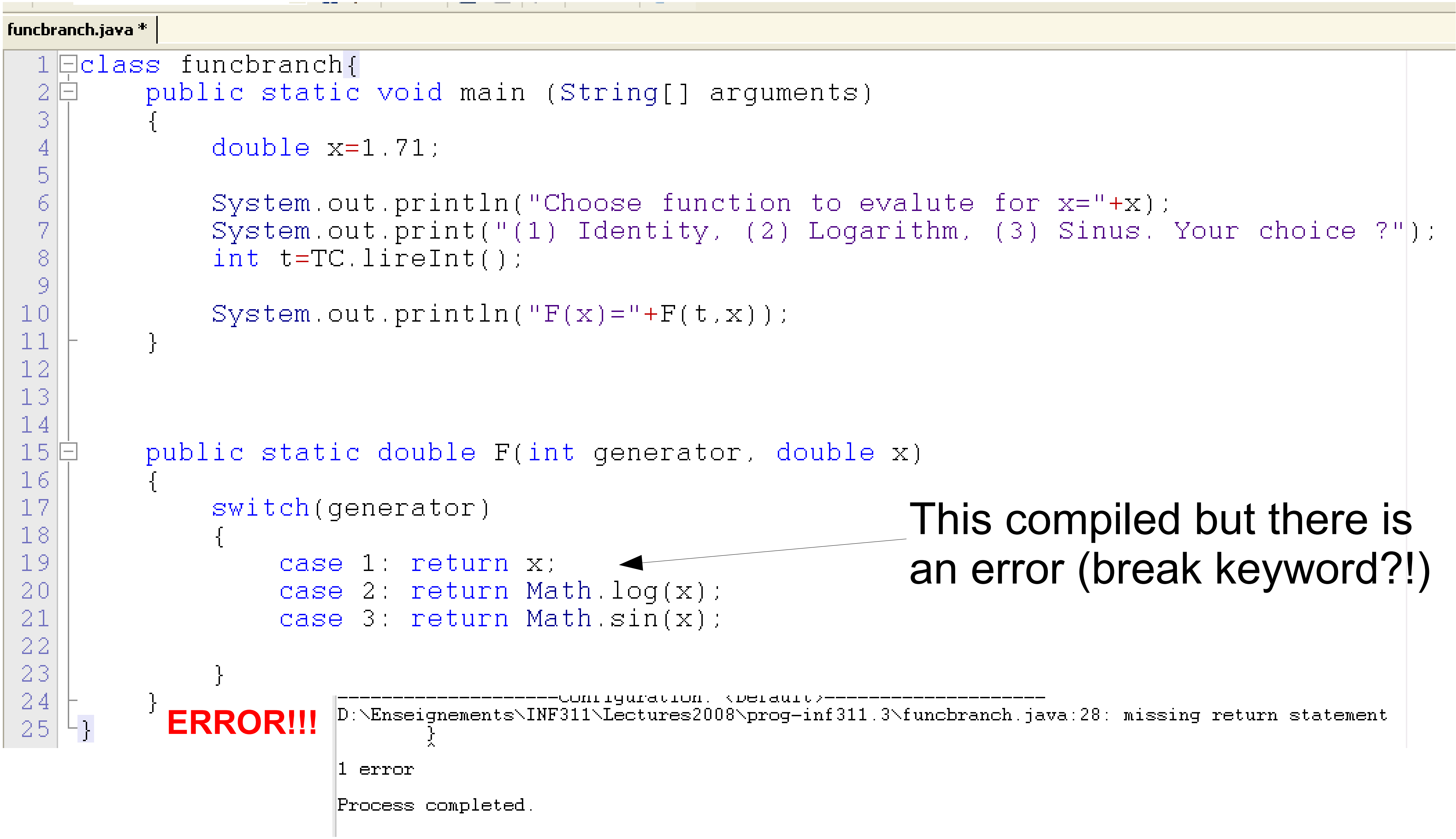
```java
class funcdecl{

    public static int square(int x)
                        {return x*x;}


    public static boolean isOdd(int p)
                        {if ((p%2)==0) return false; else return true;}

    public static double distance(double x, double y)
                        {if (x>y) return x-y; else return y-x;}

    public static void display(double x, double y)
                            {System.out.println("("+x+","+y+")");
                            }


    public static void main (String[] args)
    {
    display(square(2),distance(5,9));

    int p=123124345;
    if (isOdd(p)) System.out.println("p is odd");
    else    System.out.println("p is even");
    }

}
```

```
(4.0,4.0)
p is odd
Press any key to continue..._
```

Benefits of using functions

- **Modularity** (ease of presentation)
- **Code re-use** (program once, re-use many times!)
  -> library (API)
- Ease certification of correctness and test routines.

Functions with branching structures



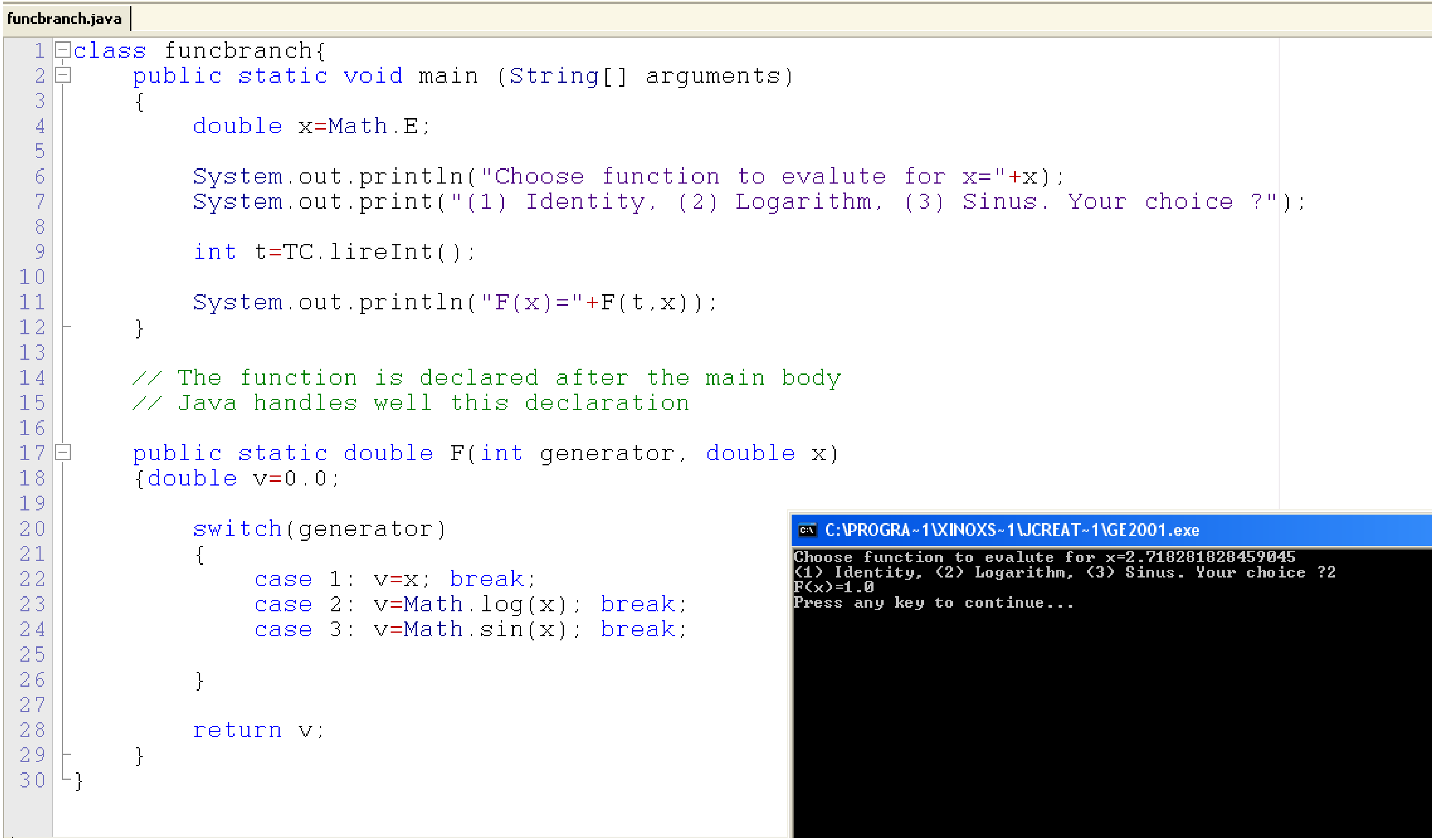
```java
class funcbranch{
    public static void main (String[] arguments)
    {
        double x=1.71;

        System.out.println("Choose function to evalute for x="+x);
        System.out.print("(1) Identity, (2) Logarithm, (3) Sinus. Your choice ?");
        int t=TC.lireInt();

        System.out.println("F(x)="+F(t,x));
    }



    public static double F(int generator, double x)
    {
        switch(generator)
        {
            case 1: return x;
            case 2: return Math.log(x);
            case 3: return Math.sin(x);

        }
    }
}
```

This compiled but there is an error (break keyword?!)

ERROR!!!

```
--------------------Configuration: <Default>--------------------
D:\Enseignements\INF311\Lectures2008\prog-inf311.3\funcbranch.java:28: missing return statement
        }
        ^

1 error

Process completed.
```

Functions with branching structures (correct program)

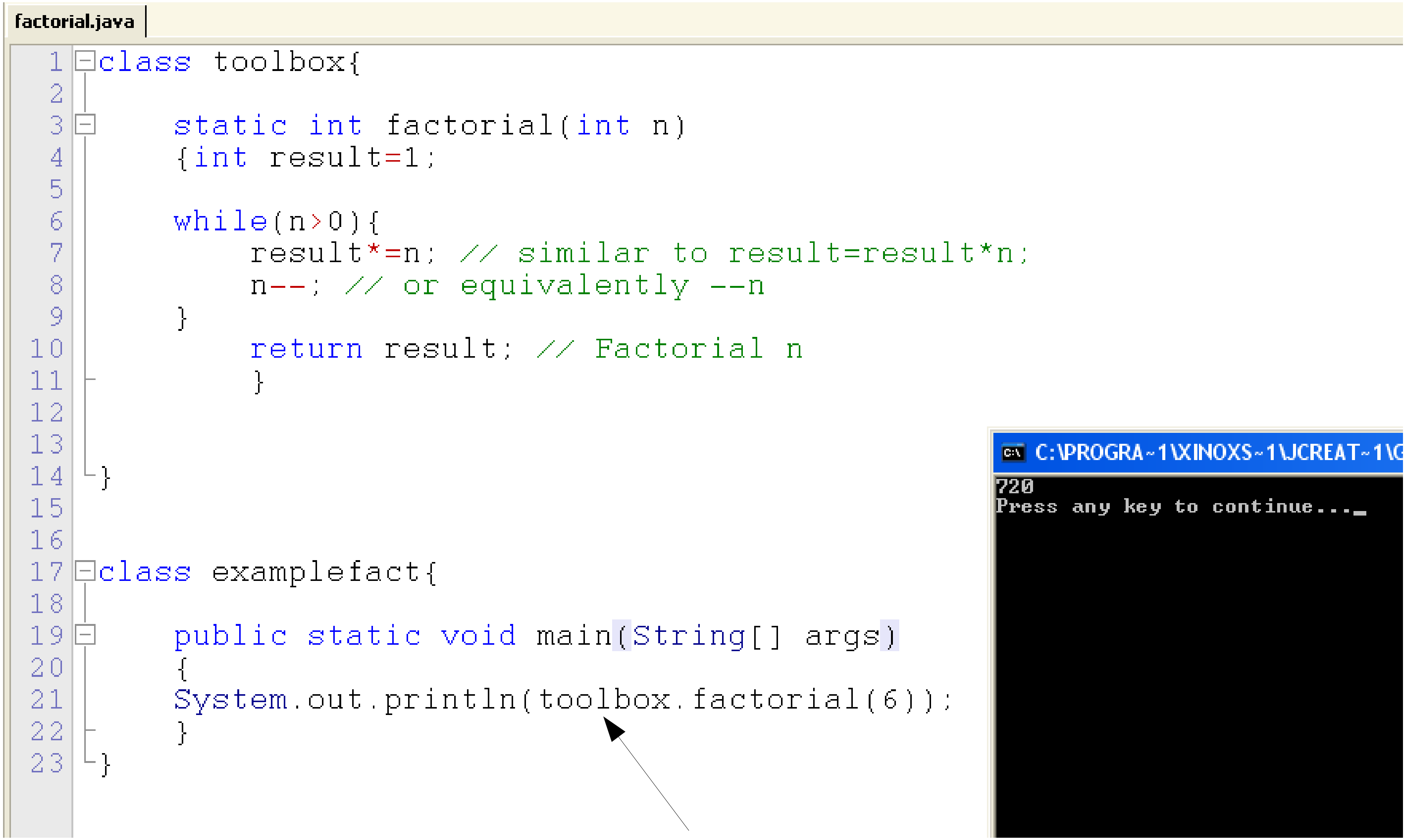


```java
class funcbranch{
    public static void main (String[] arguments)
    {
        double x=Math.E;

        System.out.println("Choose function to evalute for x="+x);
        System.out.print("(1) Identity, (2) Logarithm, (3) Sinus. Your choice ?");

        int t=TC.lireInt();

        System.out.println("F(x)="+F(t,x));
    }

    // The function is declared after the main body
    // Java handles well this declaration

    public static double F(int generator, double x)
    {double v=0.0;

        switch(generator)
        {
            case 1: v=x; break;
            case 2: v=Math.log(x); break;
            case 3: v=Math.sin(x); break;

        }

        return v;
    }
}
```

```
Choose function to evalute for x=2.718281828459045
(1) Identity, (2) Logarithm, (3) Sinus. Your choice ?2
F(x)=1.0
Press any key to continue...
```

Factorial function n! in Java

$n! = \prod_{k=1}^{n} k \qquad \forall n \in \mathbb{N}. \qquad 6! = 1 \times 2 \times 3 \times 4 \times 5 \times 6 = 720$



```
class toolbox{

    static int factorial(int n)
    {int result=1;

    while(n>0){
        result*=n; // similar to result=result*n;
        n--; // or equivalently --n
    }
        return result; // Factorial n
        }



}


class examplefact{

    public static void main(String[] args)
    {
    System.out.println(toolbox.factorial(6));
    }
}
```

```
720
Press any key to continue..._
```

Call function factorial in class « toolbox »

Calling functions: Inner Mechanism

```
TypeF result=F(param1, param2, ..., paramN);
```

param1, ..., paramN should be of the **same types** as the ones declared in the function

A function call can be used inside an expression,
or even as a parameter of another function (nested calls)
Example: `F1(F2(x),F3(x))`

