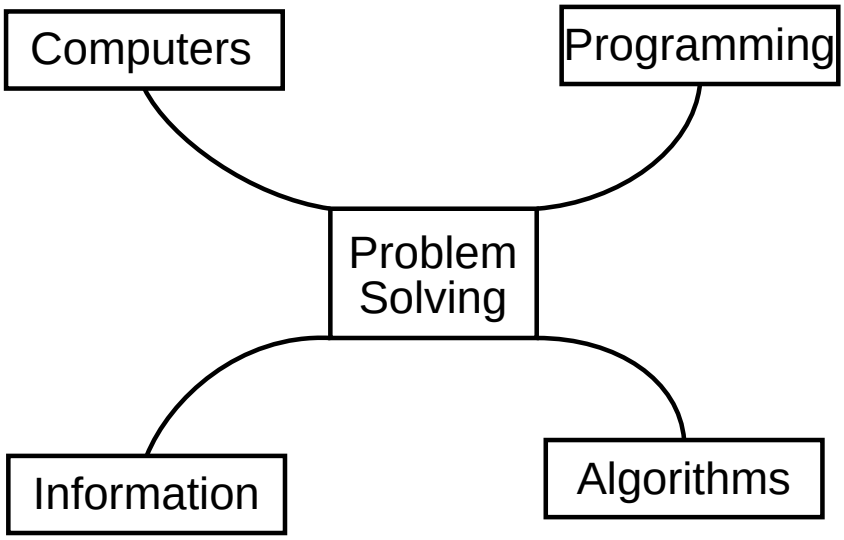
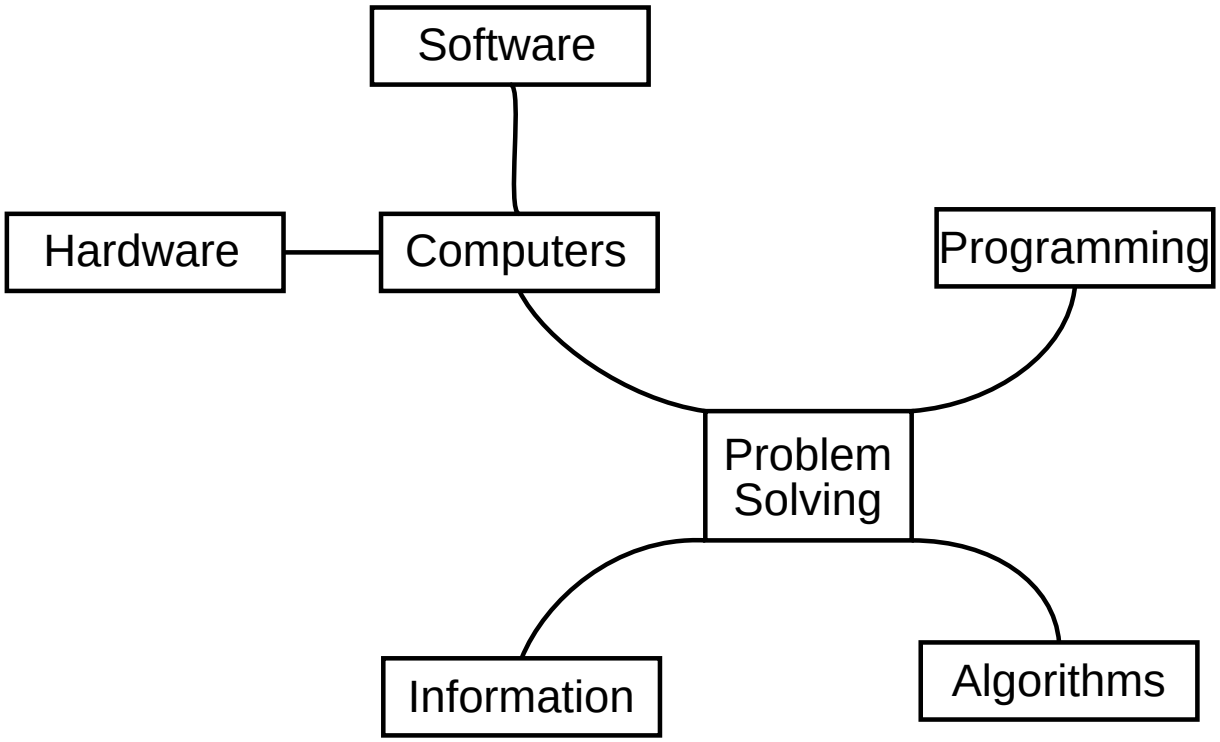
Reminder