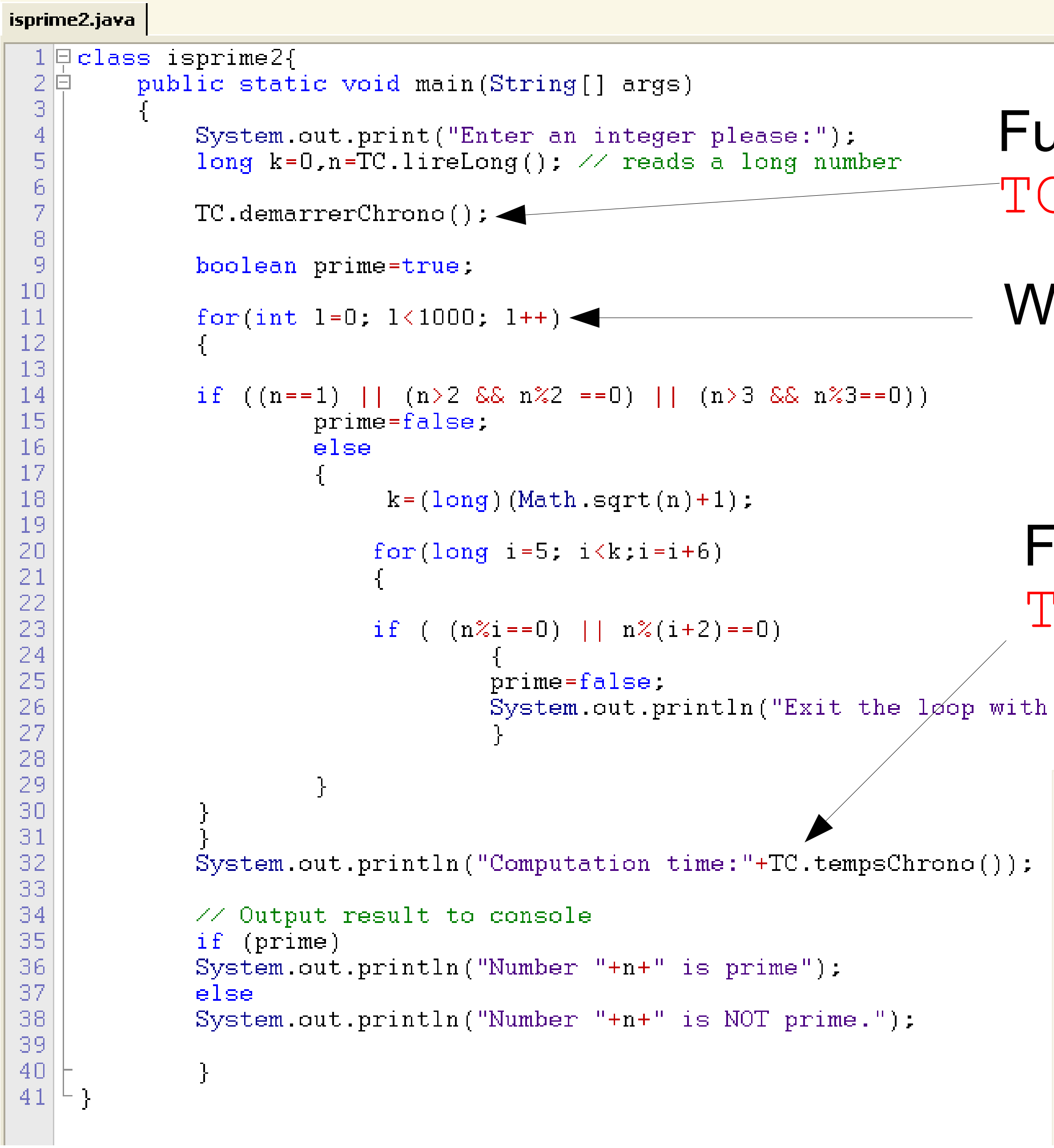
Assignment's rule checks **at compile time** for type equivalence:

```
System.out.println(IsPrime(23121971));
double dist=distance(u,v);
```

Beyond the scope of the function's class, we need to put the function' class with a dot. Requires the function to be public.

```
Math.cos(x);
TD2.factorial(n);
TC.lireInt();
```

Revisiting IsPrime: measuring time

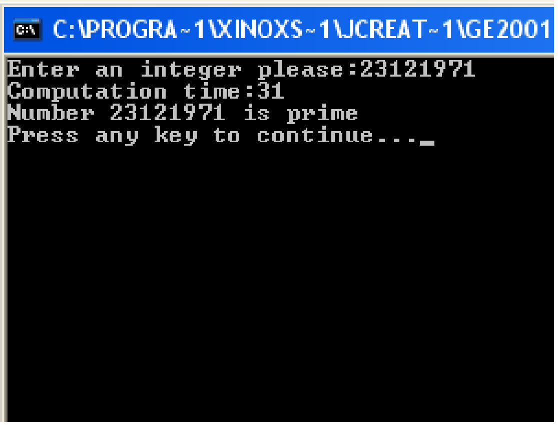


```java
class isprime2{
	public static void main(String[] args)
	{
		System.out.print("Enter an integer please:");
		long k=0,n=TC.lireLong(); // reads a long number

		TC.demarrerChrono();

		boolean prime=true;

		for(int l=0; l<1000; l++)
		{

		if ((n==1) || (n>2 && n%2 ==0) || (n>3 && n%3==0))
			prime=false;
			else
			{
			  k=(long)(Math.sqrt(n)+1);

			   for(long i=5; i<k;i=i+6)
			   {

			   if ( (n%i==0) || n%(i+2)==0)
					{
					prime=false;
					System.out.println("Exit the loop with k="+k);
					}


			}
		}
		}
		System.out.println("Computation time:"+TC.tempsChrono());

		// Output result to console
		if (prime)
		System.out.println("Number "+n+" is prime");
		else
		System.out.println("Number "+n+" is NOT prime.");

		}
}
```

Function call in class TC:
TC.demarrerChrono();

We repeat this computation 1000 times to measure the elapsed time

Function call in class TC:
TC.tempsChrono();



```
Enter an integer please:23121971
Computation time:31
Number 23121971 is prime
Press any key to continue..._
```

Potential side effects of functions: Static variables (effet de bord)

- Function that ***might modify/alterate*** the environment

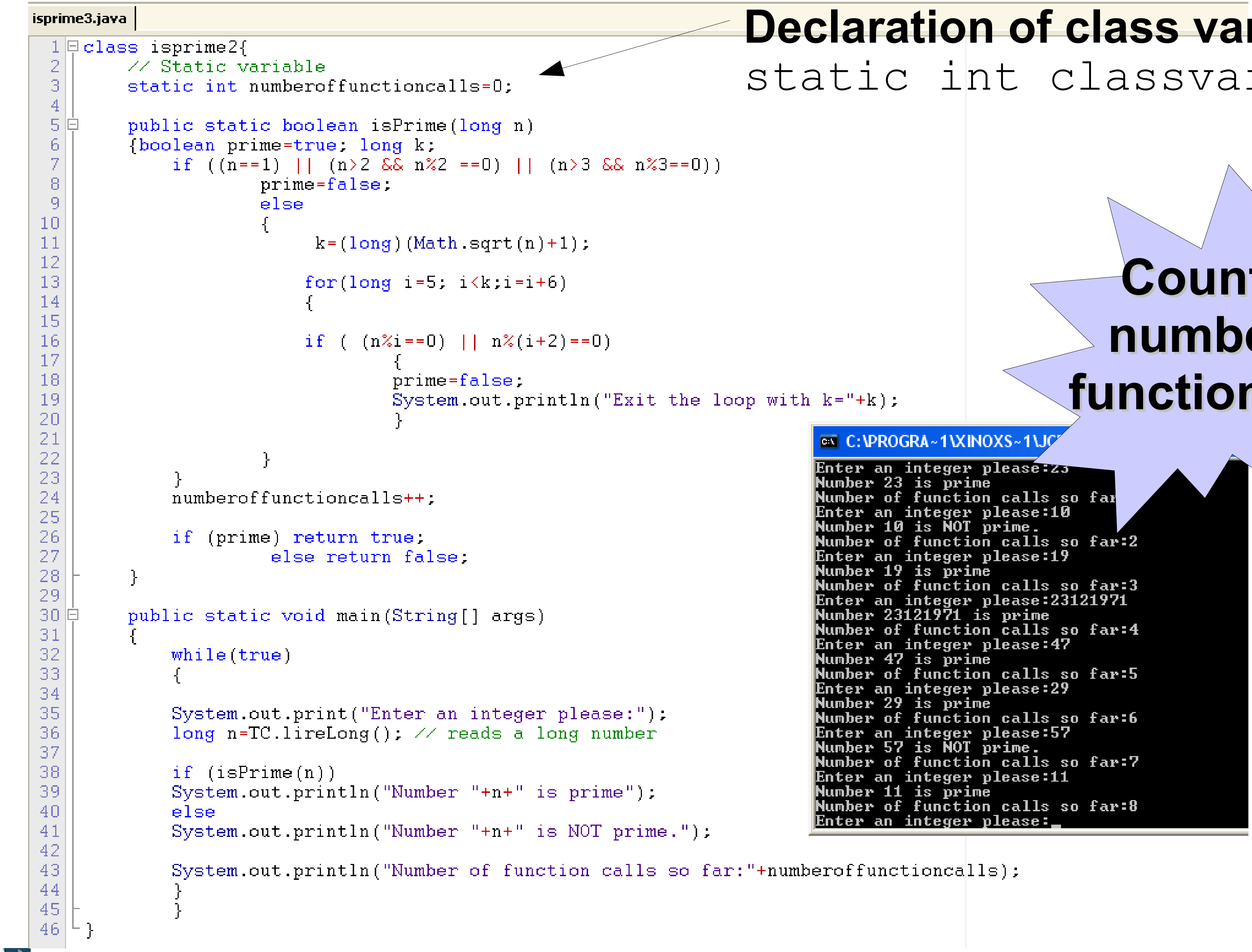
For example:

... displaying a value
... But also *modify a variable* of the base class

- A **class variable** is declared inside the class scope,
...not in function bodies

- Class variables are declared using the keyword **static**

Side effects of functions: Static variables



isprime3.java

```java
class isprime2{
    // Static variable
    static int numberoffunctioncalls=0;

    public static boolean isPrime(long n)
    {boolean prime=true; long k;
        if ((n==1) || (n>2 && n%2 ==0) || (n>3 && n%3==0))
                prime=false;
                else
                {
                     k=(long)(Math.sqrt(n)+1);

                    for(long i=5; i<k;i=i+6)
                    {

                    if ( (n%i==0) || n%(i+2)==0)
                            {
                            prime=false;
                            System.out.println("Exit the loop with k="+k);
                            }


                }
        }
        numberoffunctioncalls++;

        if (prime) return true;
                 else return false;
    }

    public static void main(String[] args)
    {
        while(true)
        {

        System.out.print("Enter an integer please:");
        long n=TC.lireLong(); // reads a long number

        if (isPrime(n))
        System.out.println("Number "+n+" is prime");
        else
        System.out.println("Number "+n+" is NOT prime.");

        System.out.println("Number of function calls so far:"+numberoffunctioncalls);
        }
        }
}
```

C:\PROGRA~1\XINOXS~1\JC

```
Enter an integer please:23
Number 23 is prime
Number of function calls so far
Enter an integer please:10
Number 10 is NOT prime.
Number of function calls so far:2
Enter an integer please:19
Number 19 is prime
Number of function calls so far:3
Enter an integer please:23121971
Number 23121971 is prime
Number of function calls so far:4
Enter an integer please:47
Number 47 is prime
Number of function calls so far:5
Enter an integer please:29
Number 29 is prime
Number of function calls so far:6
Enter an integer please:57
Number 57 is NOT prime.
Number of function calls so far:7
Enter an integer please:11
Number 11 is prime
Number of function calls so far:8
Enter an integer please:_
```

Declaration of class variable

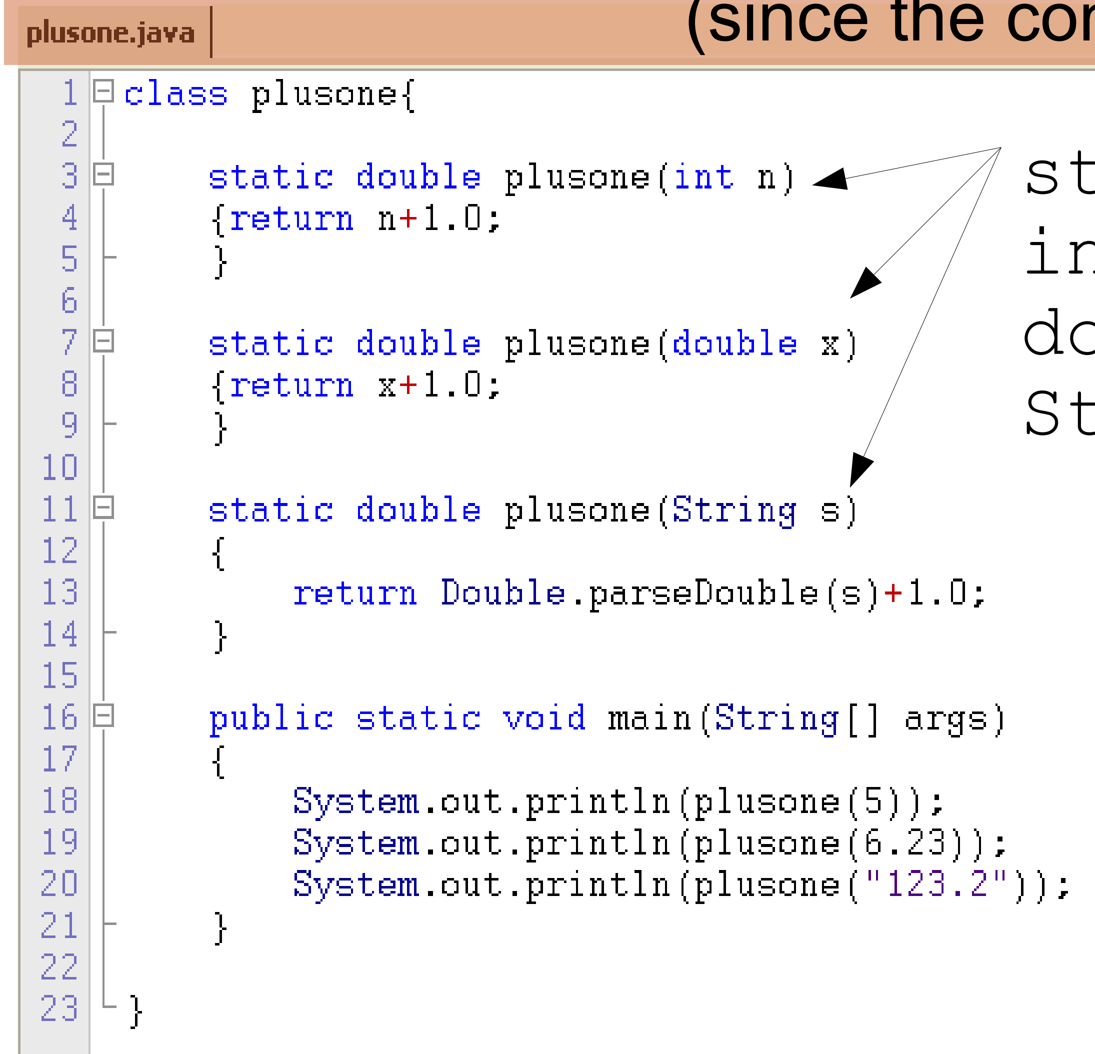
`static int classvar;`

Counting number of function calls

Function: Signature and overloading

signature of a function = **ordered sequence of parameter types**

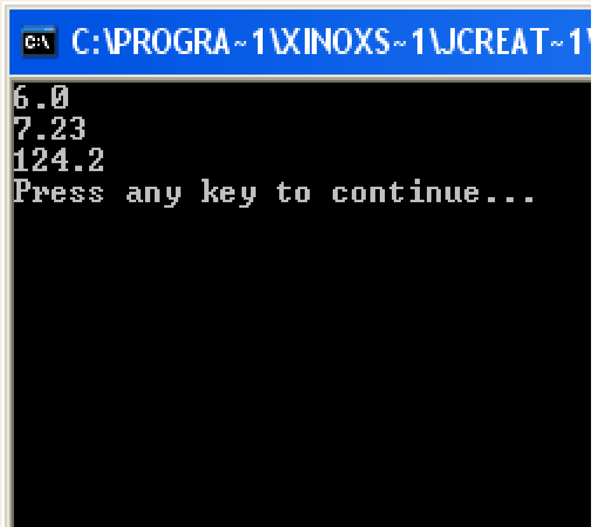
Two functions with **different signatures** can bear the **same name** (since the compiler can distinguish them!)