- Midterm today at 6:00pm at MAASS 112

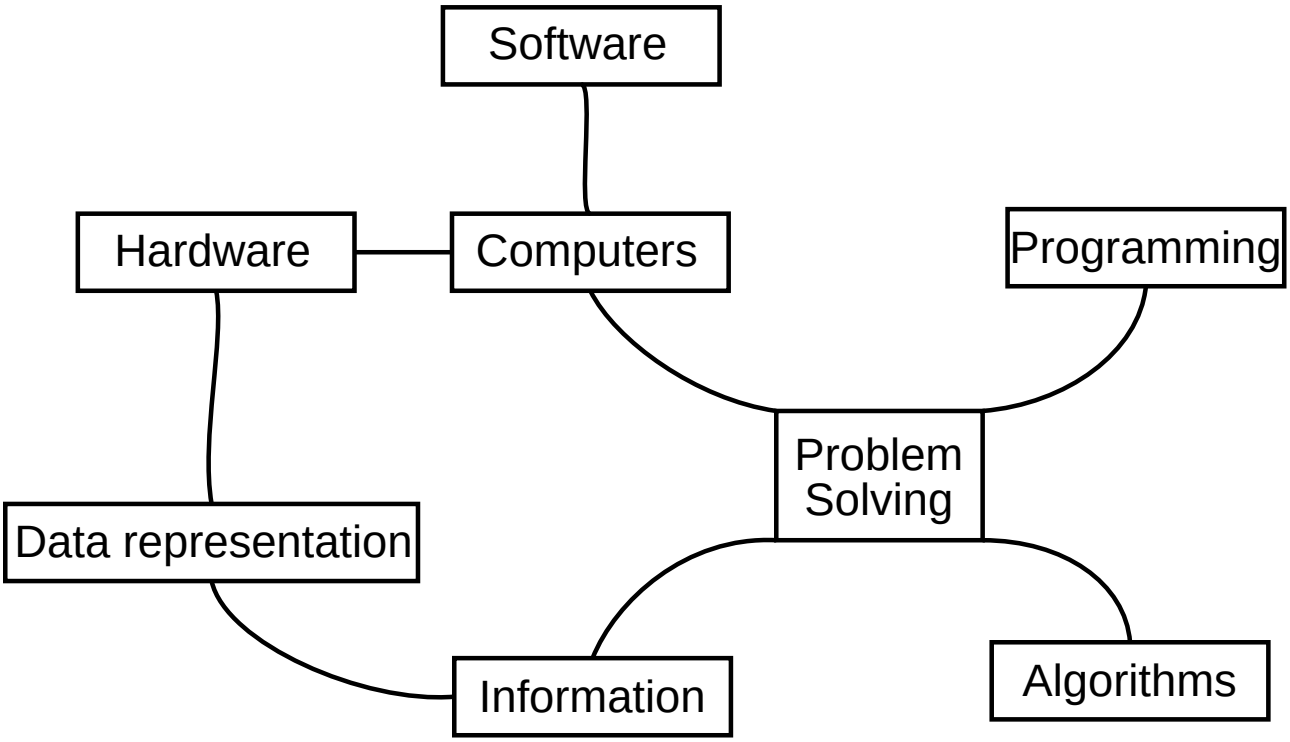
Review


Computers
Programming
Problem Solving
Information
Algorithms

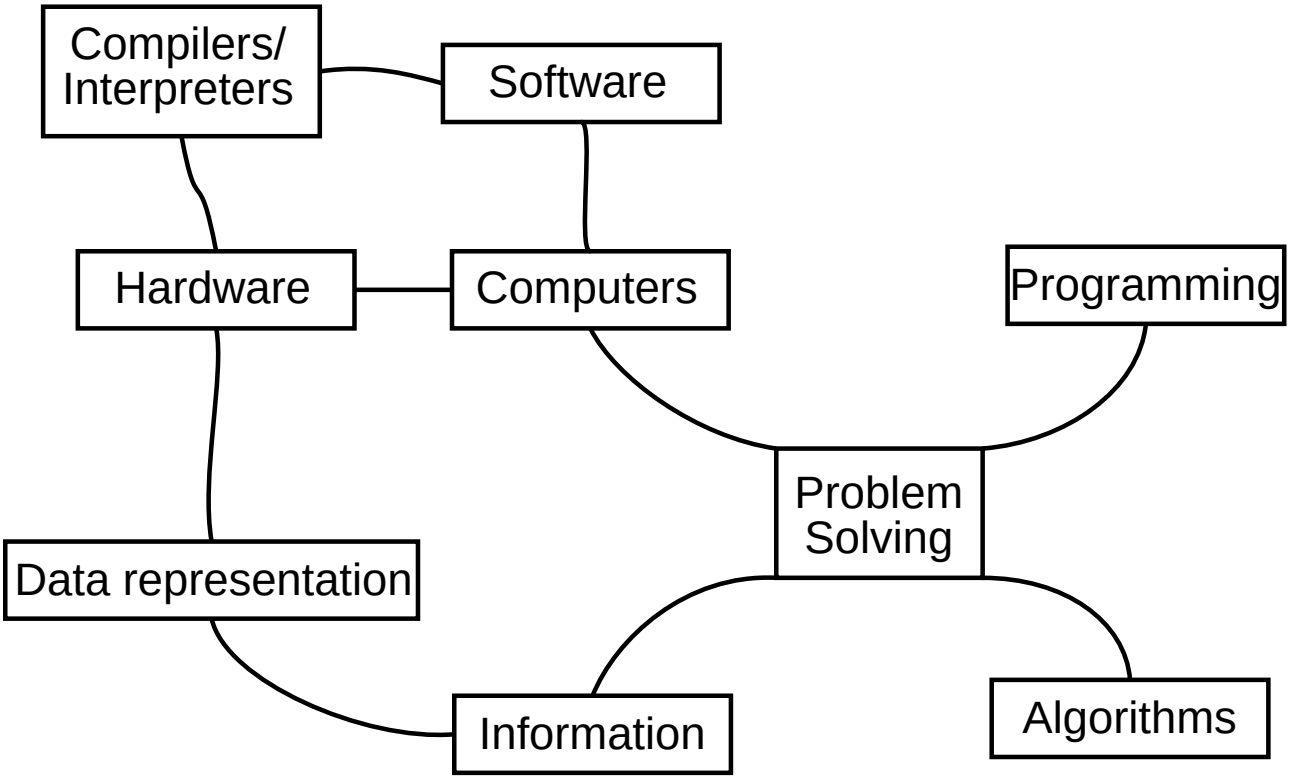
Review


Software
Hardware
Computers
Programming
Problem Solving
Information
Algorithms

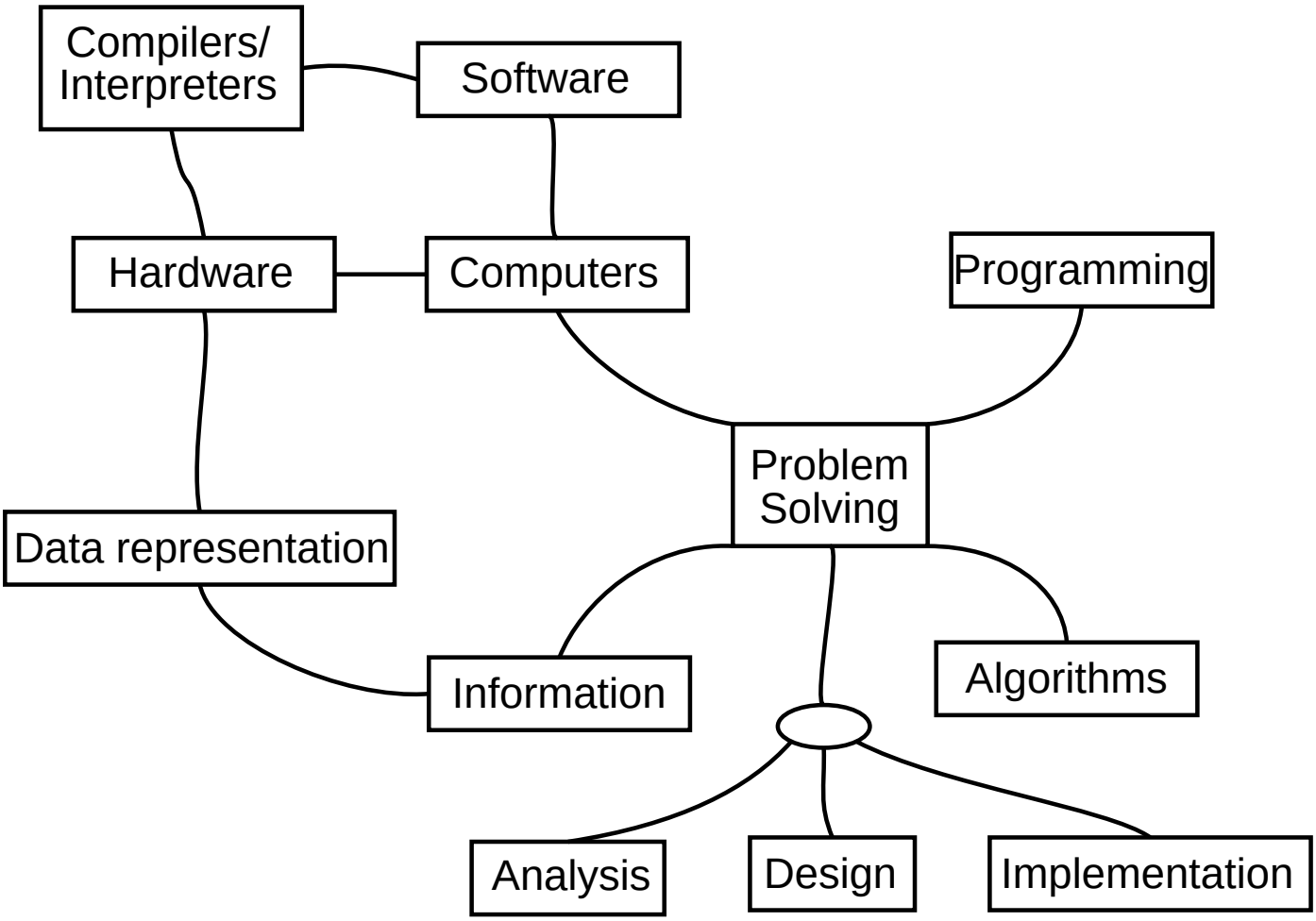
Review


Software
Hardware
Computers
Programming
Problem Solving
Data representation
Information
Algorithms

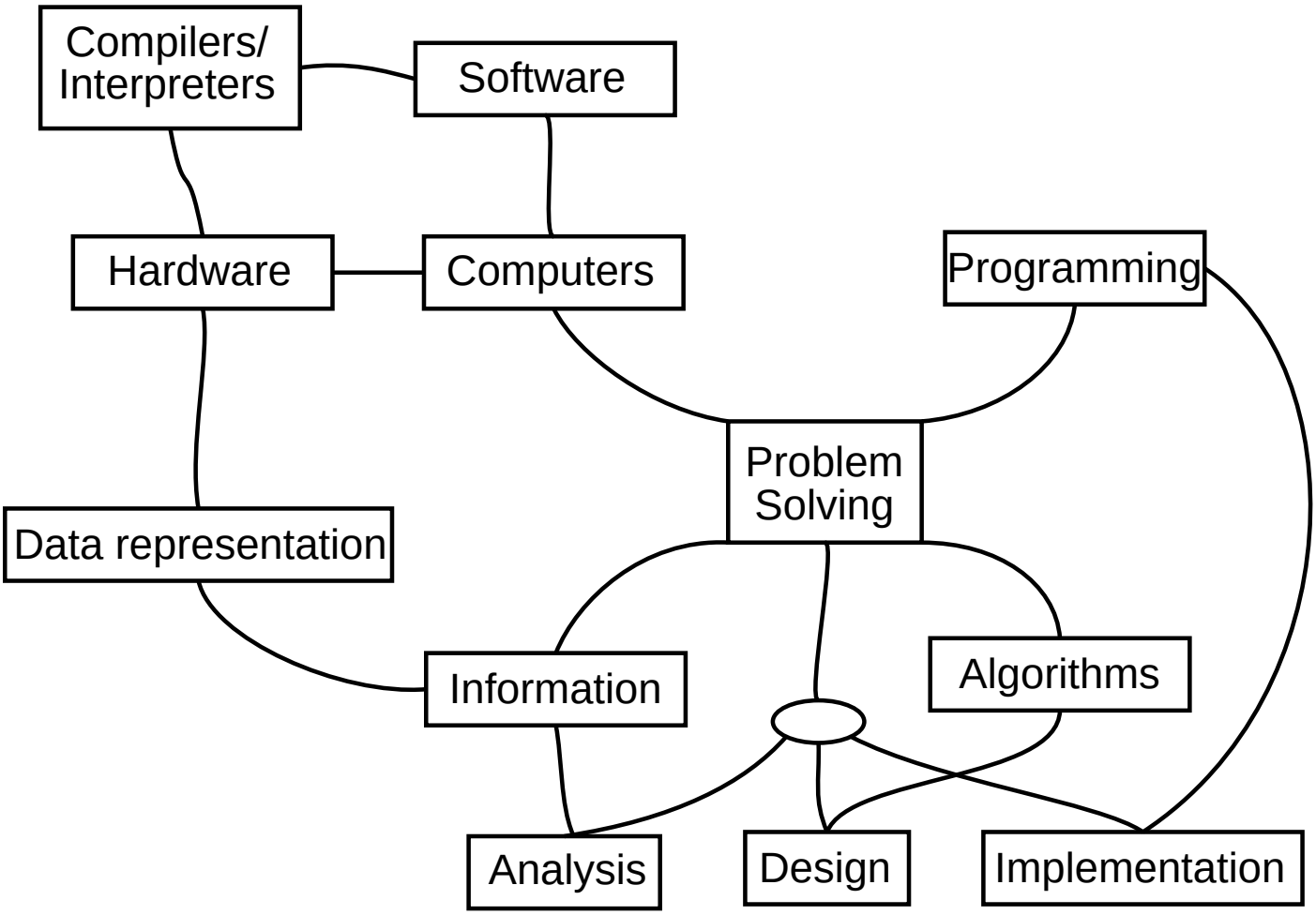
Review


Compilers/
Interpreters
Software
Hardware
Computers
Programming
Problem
Solving
Data representation
Information
Algorithms

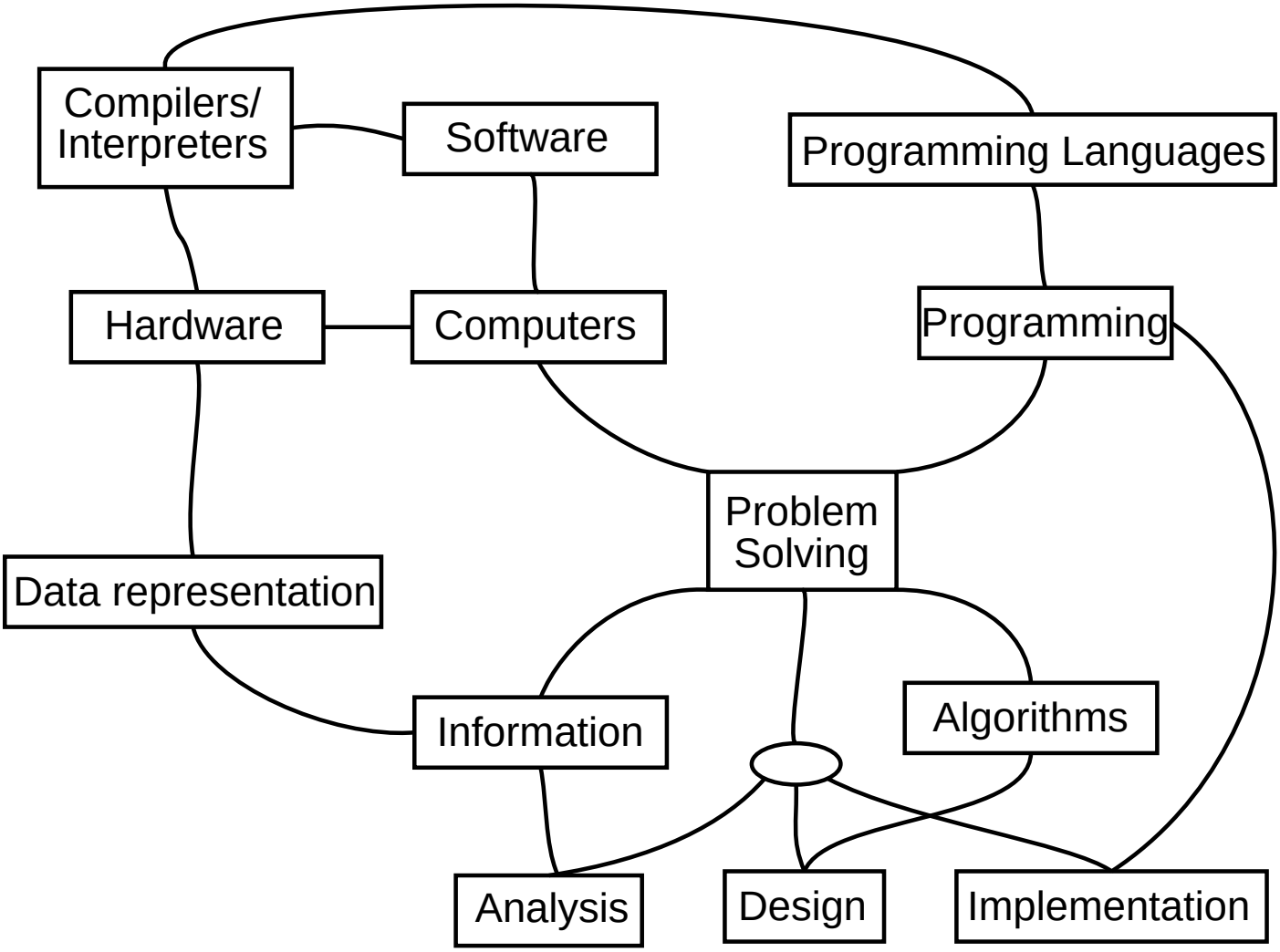
Review


Compilers/
Interpreters
Software
Hardware
Computers
Programming
Problem
Solving
Data representation
Information
Algorithms
Analysis
Design
Implementation

Review


Compilers/
Interpreters
Software
Hardware
Computers
Programming
Problem
Solving
Data representation
Information
Algorithms
Analysis
Design
Implementation

Review


Compilers/
Interpreters
Software
Programming Languages
Hardware
Computers
Programming
Problem
Solving
Data representation
Information
Algorithms
Analysis
Design
Implementation

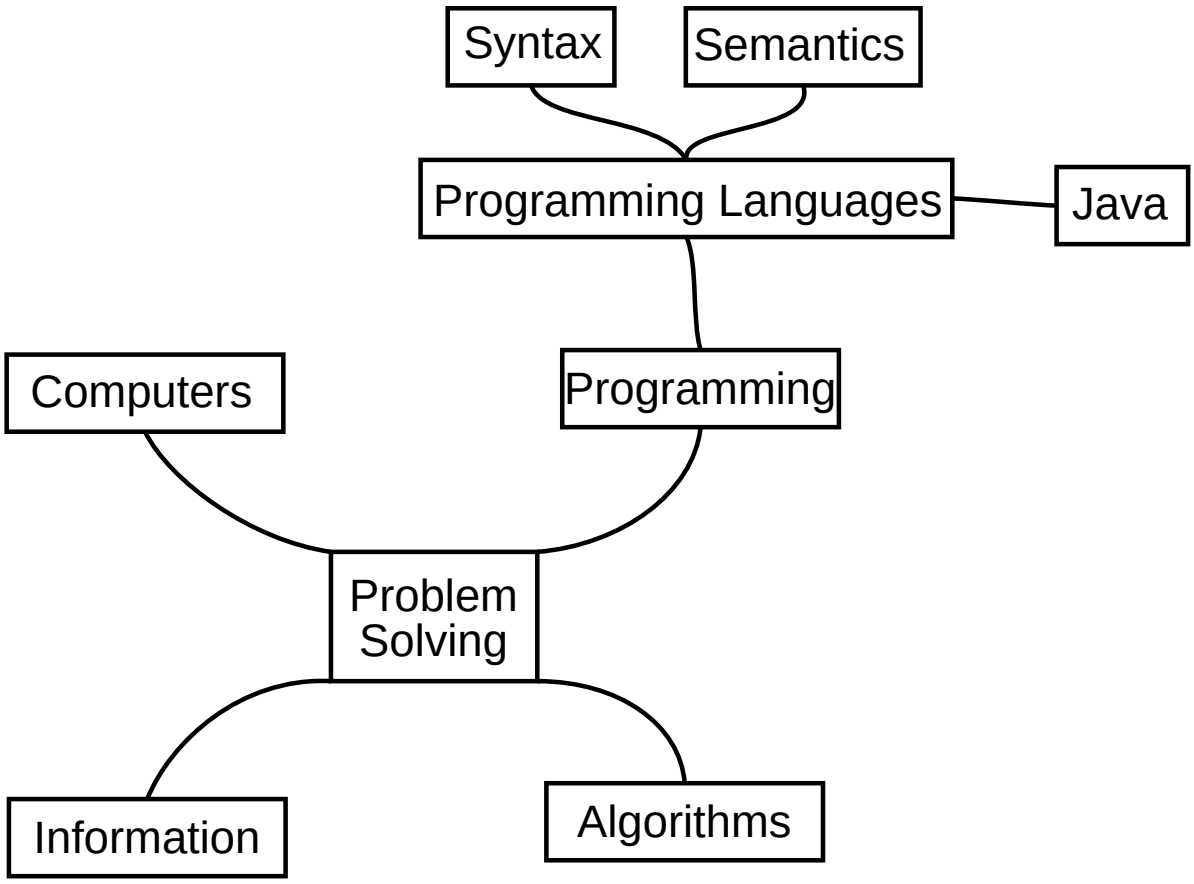
Problem solving

- Clear statement of the problem
- Analysis (of the problem)
- Design
- Implementation
- Testing / Verification
- Maintenance