plusone.java

```
class plusone{

    static double plusone(int n)
    {return n+1.0;
    }

    static double plusone(double x)
    {return x+1.0;
    }

    static double plusone(String s)
    {
        return Double.parseDouble(s)+1.0;
    }

    public static void main(String[] args)
    {
        System.out.println(plusone(5));
        System.out.println(plusone(6.23));
        System.out.println(plusone("123.2"));
    }

}
```

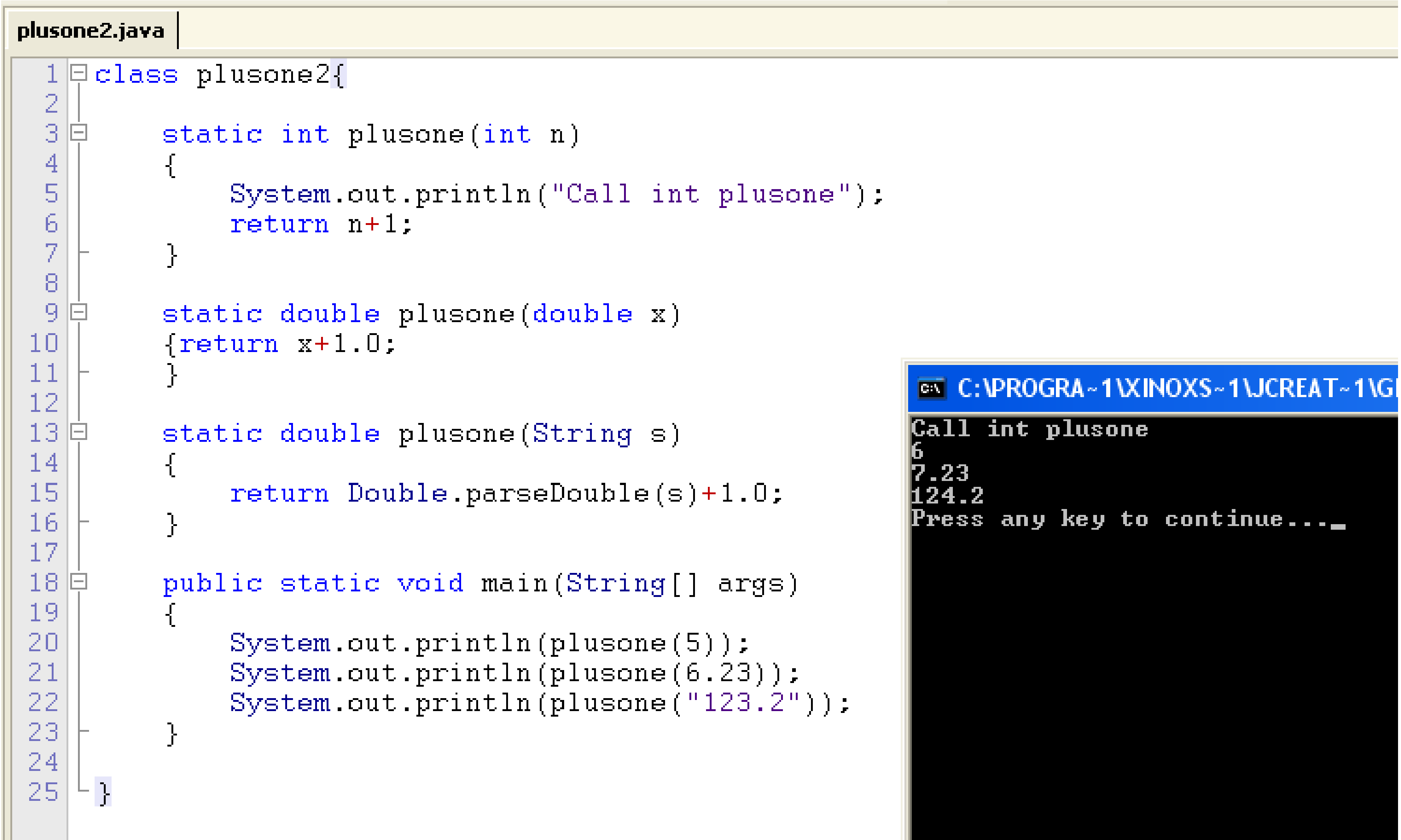
static double plusone(...)
int
double
String



```
6.0
7.23
124.2
Press any key to continue...
```

Function: Signature and overloading

Although the function result type is important,
Java *does not* take into account it for creating signatures...



```java
class plusone2{

    static int plusone(int n)
    {
        System.out.println("Call int plusone");
        return n+1;
    }

    static double plusone(double x)
    {return x+1.0;
    }

    static double plusone(String s)
    {
        return Double.parseDouble(s)+1.0;
    }

    public static void main(String[] args)
    {
        System.out.println(plusone(5));
        System.out.println(plusone(6.23));
        System.out.println(plusone("123.2"));
    }

}
```

```
Call int plusone
6
7.23
124.2
Press any key to continue..._
```

Function: Signature and overloading

```
static int plusone (int n)
static double plusone(int n)
```

!!! COMPILATION ERROR !!!

```
class SignatureError{
  public static int plusone(int n)
  {return n+1;}

  public static double plusone(int n)
  {return n+1.0;}

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

```
C:\J\Signature.java:6: plusone(int) is already defined in SignatureError
 static double plusone(int n)
```

Executing functions in Java

- **Work place** of the function is created when the function is called

- ... and destroyed once it is executed (value returned)

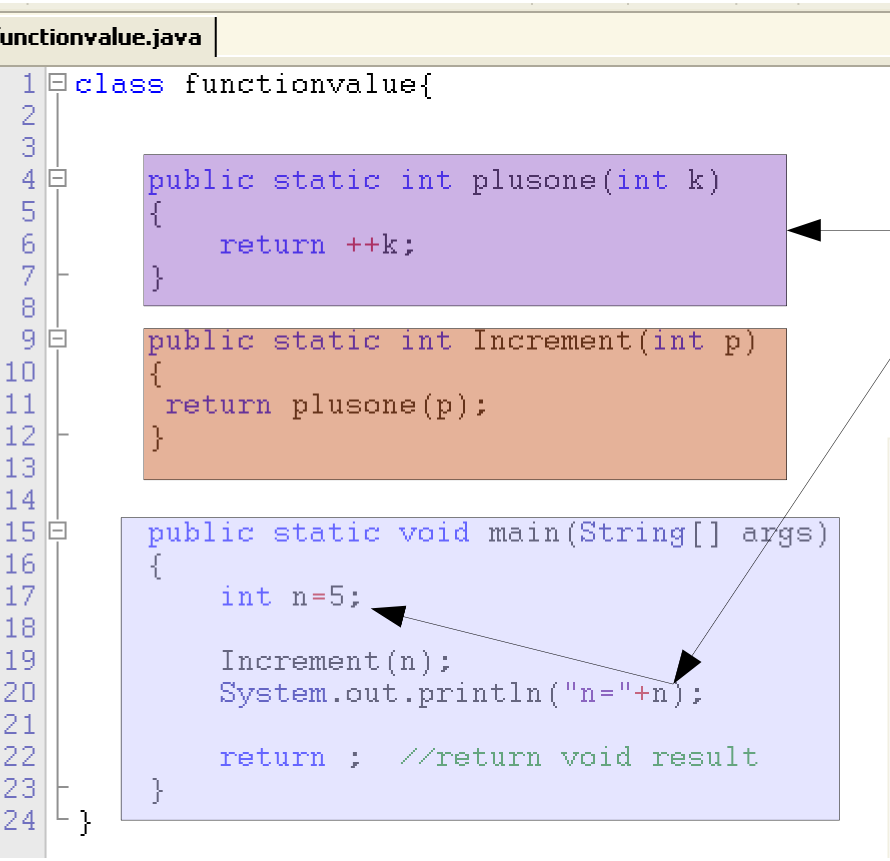
- Parameter values are equal to the results of the expressions

- Function parameters are allocated in memory reserved for the function

- If a parameter is modified inside the function body,
  it remains unchanged in the calling function.

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

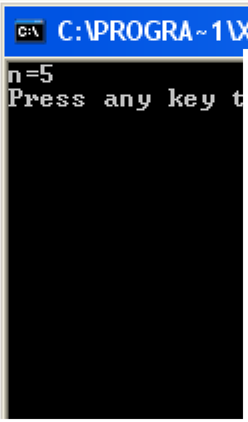
Executing functions in Java



unctionvalue.java

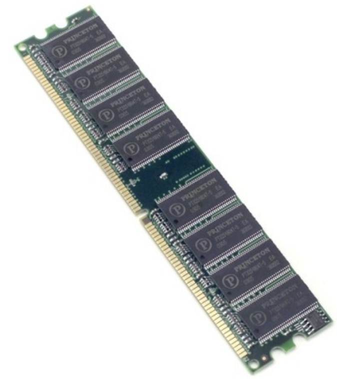
```
class functionvalue{


    public static int plusone(int k)
    {
        return ++k;
    }

    public static int Increment(int p)
    {
     return plusone(p);
    }

    public static void main(String[] args)
    {
        int n=5;

        Increment(n);
        System.out.println("n="+n);

        return ;  //return void result
    }
}
```

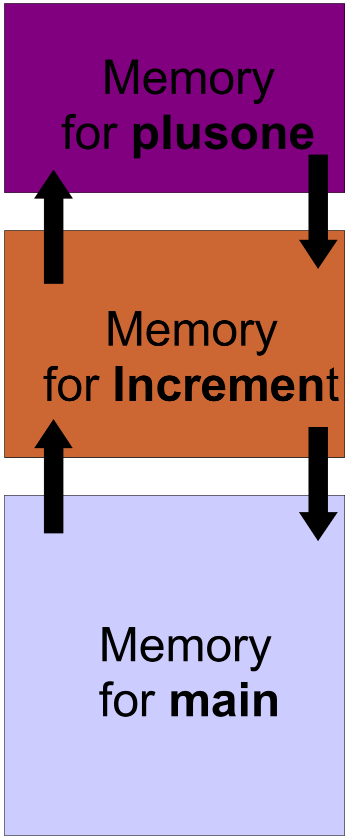
As soon as we exit this function, k takes its original value of (5)



C:\PROGRA~1\XINOXS~1\JCREAT~1\GE2

```
n=5
Press any key t
```

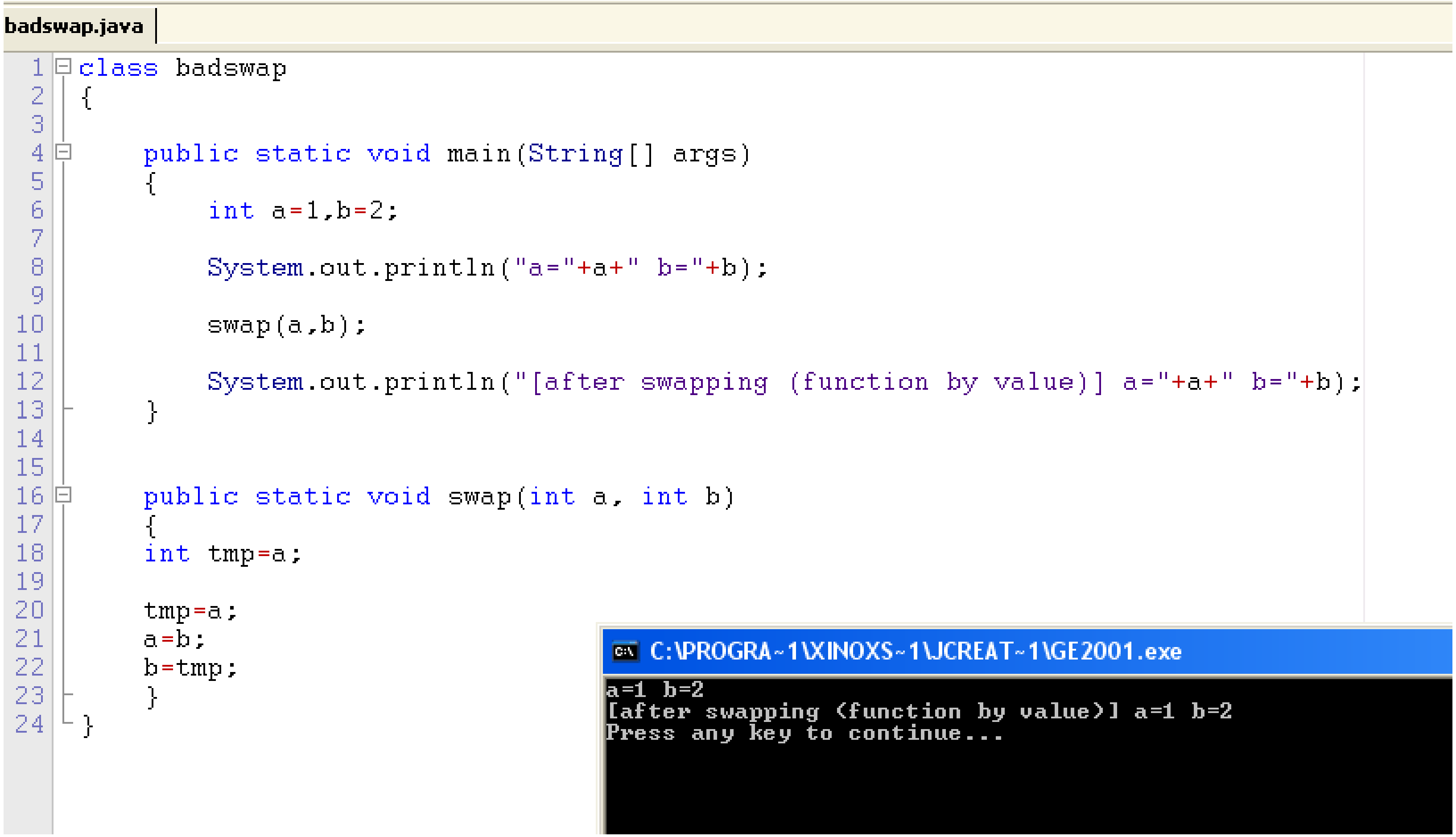




Memory (stack)

passage par valeur

Executing functions in Java

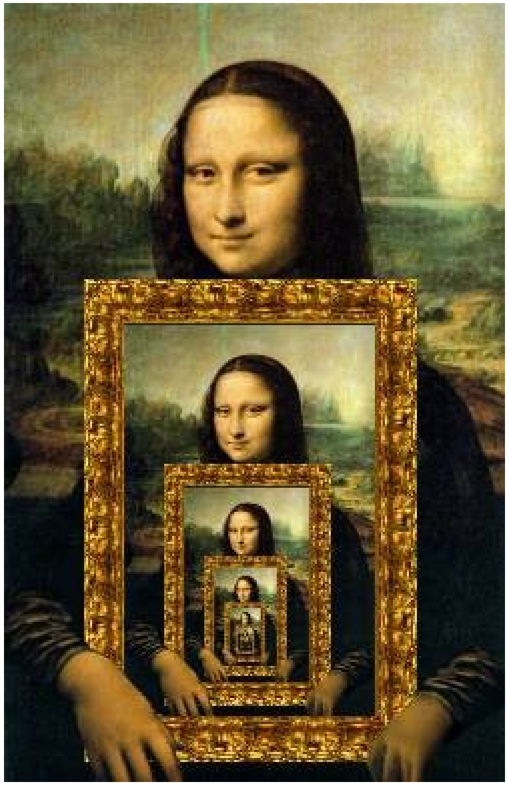


```java
class badswap
{

    public static void main(String[] args)
    {
        int a=1,b=2;

        System.out.println("a="+a+" b="+b);

        swap(a,b);

        System.out.println("[after swapping (function by value)] a="+a+" b="+b);
    }


    public static void swap(int a, int b)
    {
    int tmp=a;

    tmp=a;
    a=b;
    b=tmp;
    }
}
```

```
a=1 b=2
[after swapping (function by value)] a=1 b=2
Press any key to continue...
```

(In C++, swapping is easy)

Principle of recursion



A beautiful **principle of computing** !
Loosely speaking, ...
...the inverse of inductivism in mathematics

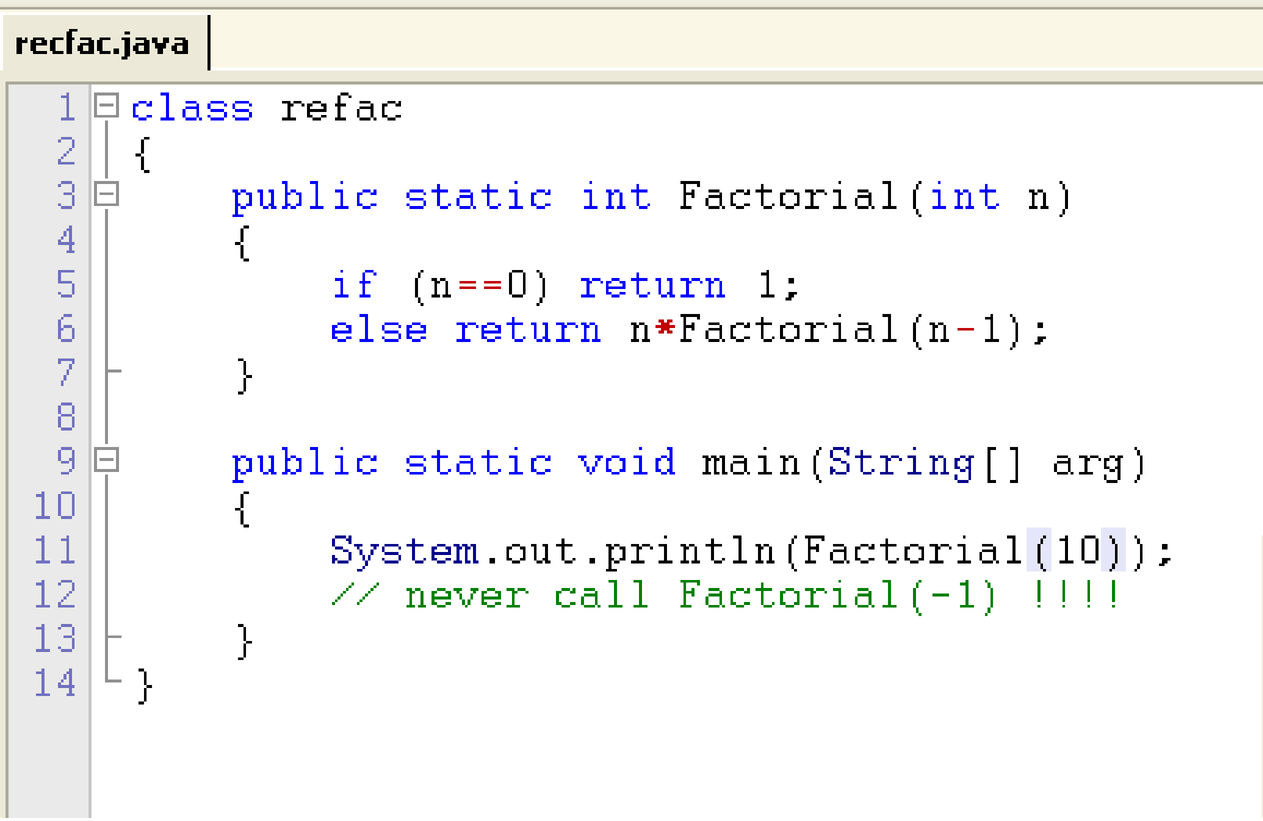
- A function that **calls itself**...

- ...not forever, so that there should be **stopping states**...

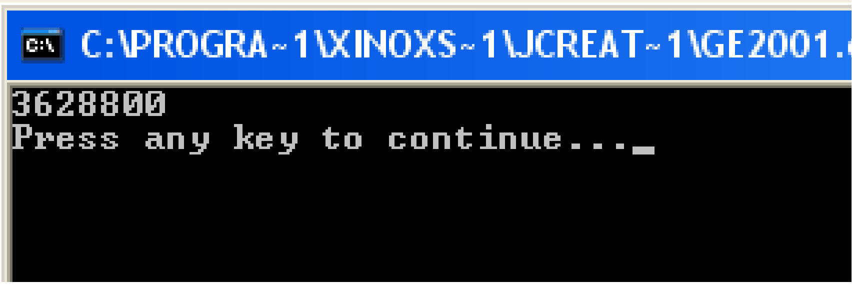
- ...Function parameters *should tend* to the ones that do not ...require recursion to finalize the computation...

But all this is an *informal glimpse* of recursion (self-structure)

Example: Revisiting the factorial



```java
class refac
{
    public static int Factorial(int n)
    {
        if (n==0) return 1;
        else return n*Factorial(n-1);
    }

    public static void main(String[] arg)
    {
        System.out.println(Factorial(10));
        // never call Factorial(-1) !!!!
    }
}
```



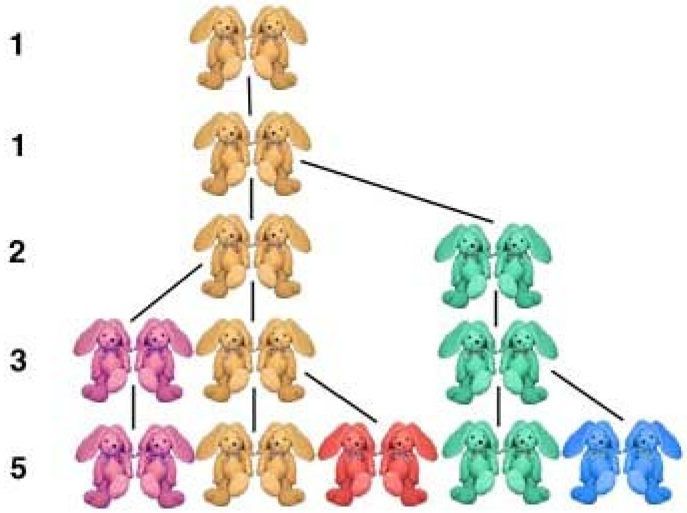
```
3628800
Press any key to continue..._
```

Example: Fibonacci numbers

Leonard de Pise
(1170- 1245)

$$\mathcal{F}_1 = \mathcal{F}_2 = 1$$

$$\mathcal{F}_{n+2} = \mathcal{F}_{n+1} + \mathcal{F}_n$$



1, 1, 2, 3, 5, 8, 13, 21, 34, 55......

Population growth:

Newly born pair of M/F rabbits are put in a field.

Newly born rabbits take a month to become mature, after which time

... They produce a new pair of baby rabbits every month

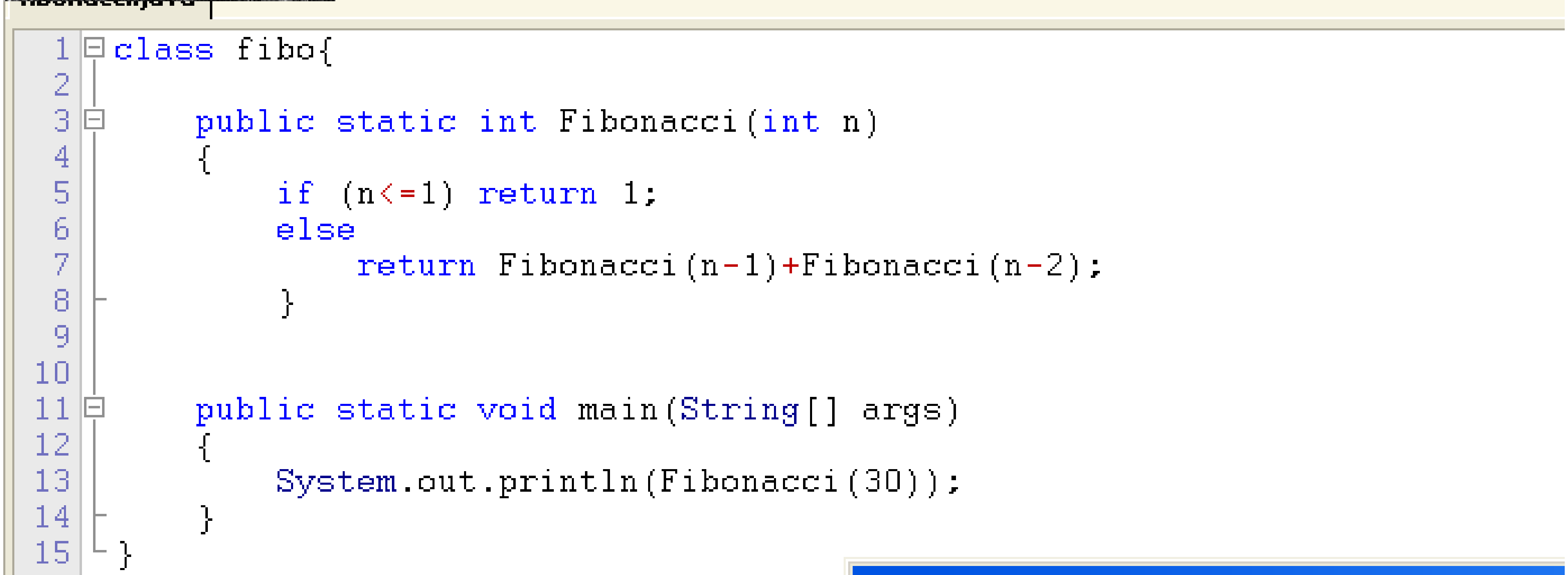
Q.: How many pairs will there be in subsequent years?

Example: Fibonacci numbers

Leonard de Pise

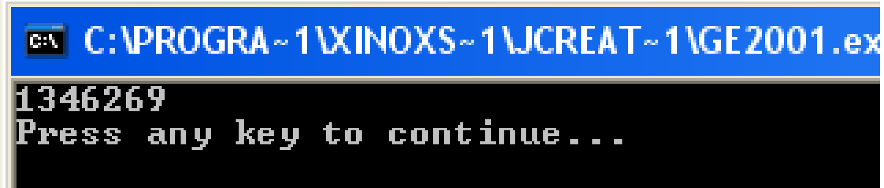
$$\mathcal{F}_1 = \mathcal{F}_2 = 1$$

$$\mathcal{F}_{n+2} = \mathcal{F}_{n+1} + \mathcal{F}_n$$



```java
class fibo{

	public static int Fibonacci(int n)
	{
		if (n<=1) return 1;
		else
			return Fibonacci(n-1)+Fibonacci(n-2);
		}


	public static void main(String[] args)
	{
		System.out.println(Fibonacci(30));
	}
}
```

Much better algorithms at....
http://fr.wikipedia.org/wiki/Suite_de_Fibonacci



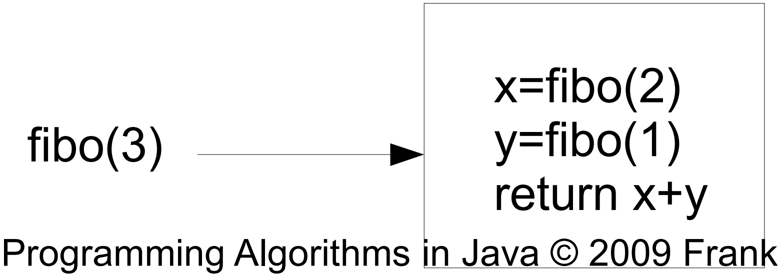
```
1346269
Press any key to continue...
```

Understanding a recursive function