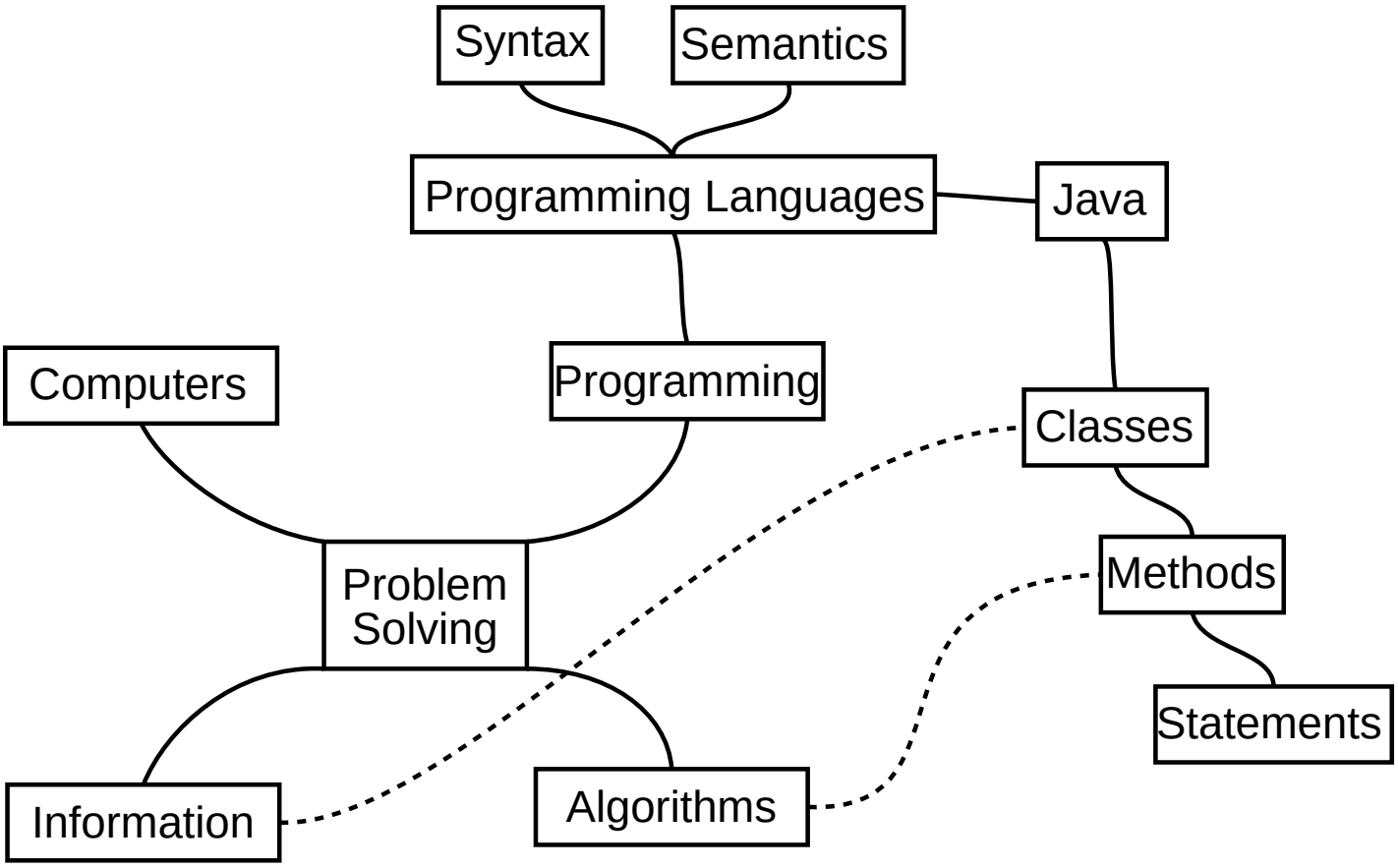
Analysis

- Goal: to obtain a precise understanding the problem
- Things to do in analysis:
  - Determine inputs and outputs
  - Determine general and specific requirements
  - Make or obtain precise definitions of concepts involved
  - Determine the relevant information to the problem
  - Determine the relationship between different elements or pieces of information of the problem
  - Make explicit any relevant assumptions

Review


Syntax
Semantics
Programming Languages
Java
Computers
Programming
Problem Solving
Information
Algorithms

Review


Syntax
Semantics
Programming Languages
Java
Computers
Programming
Classes
Problem Solving
Methods
Statements
Information
Algorithms

Statements

- Variable declarations
- Assignment
- Conditionals
- Loops
- Method calls

Statements

- Variables
  - A variable is a memory cell with a name (identifier)
  - A variable can change its value
  - A variable has a *data type*
  - Variable declaration

```
type identifier;
type identifier = expression;
type id1, id2, id3;
type id1 = e1, id2 = e2, id3 = e3;
```

  - The type of a variable can be
    - Primitive (e.g. int, boolean, etc.)
    - User-defined (i.e. a class)

Statements

- Assignment
  - Changes the value of a variable

    variable = *expression*;

  - Expressions:
    * arithmetic: int type
    * boolean
    * string

Statements

- Conditionals: making decisions
  - if

```
if (condition)
  statement_or_block;
```

  - if-else

```
if (condition)
  statement_or_block1;
else
  statement_or_block2;
```

  - switch

```
switch(int_or_char_expr) {
  case int_or_char_expr1: statements1;
  case int_or_char_expr2: statements2;
  ...
  default: statementsn;
}
```

Statements

- Loops: Repetition
  - while

```
while (condition)
  statement_or_block;
```

  - do-while

```
do
  statement_or_block
while(condition);
```

  - for

```
for (init; condition; statement)
  statement_or_block;
```

Statements

- Method calls
  - normal

    objectreference.*methodname*(*arg1*, *arg2*, ..., *argn*

  - static

    classname.*methodname*(*arg1*, *arg2*, ..., *argn*)

- Messages to self

  methodname(*arg1*, *arg2*, ..., *argn*)
  this.*methodname*(*arg1*, *arg2*, ..., *argn*)

Control-flow

- Statements are executed sequentially
- The order in which statements appear matters

```
int x;
x = 8;
x = x + 3;
if (x >= 10) {
  System.out.println(“OK”);
}
else {
  System.out.println(“Error”);
}
```

Control-flow

- Statements are executed sequentially
- The order in which statements appear matters

```
int x;
x = 8;
if (x >= 10) {
  System.out.println(“OK”);
}
else {
  System.out.println(“Error”);
}
x = x + 3;
```

Control-flow

- Loops
- Termination: a loop may fail to terminate

```
int i;
i = 0;
while (i < 10) {
  System.out.print(i);
}
```

- To terminate, the body of the loop must have some "advance" statements, i.e. statements which will eventually make the condition false.

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

Classes and objects

- Objects are composite data that can react to messages
- Classes define the structure and behaviour of objects
- Objects are not classes
  - A class is a data type
  - An object is an instance of a class

Classes and objects

- Defining classes
  - Attributes (instance variables): characteristics of the objects
  - Methods: how objects react to messages
- Using classes
  - Creating objects
  - Accessing attributes
  - Invoking (calling) methods: sending messages

Classes and objects

- Defining classes:

```
public class ClassName {
  // Attribute definitions
  // Method definitions
}
```

- Defining attributes:

```
type identifier;
```

Note: different classes can have attributes with the same name

Classes and objects

- Defining methods:
  - normal

```
type name(type1 p1, type2 p2, ..., typen pn)
{
  list_of_statements;
}
```

  - static

```
static type name(type1 p1, type2 p2, ..., typen
{
  list_of_statements;
}
```

Note: different classes can have methods with the same name

Classes and objects

- Using classes:
  - Creating objects:
    type objectreference;
    objectreference = new *type*(*args*);
    where *type* is a class.

- Accessing attributes:

  objectreference.*attributename*

  only if the attribute is defined in *objectreference*'s class.

- Calling methods:

  objectreference.*methodname*(*arg1*,*arg2*,...,*argn*)

  only if the method is defined in *objectreference*'s class, and the types of the arguments are the same as the types of the parameters.

Methods calls

- If class A is defined as

```
class A {
  void m() { ... }
}
```

- and a variable x has a reference to an A object:

```
A x;
x = new A();
```

- then method m can be applied to x

```
x.m();
```

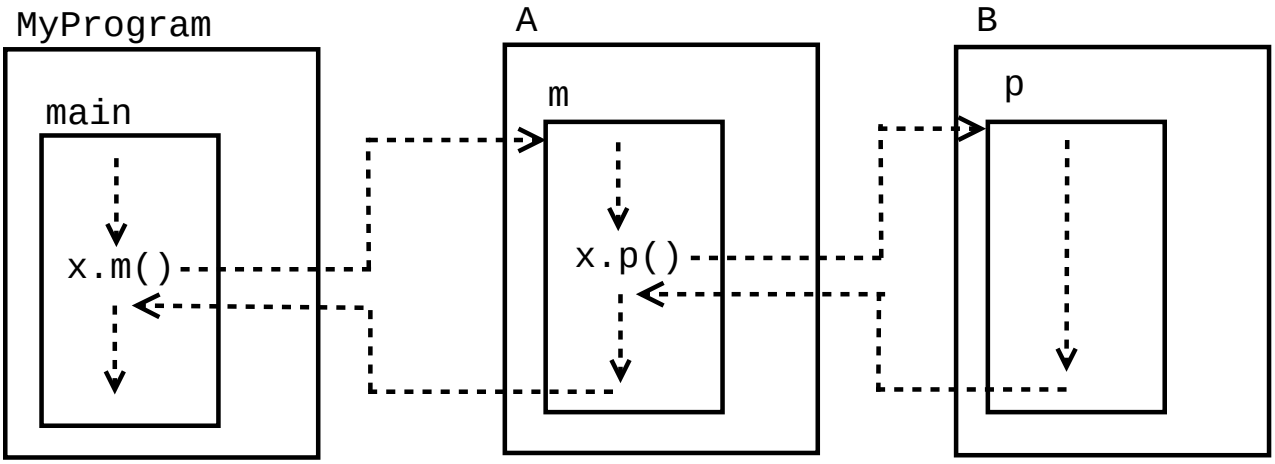
- But if method p is not defined in A, it cannot be applied to x

```
x.p(); // Error
```

Method calls

- Control flow
- Frames and parameter passing
- Access to variables
  - Parameters
  - Local variables
  - Object attributes
- Static methods
- Static variables
- Recursive methods

Methods calls: control flow

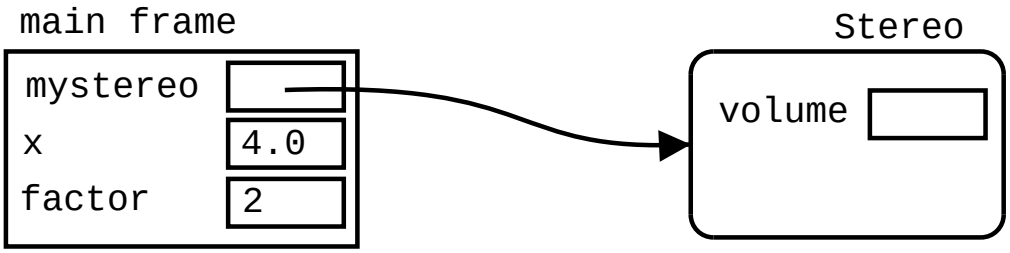


Methods calls: parameter passing

```
public class Stereo {
    double volume;
    void set_volume(double v)
    {
        volume = v;
    }
    double get_volume()
    {
        return volume;
    }
}
public class SoundSystem {
    public static void main(String[] args)
    {
        Stereo mystereo = new Stereo();
        double x, factor = 2;
        System.out.println("Testing...");
        x = 4.0;
        mystereo.set_volume( x * factor );
        System.out.println(mystereo.get_volume());
    }
}
```