```
int fibo(int n)
{int x,y;
   if(n <= 1) return 1;
   x=fibo(n-1);
  y=fibo(n-2);
   return  x+y;}
```

recursive function called:

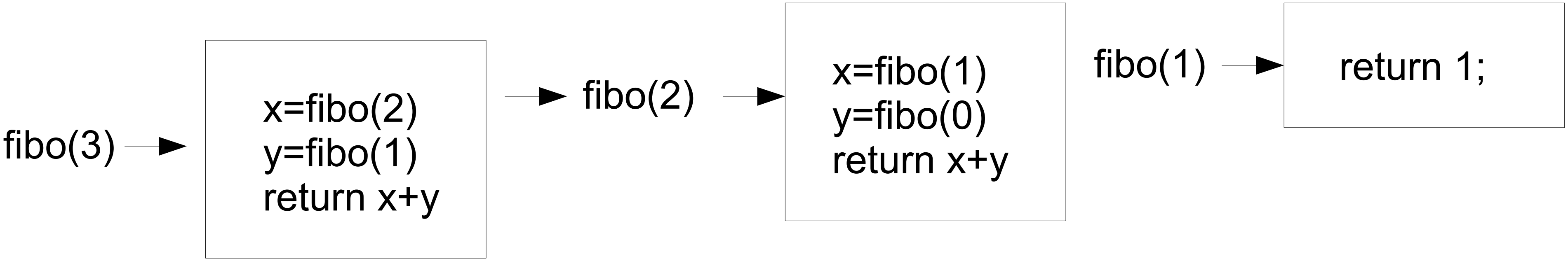
- Allocation of memory for local variables
- Stack operations to compute
- ... Call the function with other parameters, if required
- Process operations that remains on the stack


fibo(3)

x=fibo(2)
y=fibo(1)
return x+y

Recursive calls

Understanding a recursive function



fibo(3)

x=fibo(2)
y=fibo(1)
return x+y

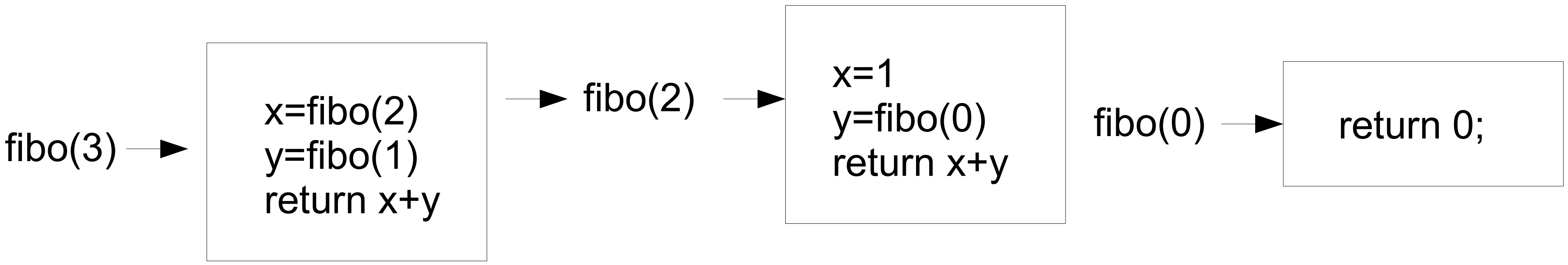
fibo(2)

x=fibo(1)
y=fibo(0)
return x+y

fibo(1)

return 1;

Understanding a recursive function



fibo(3)

x=fibo(2)
y=fibo(1)
return x+y

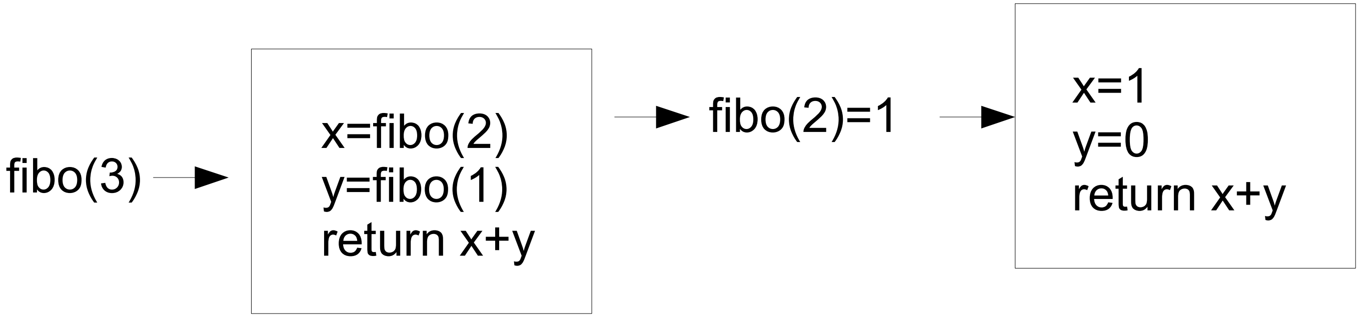
fibo(2)

x=1
y=fibo(0)
return x+y

fibo(0)

return 0;

Understanding a recursive function



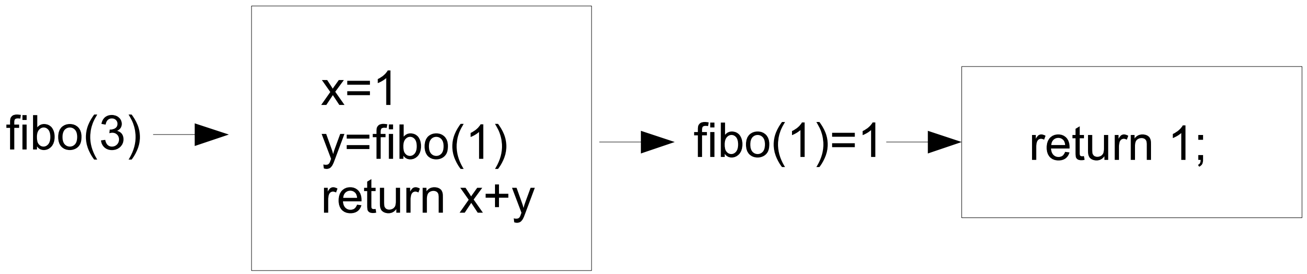
fibo(3)

x=fibo(2)
y=fibo(1)
return x+y

fibo(2)=1

x=1
y=0
return x+y

Understanding a recursive function



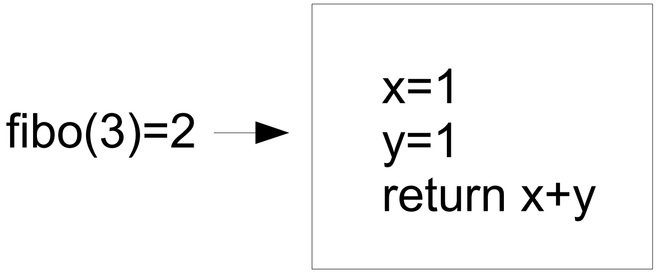
fibo(3)

x=1
y=fibo(1)
return x+y

fibo(1)=1

return 1;

Understanding a recursive function



fibo(3)=2

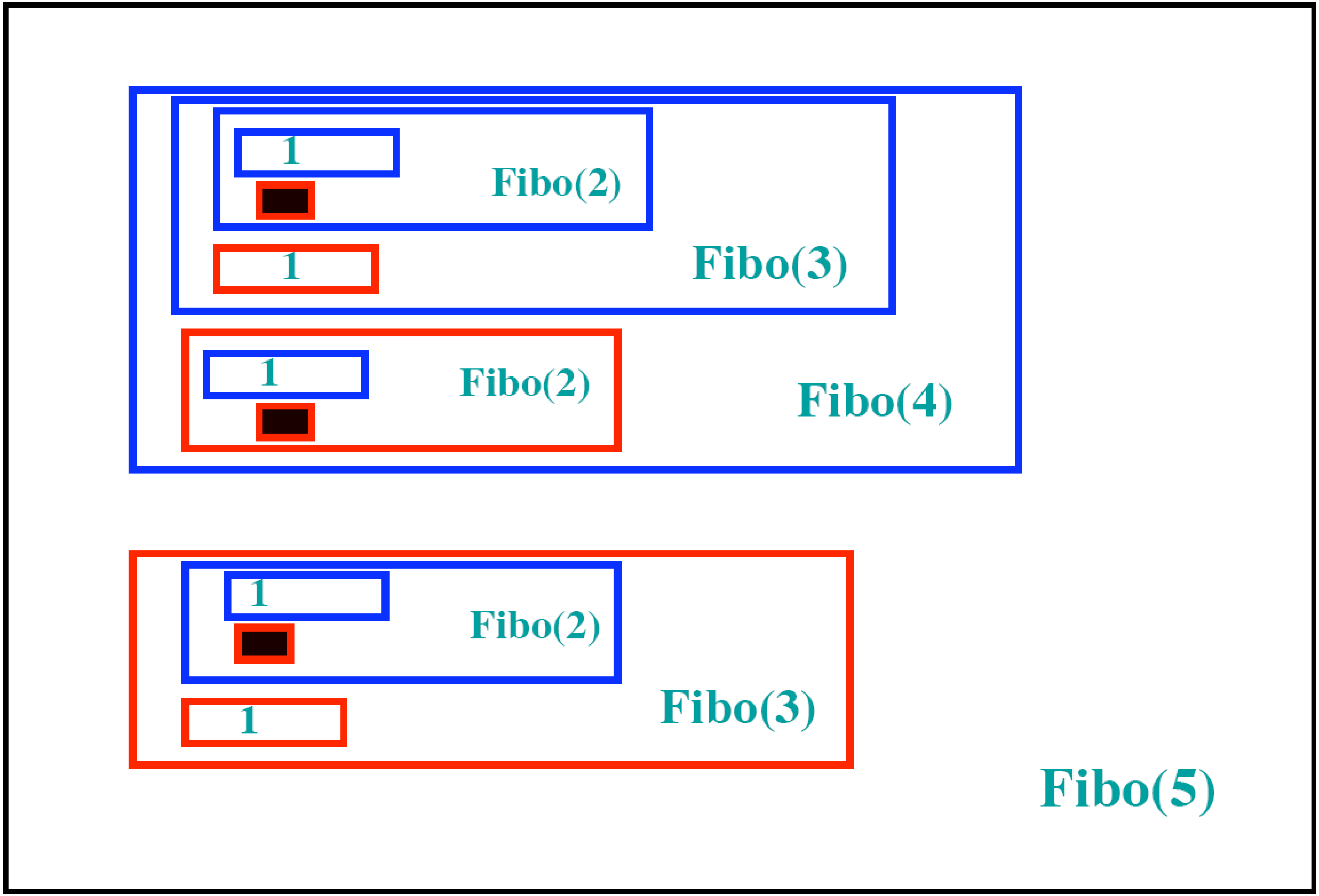
x=1
y=1
return x+y

As we can see, there is a lot of redundant work here.
-> Very inefficient algorithm.

Can cause **stack overflow** if the #recursive calls...
...become too large

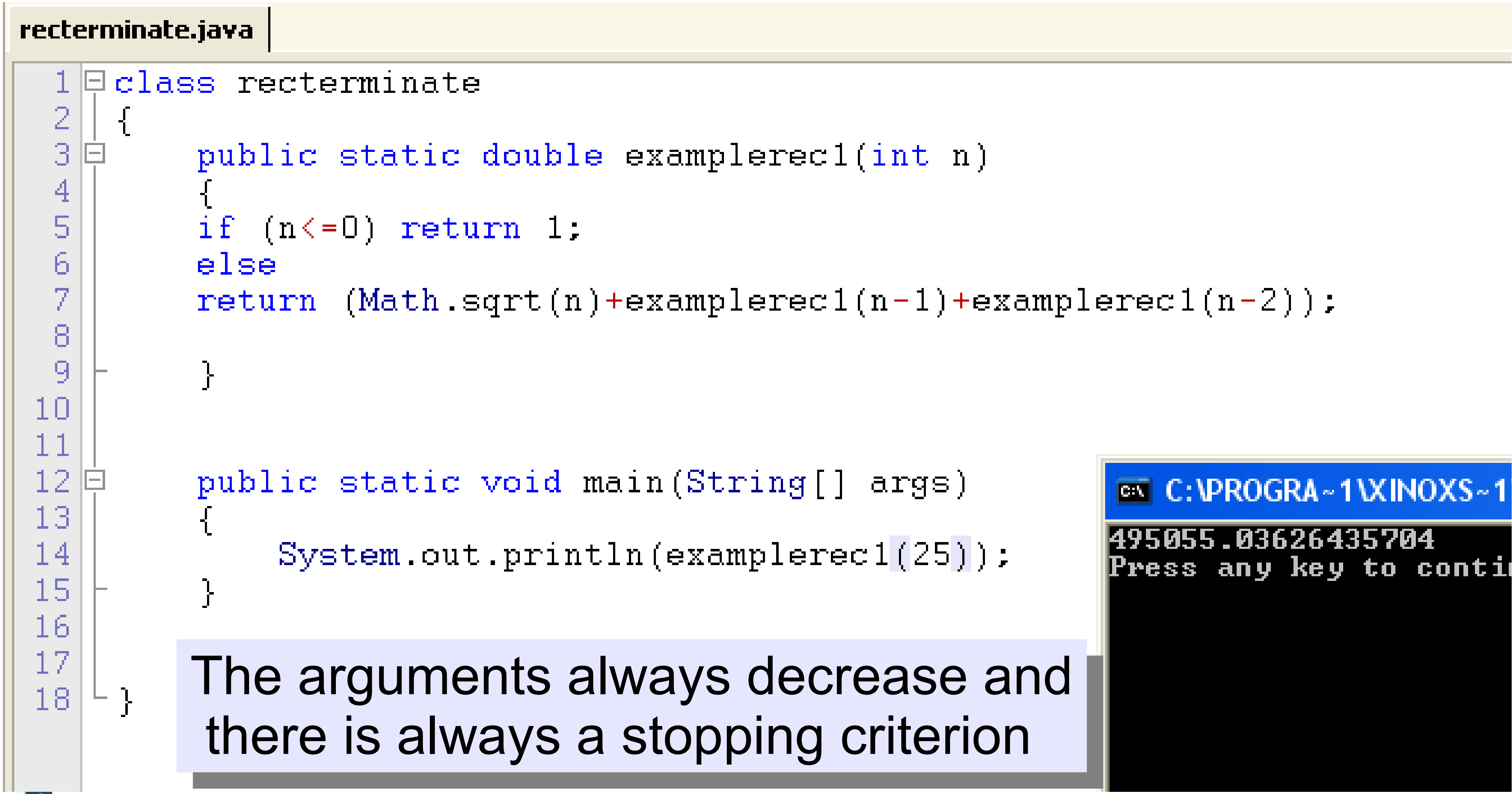
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, ..

Understanding a recursive function



Recursion: Halting problem

When does a recursive program terminate?



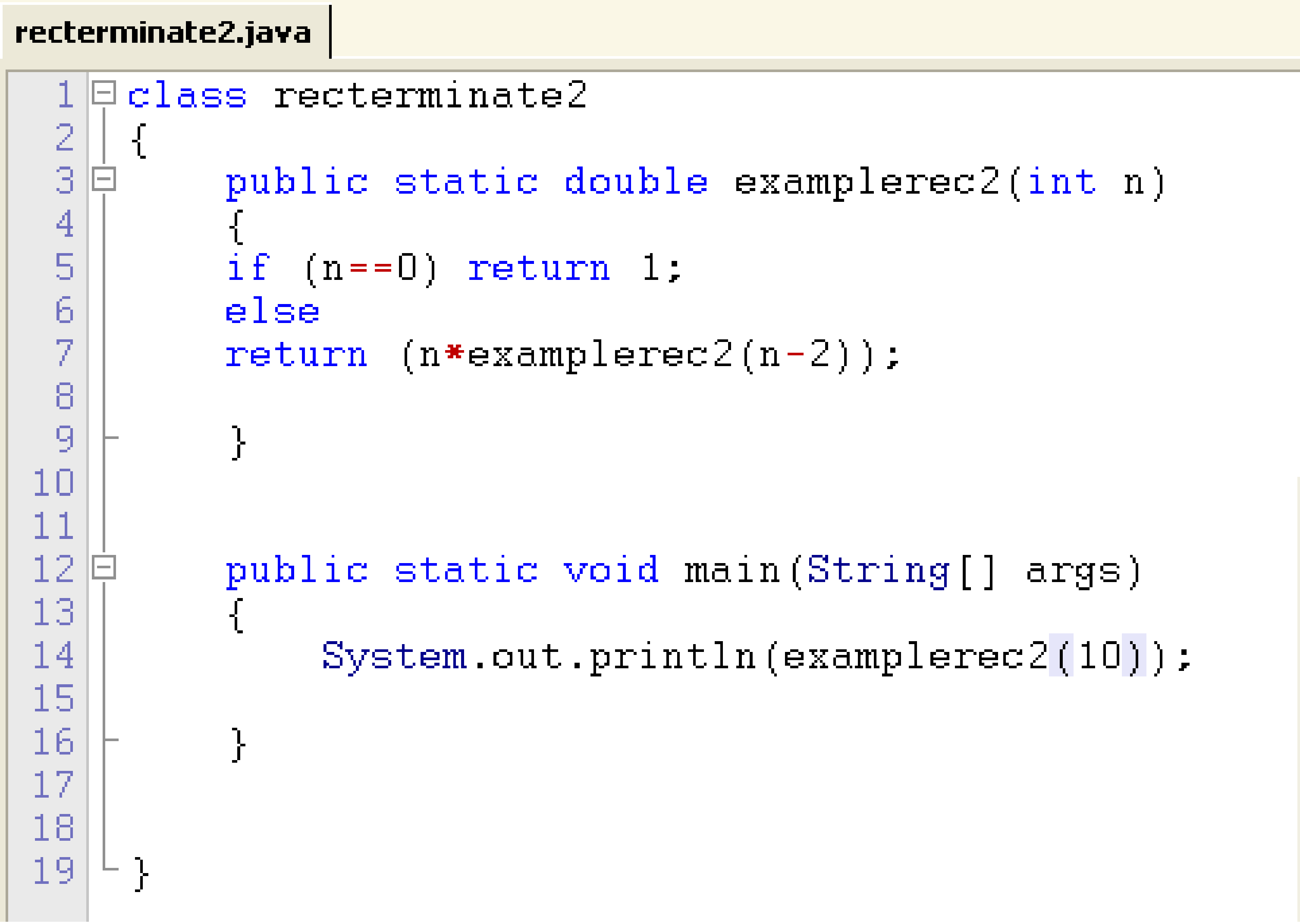
recterminate.java

```java
class recterminate
{
	public static double examplerec1(int n)
	{
	if (n<=0) return 1;
	else
	return (Math.sqrt(n)+examplerec1(n-1)+examplerec1(n-2));

	}


	public static void main(String[] args)
	{
		System.out.println(examplerec1(25));
	}