Method invocation: Memory structure

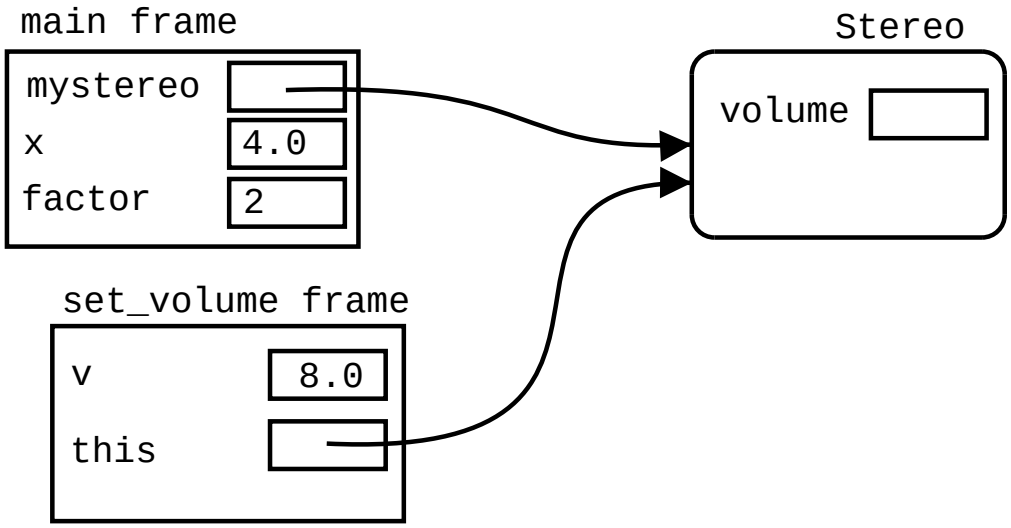
Before calling `mystereo.set_volume(x*factor)`



First its arguments (`x*factor`) are evaluated:
Evaluating `x*factor` in the main frame results in `8.0`

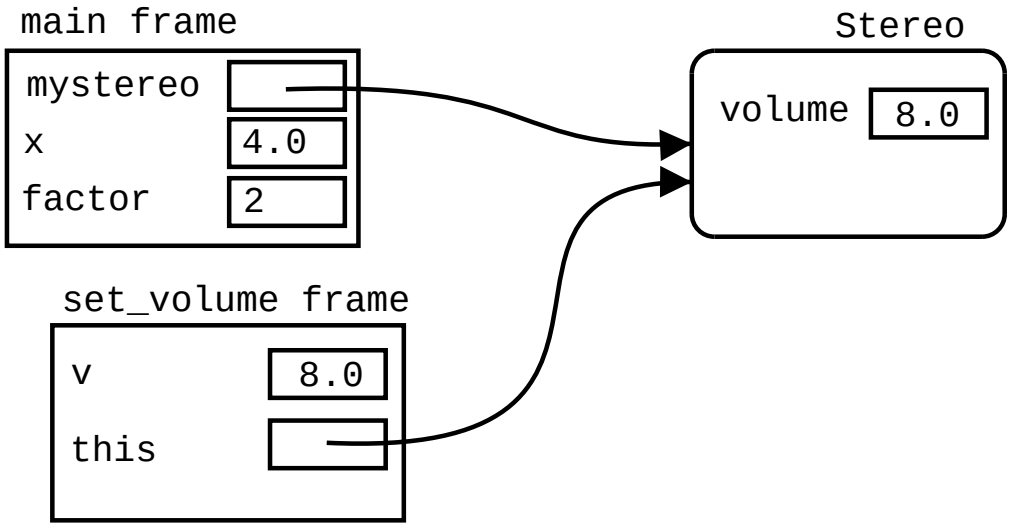
Method invocation: Memory structure

A frame for set_volume is created, and
the argument is assigned to the parameter: `v = 8.0;`



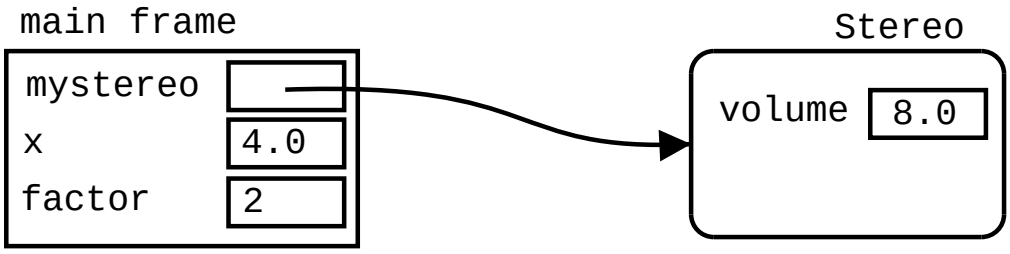
Method invocation: Memory structure

The current method (main) is suspended, and
the body of the called method (set_volume) is executed
in the context of the current frame (the set_volume frame):



Method invocation: Memory structure

Finally the called method frame is discarded, and computation of the calling method (main) is resumed in the instruction immediately after the method call.



Static methods

- Static methods have no direct effect on objects
- When calling a static method, the frame does not have a "this" reference
- Therefore they cannot access any object attributes
- They can access only:
  - Parameters
  - Local variables
  - static variables (shared between all objects in the class)
- Non-static methods can access both static and non-static methods and variables
- Static methods cannot access non-static methods and non-static instance variables

Objects as first class values

- Objects can be passed as parameters and returned as values

```
public class Rabbit {
  void jump() { ... }
}
public class Cage {
  void shake(Rabbit a)
  {
    a.jump();
  }
  Rabbit create()
  {
    return new Rabbit();
  }
}
```

...elsewhere...

```
Rabbit bugs = new Rabbit();
Cage c = new Cage();
c.shake(bugs);
Rabbit wester = c.create();
```

Objects as first class values

- Objects can be attibutes of other objects

```
public class Rabbit {
  void jump() { ... }
}
public class Cage {
  Rabbit my_rabbit;
  void put(Rabbit a)
  {
    my_rabbit = a;
  }
  Rabbit get()
  {
    return my_rabbit;
  }
}
```

...elsewhere...

```
Rabbit bugs = new Rabbit();
Cage c = new Cage();
c.put(bugs);
Rabbit wester = c.get();
```

Methods: divide and conquer

If a problem is too complex, then its solution should be done by dividing the problem into smaller subproblems, and solving each subproblem in a separate method. The solution to the entire problem is then achieved by combining the solutions of the subproblems (by calling the necessary methods.)

Methods: *reusable* abstractions

- A method can be reused in different contexts
- Calling a method is "the same" as substituting its body in place of its call (replacing the parameters by the actual arguments,) but
- If we define a method, we can simply call it from more than one context without having to do copy and paste.

Methods: reusable abstractions

Determining whether n is a prime number or not:

```
boolean result;
int i;

result = true;
i = 2;
while (i < n && result) {
  if (n % i == 0) {
    result = false;
  }
  i++;
}
```

Methods: reusable abstractions

```
public class MyMathProcedures {
  static void print_primes(int m)
  {
    boolean result;
    int n;

    n = 1;
    while (n <= m) {

      // Find out if n is prime...
      if (result)
        System.out.println(n);
      n++;
    }
  }
}
```

Methods: reusable abstractions

```
public class MyMathProcedures {
  static void print_primes(int m)
  {
    boolean result;
    int i, n;

    n = 1;
    while (n <= m) {
      result = true;
      i = 2;
      while (i < n && result) {
        if (n % i == 0) {
          result = false;
        }
        i++;
      }
      if (result)
        System.out.println(n);
      n++;
    }
  }
}
```

Methods: reusable abstractions

```
public class MyMathProcedures {
  static boolean is_prime(int n)
  {
    boolean result;
    int i;

    result = true;
    i = 2;
    while (i < n && result) {
      if (n % i == 0) {
        result = false;
      }
      i++;
    }
    return result;
  }
  //... rest of the class
}
```

Methods: reusable abstractions

```
public class MyMathProcedures {
  static void print_primes(int m)
  {
    boolean result;
    int i, n;

    n = 1;
    while (n <= m) {
      result = true;
      i = 2;
      while (i < n && result) {
        if (n % i == 0) {
          result = false;
        }
        i++;
      }
      if (result)
        System.out.println(n);
      n++;
    }
  }
}
```

Methods: reusable abstractions

```
public class MyMathProcedures {
  static boolean is_prime(int n) { ... }

  static void print_primes(int m)
  {
    boolean result;
    int n;

    n = 1;
    while (n <= m) {
      result = is_prime(n);
      if (result)
        System.out.println(n);
      n++;
    }
  }
}
```

Methods: reusable abstractions

```
public class MyMathProcedures {
  static boolean is_prime(int n) { ... }

  static void print_primes(int m)
  {
    int n;

    n = 1;
    while (n <= m) {
      if (is_prime(n))
        System.out.println(n);
      n++;
    }
  }
}
```

Methods: reusable abstractions

Problem: given three numbers, determine whether all of them are prime or their sum is prime

Methods: reusable abstractions

```
public class MyMathProcedures {
  static boolean is_prime(int n) { ... }

  static void threenumbers(int a, int b, int c)
  {
    if (is_prime(a) && is_prime(b) && is_prime(c)
       || is_prime(a+b+c)) {
      return true;
    }
    return false;
  }
}
```