}
```

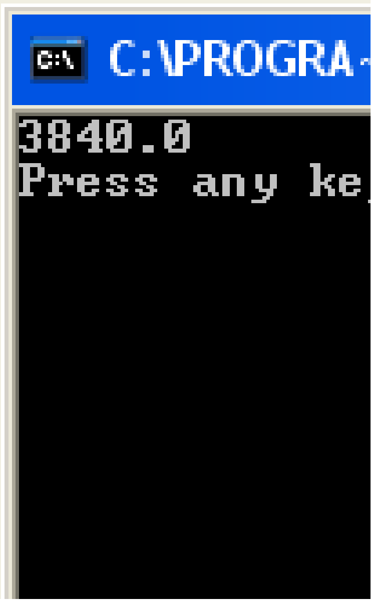
```
495055.03626435704
Press any key to conti
```

The arguments always decrease and there is always a stopping criterion

Recursion: Halting problem

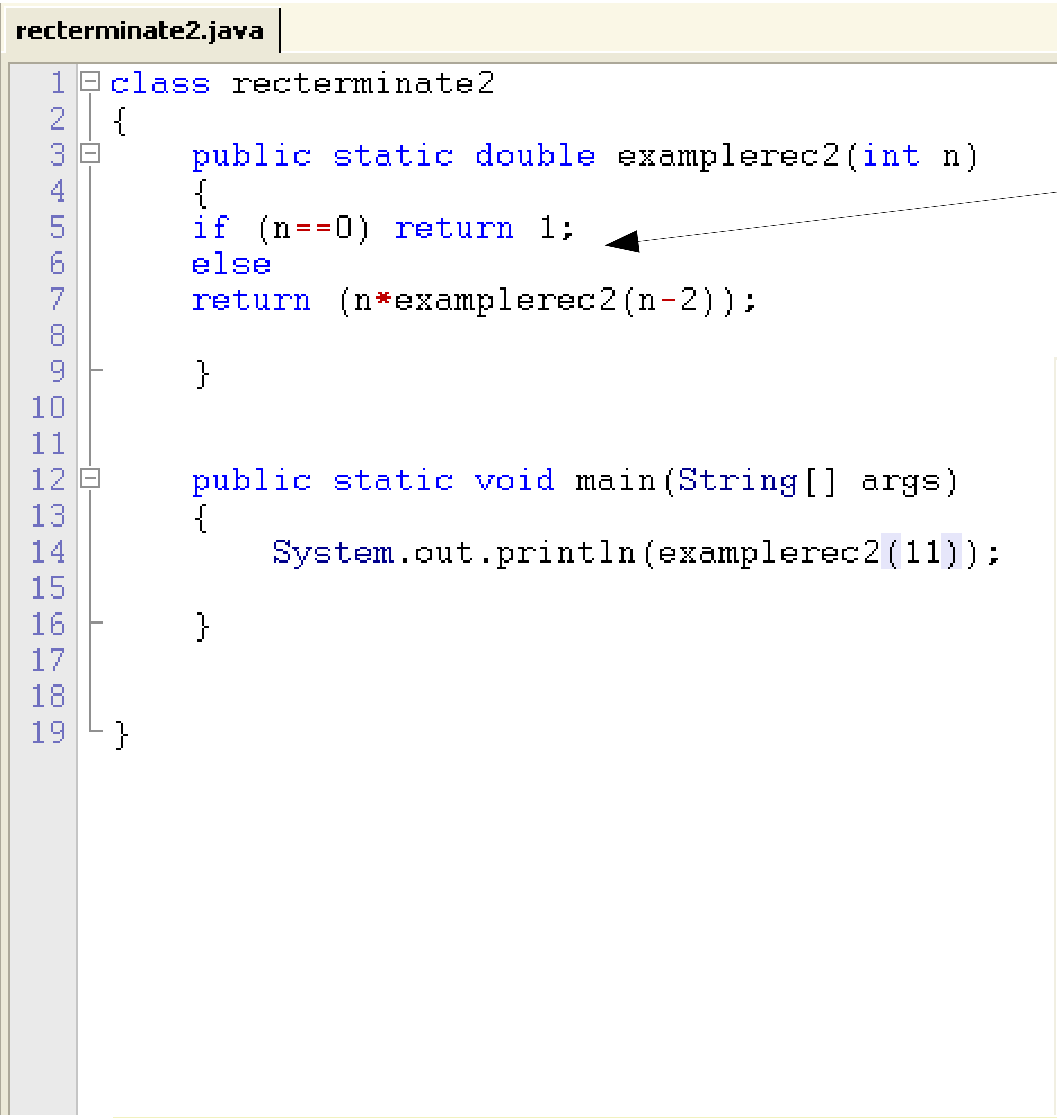


```
class recterminate2
{
    public static double examplerec2(int n)
    {
    if (n==0) return 1;
    else
    return (n*examplerec2(n-2));

    }


    public static void main(String[] args)
    {
        System.out.println(examplerec2(10));

    }


}
```



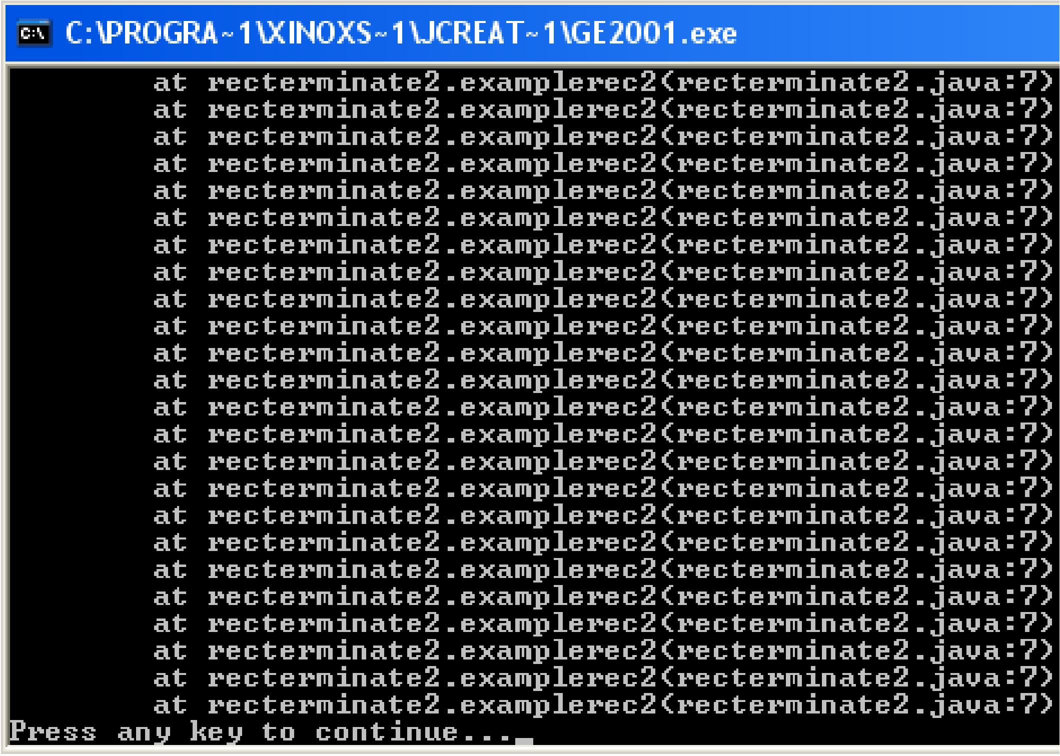
```
3840.0
Press any ke
```

Recursion: Halting problem



```java
class recterminate2
{
	public static double examplerec2(int n)
	{
	if (n==0) return 1;
	else
	return (n*examplerec2(n-2));

	}


	public static void main(String[] args)
	{
		System.out.println(examplerec2(11));

	}


}
```

Do we always reach that terminal state?

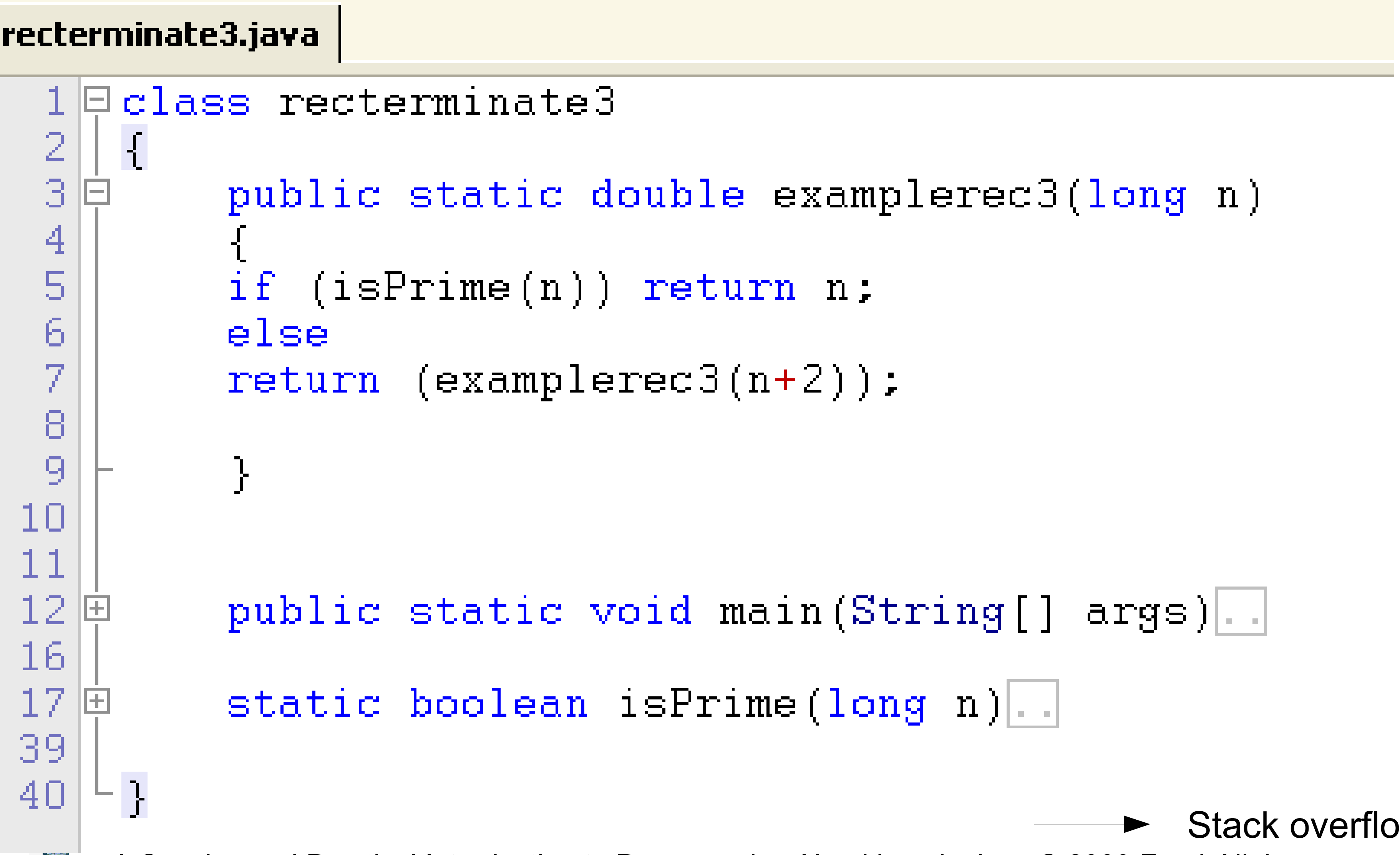


```
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
        at recterminate2.examplerec2(recterminate2.java:7)
Press any key to continue...
```

Does not always halt because
we may never reach terminal case (n=0) for odd numbers

Recursion: Halting problem

What do you think of this one?



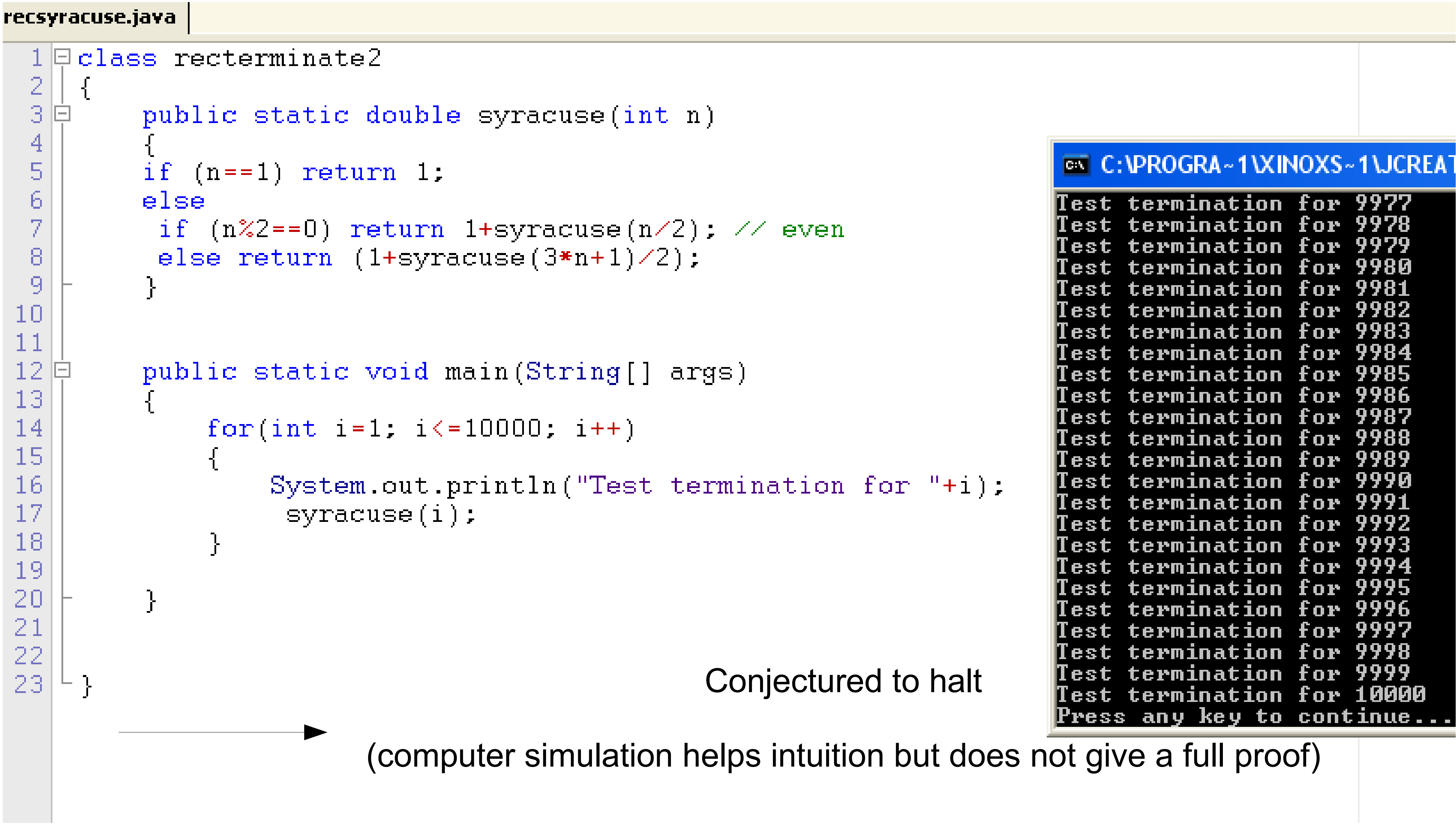
recterminate3.java

```
class recterminate3
{
    public static double examplerec3(long n)
    {
    if (isPrime(n)) return n;
    else
    return (examplerec3(n+2));

    }


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

    static boolean isPrime(long n)..

}
```

→ Stack overflow

Recursion: Halting problem

Syracuse problem and termination conjecture



recsyracuse.java

```java
class recterminate2
{
	public static double syracuse(int n)
	{
	if (n==1) return 1;
	else
	 if (n%2==0) return 1+syracuse(n/2); // even
	 else return (1+syracuse(3*n+1)/2);
	}


	public static void main(String[] args)
	{
		for(int i=1; i<=10000; i++)
		{
			System.out.println("Test termination for "+i);
			 syracuse(i);
		}

	}

}
```

```
Test termination for 9977
Test termination for 9978
Test termination for 9979
Test termination for 9980
Test termination for 9981
Test termination for 9982
Test termination for 9983
Test termination for 9984
Test termination for 9985
Test termination for 9986
Test termination for 9987
Test termination for 9988
Test termination for 9989
Test termination for 9990
Test termination for 9991
Test termination for 9992
Test termination for 9993
Test termination for 9994
Test termination for 9995
Test termination for 9996
Test termination for 9997
Test termination for 9998
Test termination for 9999
Test termination for 10000
Press any key to continue...
```

Conjectured to halt

(computer simulation helps intuition but does not give a full proof)

Halting problem: Computer Science

There is provably no algorithm that can take as input a program (binary string) and return true if and only if this program halts.

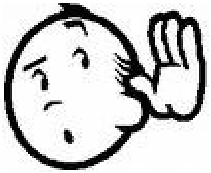
Proof skipped

http://en.wikipedia.org/wiki/Halting_problem

Terminal recursion

```
if (n<=1) return 1; else
return n*f(n-1);
```

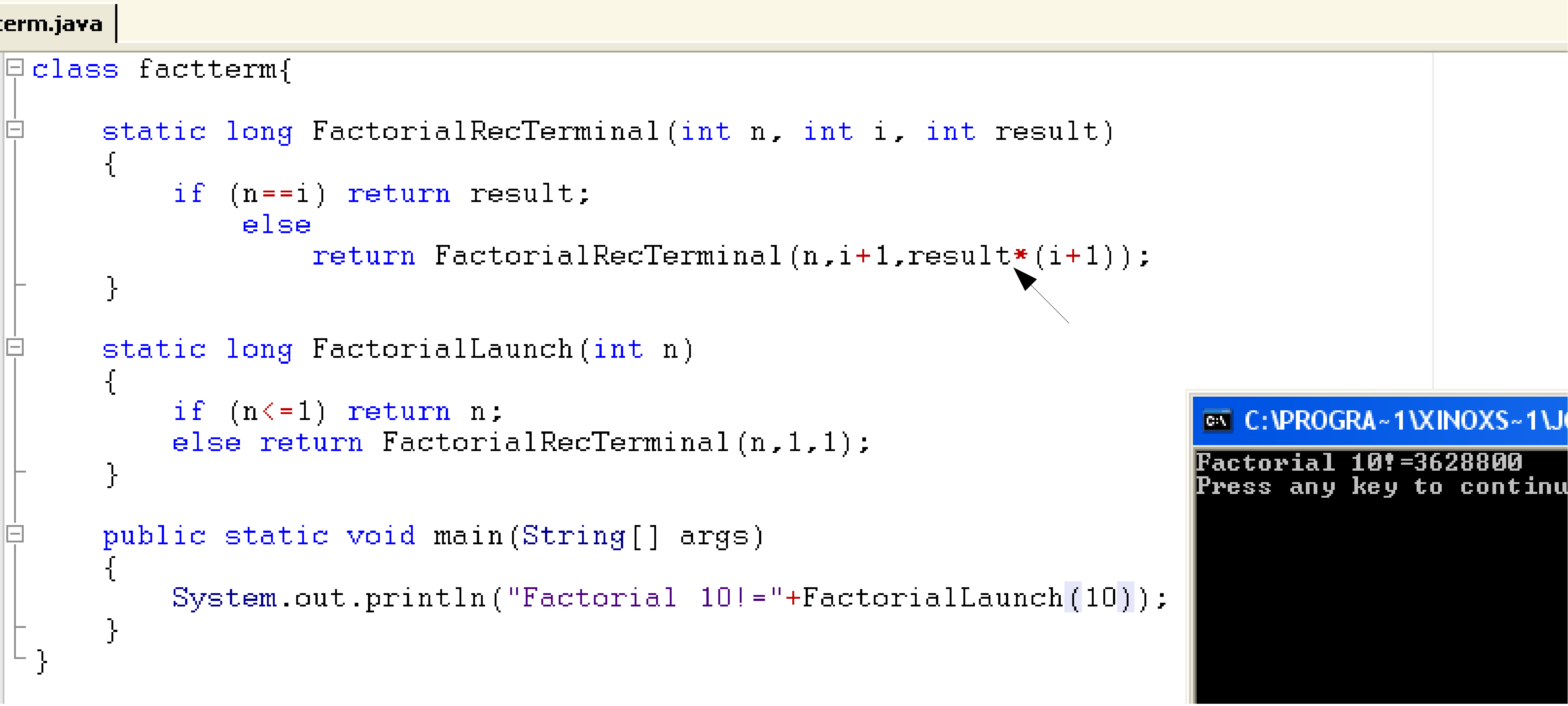
What happens if we call Factorial(100) ?



Recursive calls are **always**of the form return f(...);
->No instruction (computation) after the function
(Factorial is not terminal since return n*f(n-1);)

Does not put function calls on the stack (thus avoid stack overflow)

factorial with terminal recursion

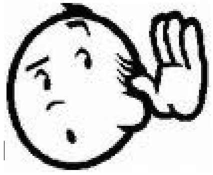


```
class factterm{

    static long FactorialRecTerminal(int n, int i, int result)
    {
        if (n==i) return result;
            else
                return FactorialRecTerminal(n,i+1,result*(i+1));
    }

    static long FactorialLaunch(int n)
    {
        if (n<=1) return n;
        else return FactorialRecTerminal(n,1,1);
    }

    public static void main(String[] args)
    {
        System.out.println("Factorial 10!="+FactorialLaunch(10));
    }
}
```

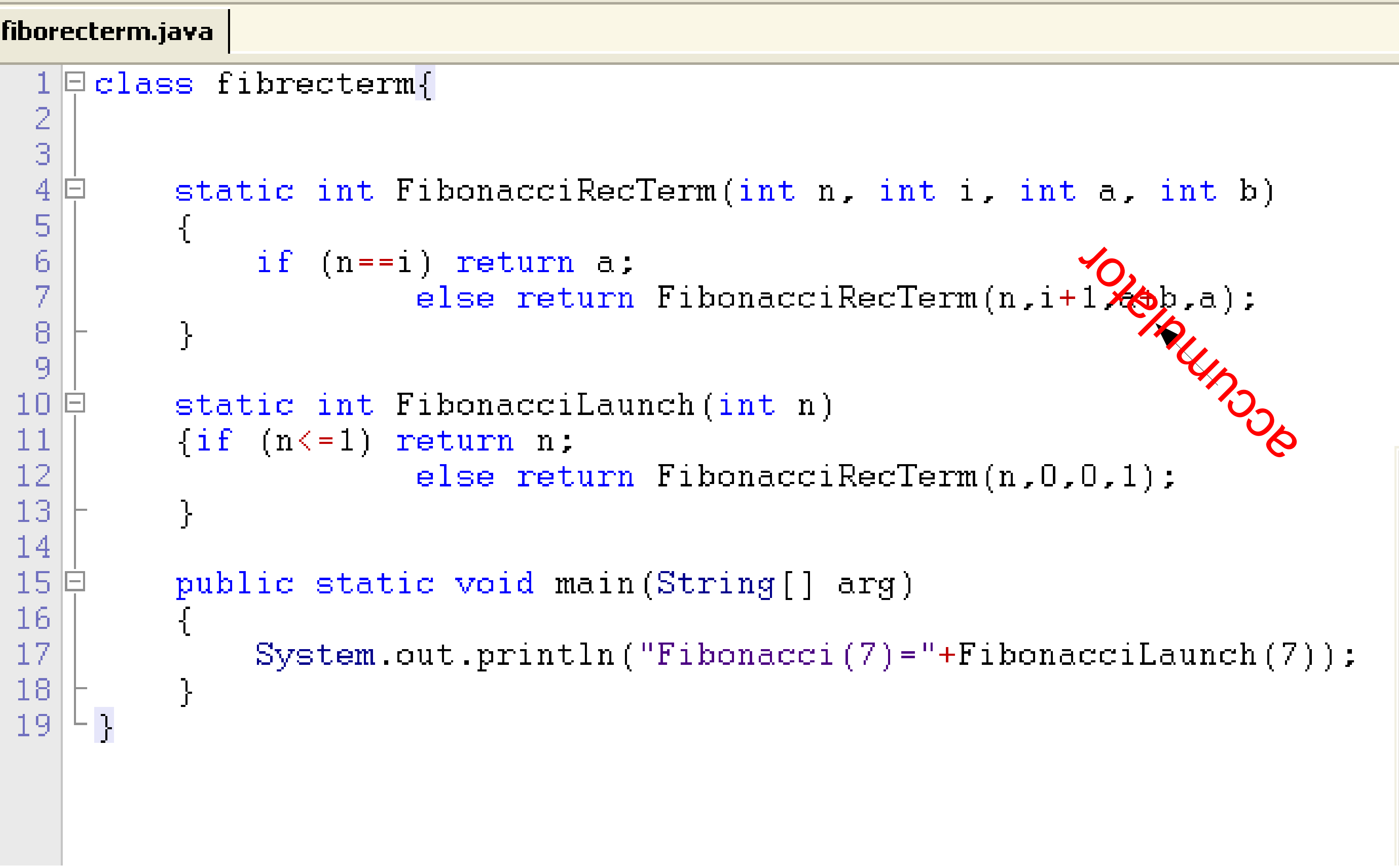
```
Factorial 10!=3628800
Press any key to continu
```

Arguments plays the role of accumulators

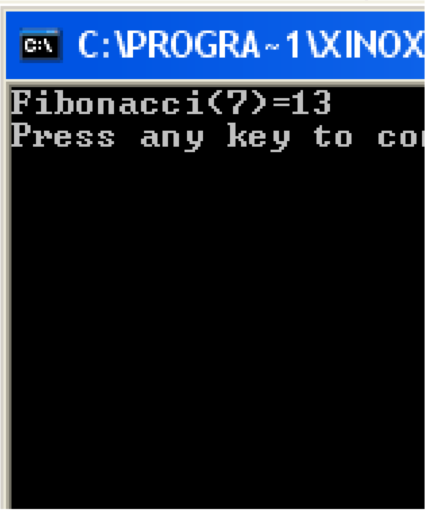
What happens if we call Factorial(100) ?



Terminal Recursion: Revisiting Fibonacci



```
class fibrecterm{


    static int FibonacciRecTerm(int n, int i, int a, int b)
    {
        if (n==i) return a;
                else return FibonacciRecTerm(n,i+1,a+b,a);
    }

    static int FibonacciLaunch(int n)
    {if (n<=1) return n;
                else return FibonacciRecTerm(n,0,0,1);
    }

    public static void main(String[] arg)
    {
        System.out.println("Fibonacci(7)="+FibonacciLaunch(7));
    }
}
```

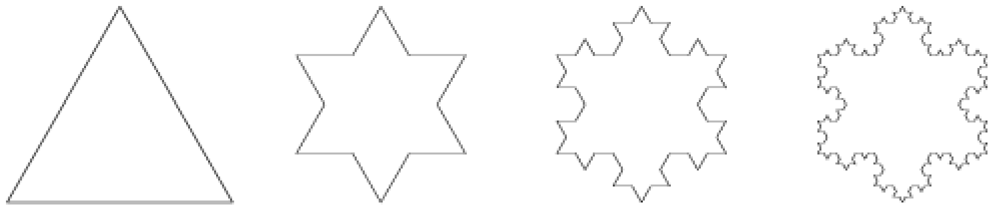