Methods: reusable abstractions

```
public class MyMathProcedures {
  static boolean is_prime(int n) { ... }

  static void threenumbers(int a, int b, int c)
  {
    return (is_prime(a) && is_prime(b) && is_prime(
         || is_prime(a+b+c));
  }
}
```

Methods: reusable abstractions

```
public class MyMathProcedures {
  static boolean is_prime(int n) { ... }

  static void threenumbers(int a, int b, int c)
  {
    boolean result1, result2, result3, result4;
    int i;

    result1 = true;
    i = 2;
    while (i < a && result1) {
      if (a % i == 0) {
        result1 = false;
      }
      i++;
    }
    result2 = true;
    i = 2;
    while (i < b && result2) {
      if (b % i == 0) {
        result2 = false;
```

```
        }
        i++;
      }
      result3 = true;
      i = 2;
      while (i < c && result3) {
        if (c % i == 0) {
          result3 = false;
        }
        i++;
      }
      result4 = true;
      i = 2;
      while (i < a+b+c && result4) {
        if ((a+b+c) % i == 0) {
          result4 = false;
        }
        i++;
      }
      return result1 && result2 && result3 || result
    }
  }
```

Methods: reusable abstractions

```
public class MyMathProcedures {
  static boolean is_prime(int n)
  {
    boolean result;
    int i;

    result = true;
    i = 2;
    while (i < Math.sqrt(n) && result) {
      if (n % i == 0) {
        result = false;
      }
      i++;
    }
    return result;
  }
  //... rest of the class
}
```

Recursion

- A recursive method is a method that calls itself (directly or indirectly.)
- A recursive definition is a definition of something in terms of itself
- Some recursive definitions don't make sense, (e.g. from Webster's: growl: to utter a growl), but others do
- For example:
  - A *list of numbers* is either:
    * A single number, or
    * A number followed by a list of numbers.
  - For example:
    * 5 is a list of numbers
    * 7, 5 is a list of numbers (because 5 is a list)
    * 6, 7, 5 is a list of numbers (because 7, 5 is a list)
    * 8, 6, 7, 5 is a list of numbers (because 6, 7, 5 is a list)

Recursive functions

- Factorial: the factorial of a natural number n, written $n!$ is the multiplication of the first n positive integers, i.e.

$$n! = 1 \cdot 2 \cdot 3 \cdot ... \cdot (n-2) \cdot (n-1) \cdot n \qquad (1)$$

But note that

$$1 \cdot 2 \cdot 3 \cdot ... \cdot (n-2) \cdot (n-1) = (n-1)! \qquad (2)$$

So by (1) and (2) we get

$$n! = (n-1)! \cdot n \qquad (3)$$

But we have to assume a "base case", by defining

$$0! = 1 \qquad (4)$$

Recursive functions (contd.)

Hence, (3) and (4) together gives us an alternative, and recursive definition of (1):

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \cdot n & \text{otherwise} \end{cases}$$

This can be implemented as a static recursive method:

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n-1)*n;
}
```

Execution of recursive methods

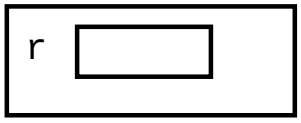
Consider the following client for this factorial function:

```
int r;
r = factorial(4);
```

Its execution proceeds as follows:

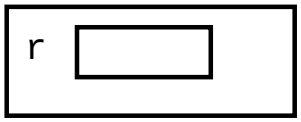
This is executed in some frame:

Some frame

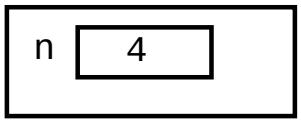


When we call `factorial(4);` a new frame for the method is created:

Some frame



factorial frame

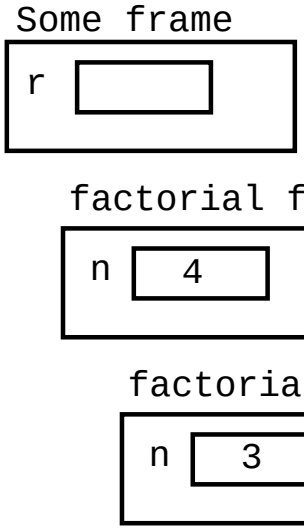


We execute the body of factorial; n is not 0 so we execute

```
return factorial(n-1)*n;
```

which in this frame is the same as

```
return factorial(4-1)*4;
```



pending computation:

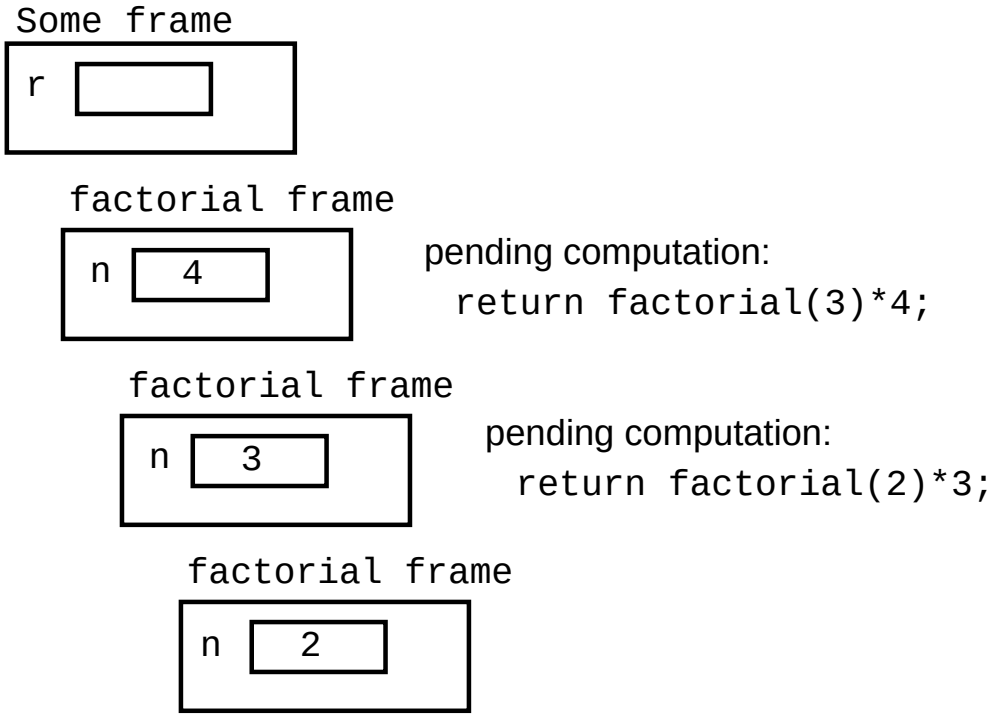
```
return factorial(3)*4;
```

Again, we execute the body of factorial;
again, n is not 0 so we execute

```
return factorial(n-1)*n;
```

which in this frame is the same as

```
return factorial(3-1)*3;
```

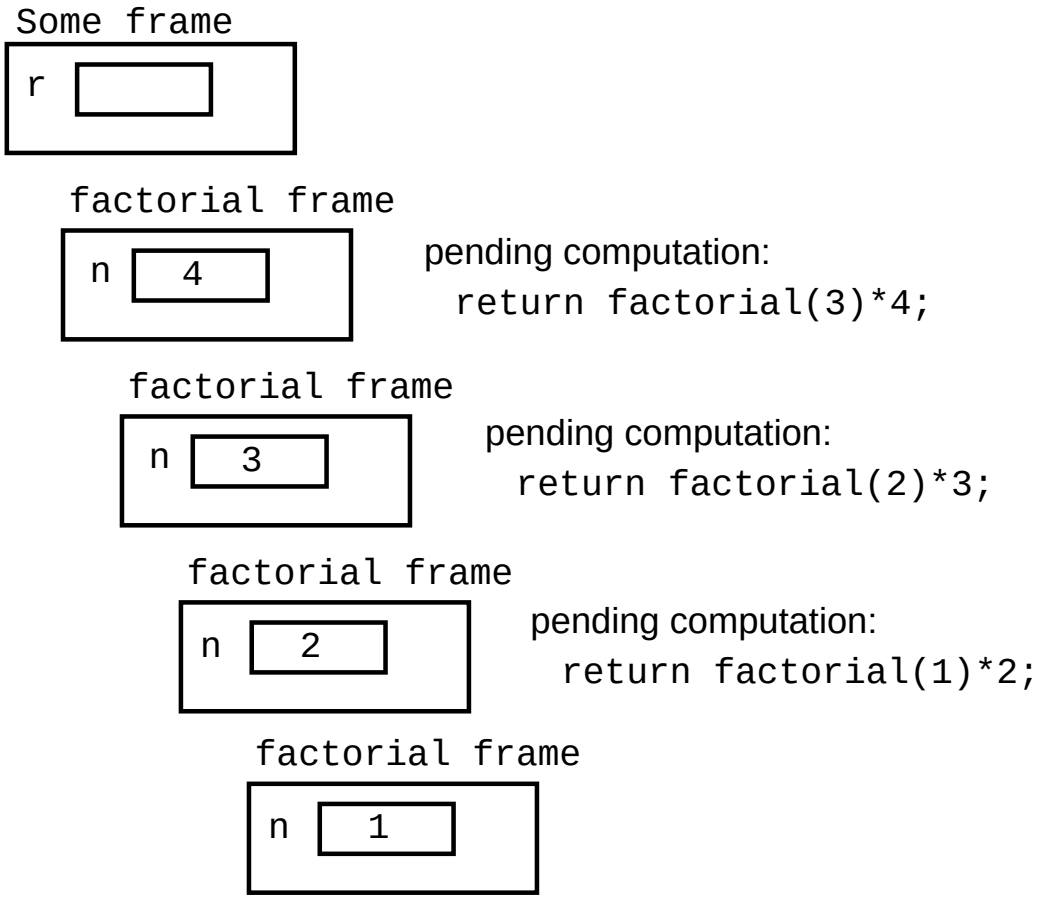


Again, we execute the body of factorial;
again, n is not 0 so we execute