```
Fibonacci(7)=13
Press any key to co
```

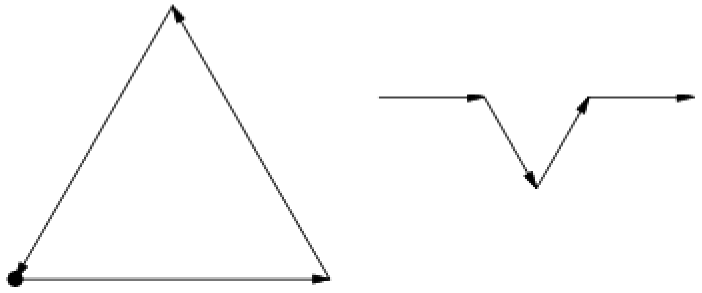
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, ...

Recursivity and Nature

Drawing fractal curves and motifs



Koch's snowflake

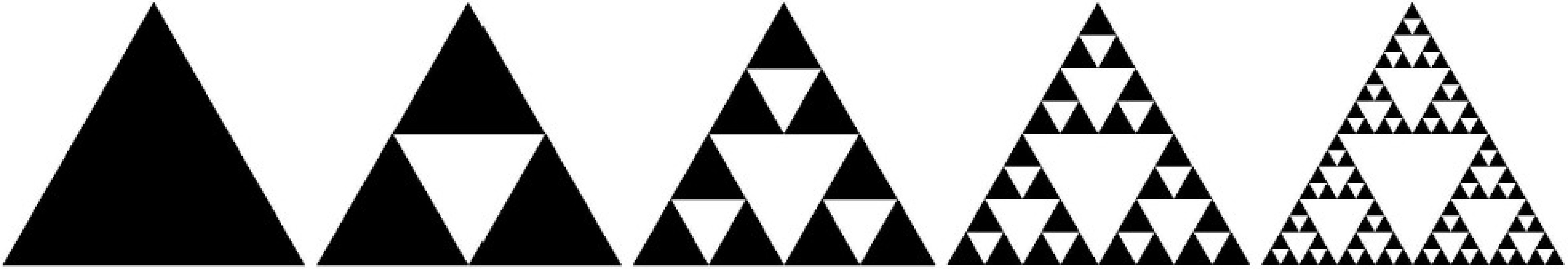


<u>Fractals:</u>

- Patterns that are present at **different scales**
- The curve at stage n is **defined recursively...**
  ....from the curve at stage n-1

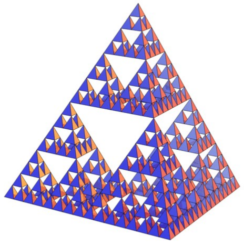
Fractal: Sierpinski motif

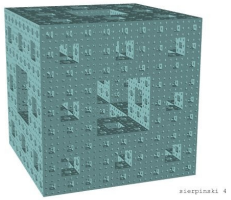
Waclaw Sierpinski
(1882-1969)
Polish mathematician



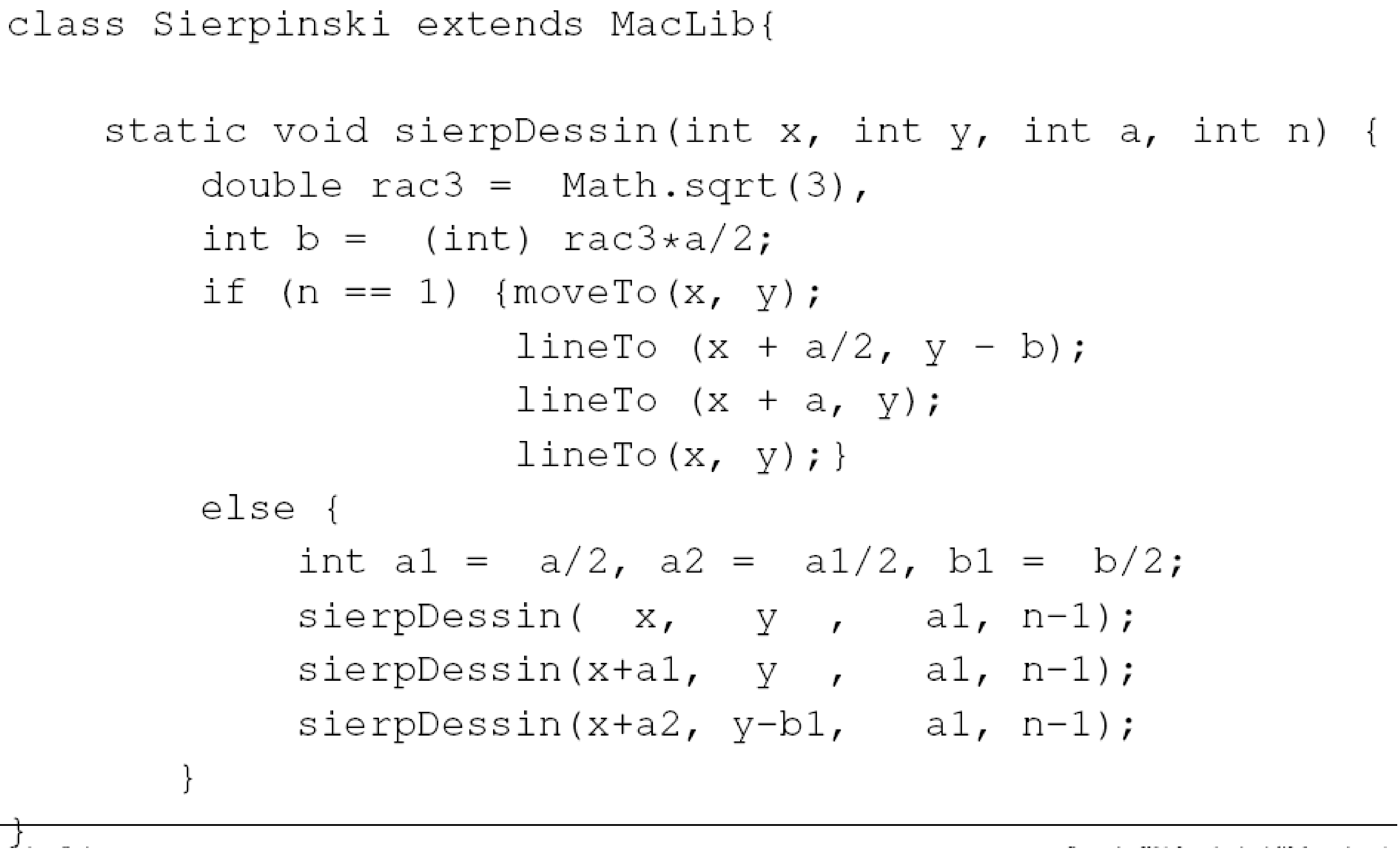
Generation 1 Generation 2 Generation 3 Generation 4 Generation 5

The recursive pattern is given by a simple rewritting rule:
Replace a triangle by 3 triangles defined by the...
midpoints of the edges of the source triangle

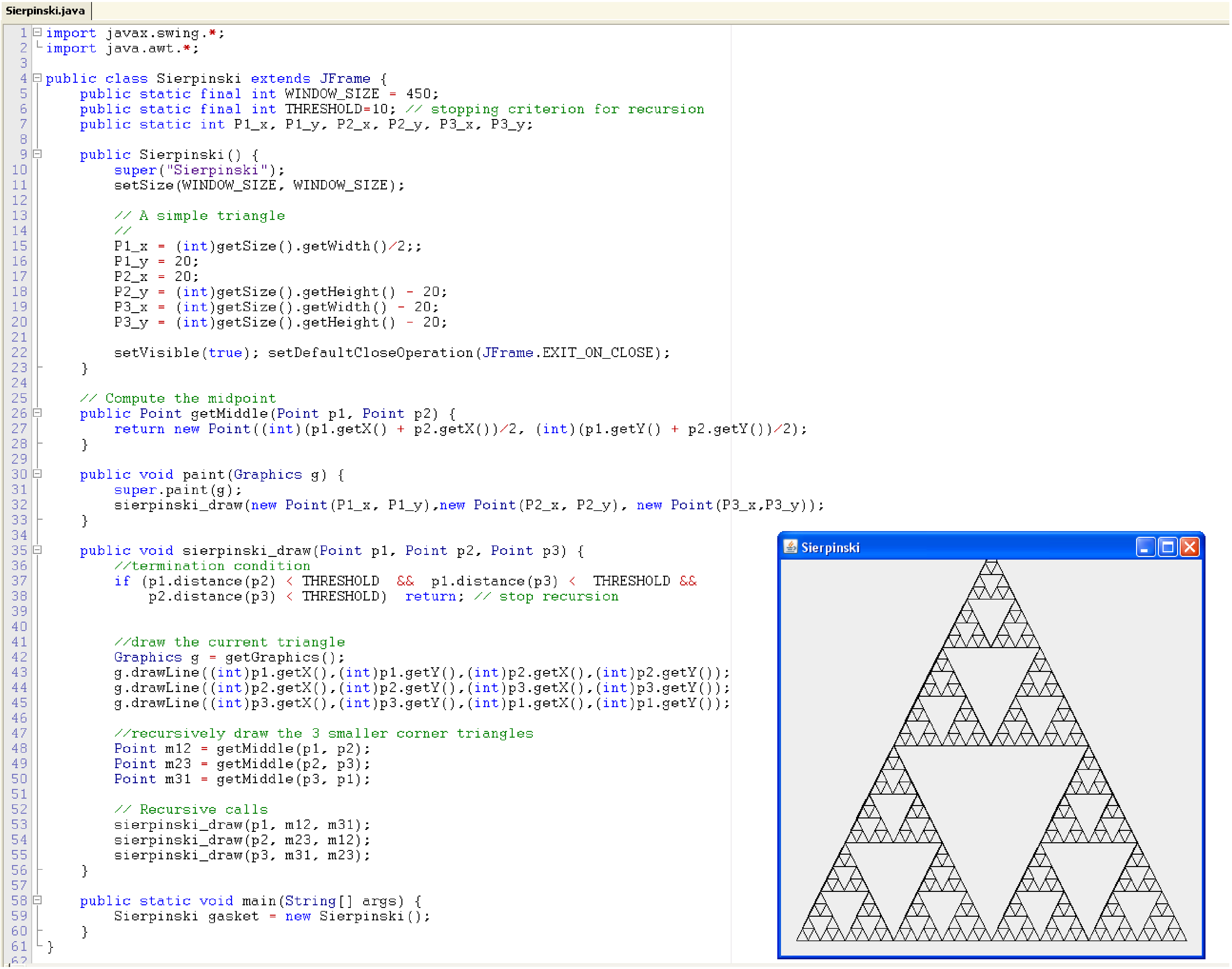




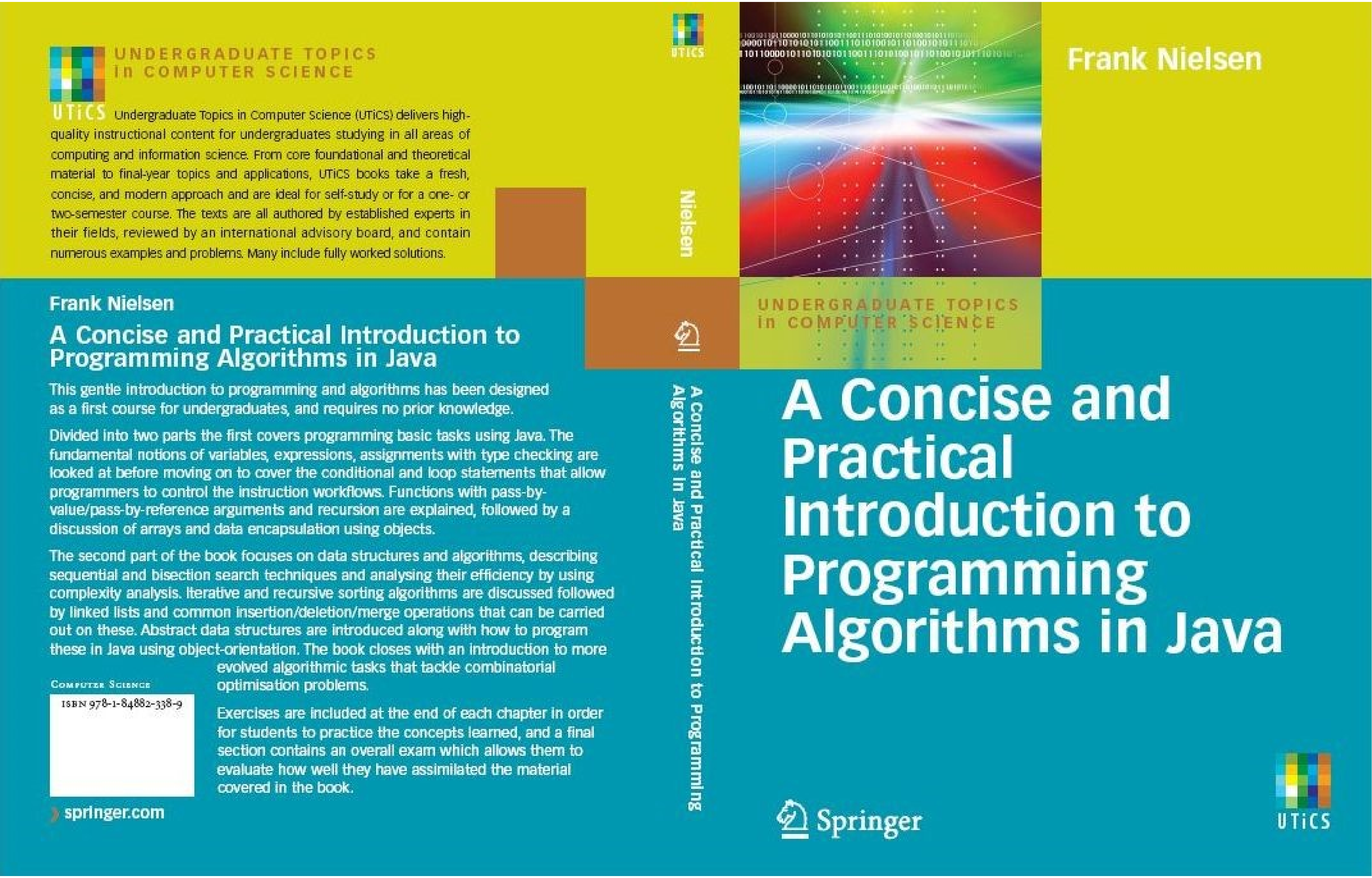
Sierpinski curve (2D pyramid)



```
class Sierpinski extends MacLib{

    static void sierpDessin(int x, int y, int a, int n) {
        double rac3 =  Math.sqrt(3),
        int b =  (int) rac3*a/2;
        if (n == 1) {moveTo(x, y);
                     lineTo (x + a/2, y - b);
                     lineTo (x + a, y);
                     lineTo(x, y);}
        else {
            int a1 =  a/2, a2 =  a1/2, b1 =  b/2;
            sierpDessin(  x,    y  ,   a1, n-1);
            sierpDessin(x+a1,   y  ,   a1, n-1);
            sierpDessin(x+a2, y-b1,    a1, n-1);
       }
}
```



```java
import javax.swing.*;
import java.awt.*;

public class Sierpinski extends JFrame {
    public static final int WINDOW_SIZE = 450;
    public static final int THRESHOLD=10; // stopping criterion for recursion
    public static int P1_x, P1_y, P2_x, P2_y, P3_x, P3_y;

    public Sierpinski() {
        super("Sierpinski");
        setSize(WINDOW_SIZE, WINDOW_SIZE);

        // A simple triangle
        //
        P1_x = (int)getSize().getWidth()/2;;
        P1_y = 20;
        P2_x = 20;
        P2_y = (int)getSize().getHeight() - 20;
        P3_x = (int)getSize().getWidth() - 20;
        P3_y = (int)getSize().getHeight() - 20;

        setVisible(true); setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    // Compute the midpoint
    public Point getMiddle(Point p1, Point p2) {
        return new Point((int)(p1.getX() + p2.getX())/2, (int)(p1.getY() + p2.getY())/2);
    }

    public void paint(Graphics g) {
        super.paint(g);
        sierpinski_draw(new Point(P1_x, P1_y),new Point(P2_x, P2_y), new Point(P3_x,P3_y));
    }

    public void sierpinski_draw(Point p1, Point p2, Point p3) {
        //termination condition
        if (p1.distance(p2) <  THRESHOLD  &&  p1.distance(p3) <  THRESHOLD &&
            p2.distance(p3) < THRESHOLD)  return; // stop recursion


        //draw the current triangle
        Graphics g = getGraphics();
        g.drawLine((int)p1.getX(),(int)p1.getY(),(int)p2.getX(),(int)p2.getY());
        g.drawLine((int)p2.getX(),(int)p2.getY(),(int)p3.getX(),(int)p3.getY());
        g.drawLine((int)p3.getX(),(int)p3.getY(),(int)p1.getX(),(int)p1.getY());

        //recursively draw the 3 smaller corner triangles
        Point m12 = getMiddle(p1, p2);
        Point m23 = getMiddle(p2, p3);
        Point m31 = getMiddle(p3, p1);

        // Recursive calls
        sierpinski_draw(p1, m12, m31);
        sierpinski_draw(p2, m23, m12);
        sierpinski_draw(p3, m31, m23);
    }

    public static void main(String[] args) {
        Sierpinski gasket = new Sierpinski();
    }
}
```



UNDERGRADUATE TOPICS in COMPUTER SCIENCE

UTiCS Undergraduate Topics in Computer Science (UTiCS) delivers high-quality instructional content for undergraduates studying in all areas of computing and information science. From core foundational and theoretical material to final-year topics and applications, UTiCS books take a fresh, concise, and modern approach and are ideal for self-study or for a one- or two-semester course. The texts are all authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems. Many include fully worked solutions.

Frank Nielsen

A Concise and Practical Introduction to Programming Algorithms in Java

This gentle introduction to programming and algorithms has been designed as a first course for undergraduates, and requires no prior knowledge.

Divided into two parts the first covers programming basic tasks using Java. The fundamental notions of variables, expressions, assignments with type checking are looked at before moving on to cover the conditional and loop statements that allow programmers to control the instruction workflows. Functions with pass-by-value/pass-by-reference arguments and recursion are explained, followed by a discussion of arrays and data encapsulation using objects.

The second part of the book focuses on data structures and algorithms, describing sequential and bisection search techniques and analysing their efficiency by using complexity analysis. Iterative and recursive sorting algorithms are discussed followed by linked lists and common insertion/deletion/merge operations that can be carried out on these. Abstract data structures are introduced along with how to program these in Java using object-orientation. The book closes with an introduction to more evolved algorithmic tasks that tackle combinatorial optimisation problems.

Exercises are included at the end of each chapter in order for students to practice the concepts learned, and a final section contains an overall exam which allows them to evaluate how well they have assimilated the material covered in the book.

Computer Science

ISBN 978-1-84882-338-9

springer.com

UTiCS

Nielsen

A Concise and Practical Introduction to Programming Algorithms in Java

Frank Nielsen

UNDERGRADUATE TOPICS in COMPUTER SCIENCE

A Concise and Practical Introduction to Programming Algorithms in Java

Springer

UTiCS