```
return factorial(n-1)*n;
```

which in this frame is the same as

```
return factorial(2-1)*2;
```

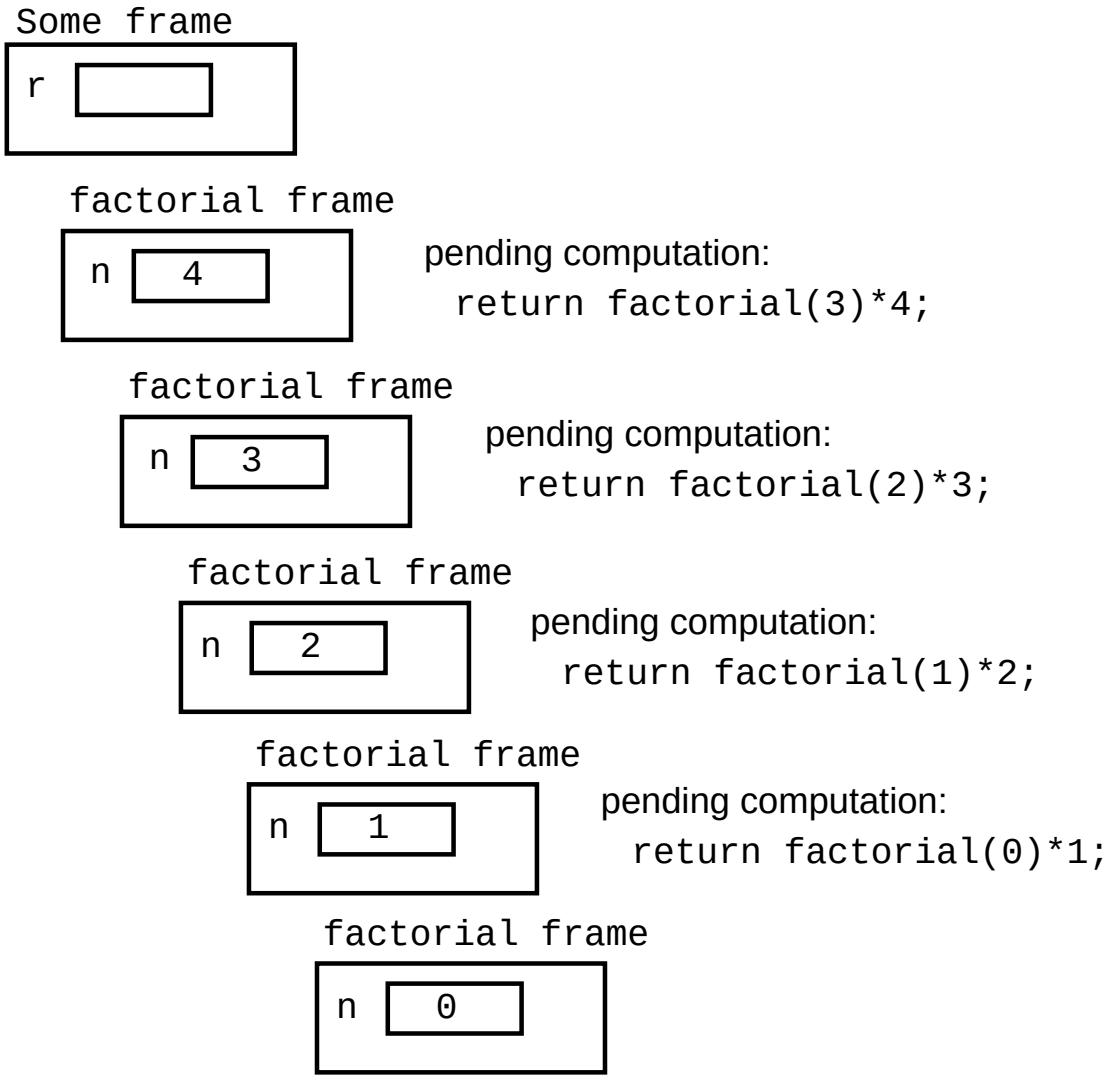


Again, we execute the body of factorial;
again, n is not 0 so we execute

```
return factorial(n-1)*n;
```

which in this frame is the same as

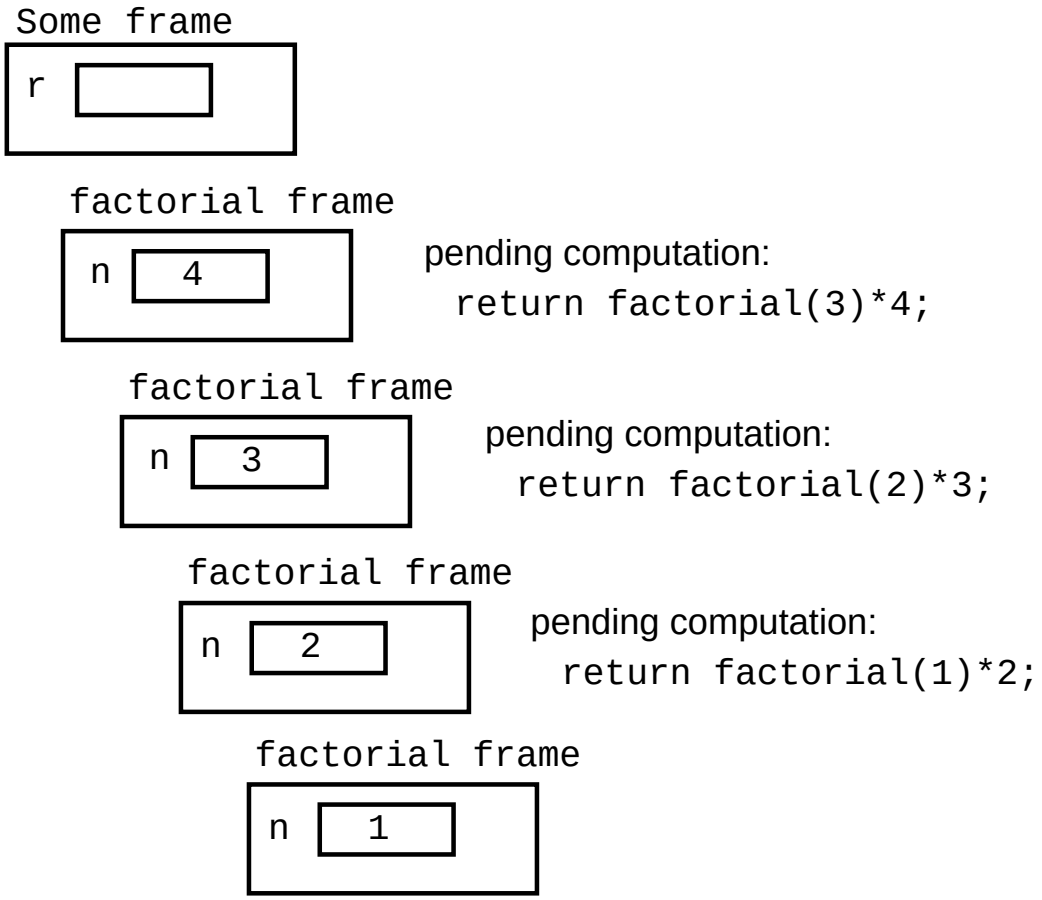
```
return factorial(1-1)*1;
```



Now, we have reached the base case, and n is 0, so we execute:

```
return 1;
```

We get rid of the frame, and pass the returned value to the caller



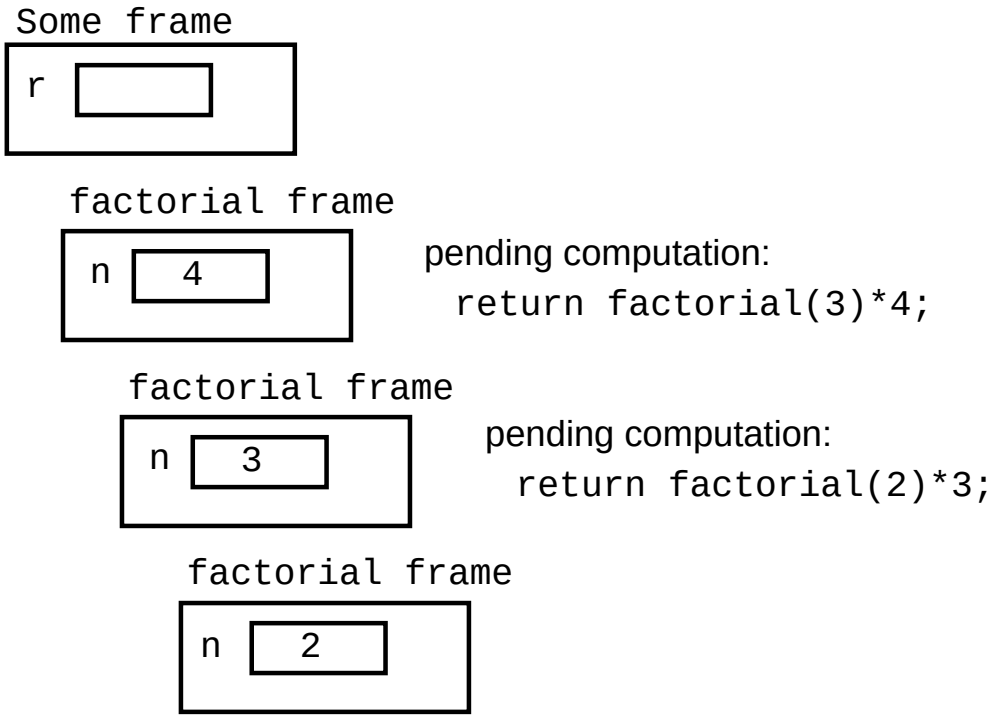
The pending computation here was:

```
return factorial(0)*1;
```

and the method called `factorial(0)`
returned 1, so this pending computation is now:

```
return 1*1;
```

We get rid of the frame, and pass the returned value to the caller



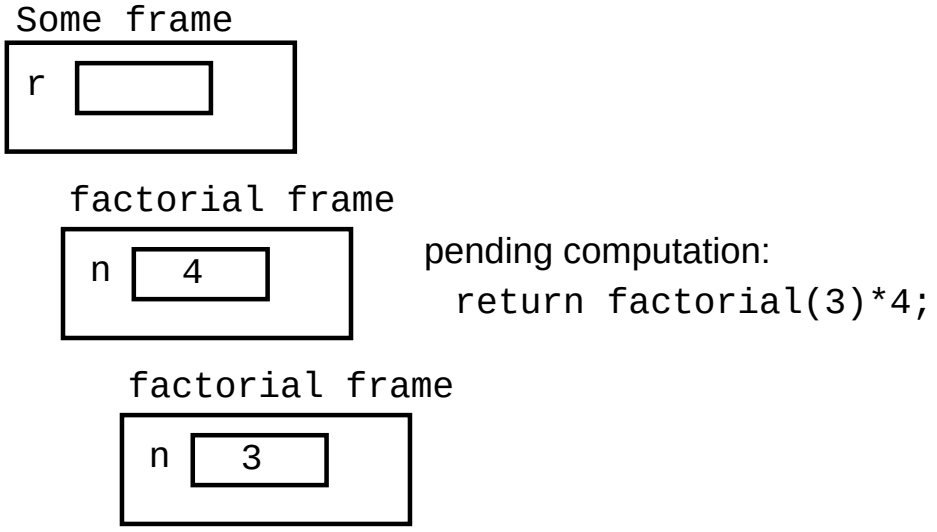
The pending computation here was:

```
return factorial(1)*2;
```

and the method called `factorial(1)`
returned 1, so this pending computation is now:

```
return 1*2;
```

We get rid of the frame, and pass the returned value to the caller



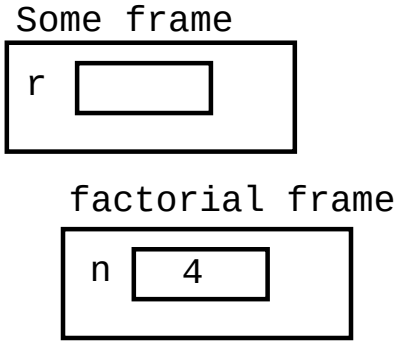
The pending computation here was:

```
return factorial(2)*3;
```

and the method called `factorial(2)`
returned 2, so this pending computation is now:

```
return 2*3;
```

We get rid of the frame, and pass the returned value to the caller



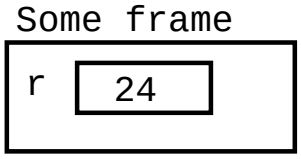
The pending computation here was:

```
return factorial(3)*4;
```

and the method called `factorial(3)`
returned 6, so this pending computation is now:

```
return 6*4;
```

We get rid of the frame, and pass the returned value to the caller



The pending computation here was:

```
r = factorial(4);
```

which returned 24, so this pending computation is now:

```
r = 24;
```

Recursion on other types

- Problem: given a string s, return the reverse of the string

- Analysis:
  - Notation:
    - $\mathsf{rev}(s)$ is the reverse of s
    - s_i is the i-th character of s
    - $\mathsf{len}(s)$ is the length of s
    - $\mathsf{rest}(s)$ is the string s without its first character s_0 (i.e. $\mathsf{rest}(s) = s_1 s_2 ... s_n$ where $n = \mathsf{len}(s) - 1$)
  - Formal definition of reverse:

$$\mathsf{rev}(s) = \begin{cases} \text{“”} & \text{if } s = \text{“”} \\ \mathsf{rev}(\mathsf{rest}(s)) + s_0 & \text{otherwise} \end{cases}$$

Reverse (contd.)

- For example:

$$
\begin{aligned}
\mathsf{rev}(\text{“}abcd\text{”}) &= \mathsf{rev}(\text{“}bcd\text{”}) +' a' \\
&= (\mathsf{rev}(\text{“}cd\text{”}) +' b') +' a' \\
&= ((\mathsf{rev}(\text{“}d\text{”}) +' c') +' b') +' a' \\
&= (((\mathsf{rev}(\text{“”}) +' d') +' c') +' b') +' a' \\
&= (((\text{“”} +' d') +' c') +' b') +' a' \\
&= ((\text{“}d\text{”} +' c') +' b') +' a' \\
&= (\text{“}dc\text{”} +' b') +' a' \\
&= \text{“}dcb\text{”} +' a' \\
&= \text{“}dcba\text{”}
\end{aligned}
$$

Reverse (contd.)

```
public class MoreStringOperations {
  static String reverse(String s)
  {
    if (s.equals(“”)) {
      return “”;
    }
    return reverse(rest(s))+s.charAt(0);
  }
  static String rest(String s)
  {
    String result =“”;
    int i = 1;
    while (i < s.length()) {
      result = result + s.charAt(i);
      i++;
    }
    return result;
  }
}
```

Double recursion

- Problem: Compute the n-th Fibonacci number
- Analysis: The Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, ...is defined by:

$$fib(n) = \begin{cases} 1 & \text{if } n \leqslant 2 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

- Implementation:

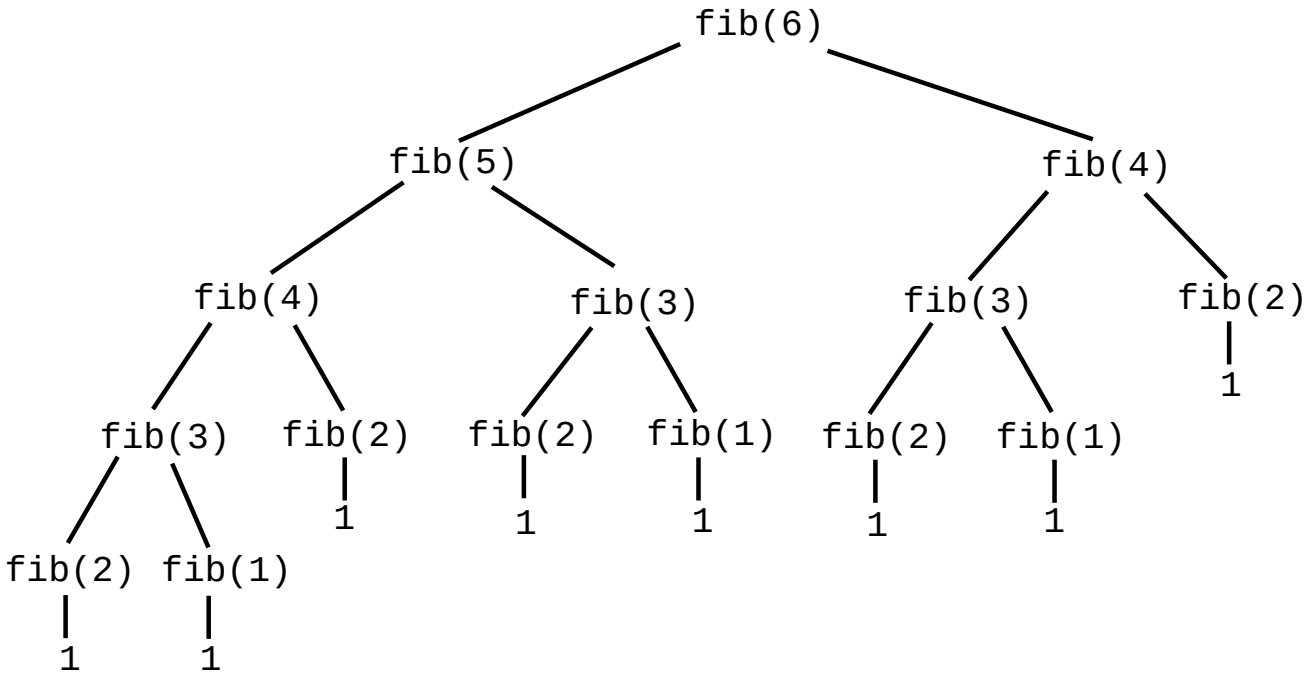
```
static int fib(int n)
{
    if (n <= 2) {
        return 1;
    }
    return fib(n-1)+fib(n-2);
}
```

Iteration vs recursion

- Iterative solution to the Fibonacci problem:

```
static int fib(int n)
{
    int a, b, c, i;
    a = 1;
    b = 1;
    c = 1;
    i = 3;
    while (i <= n) {
        c = a + b;
        a = b;
        b = c;
        i++;
    }
    return c;
}
```

Execution trees



Method overloading

- There can be several (static or not) methods with the same name...
- ...but the type or number of parameters must be different

Example

```
public class A {
  void f(int x)
  {
    System.out.println(“one: ”+x)
  }
  void f(boolean x)
  {
    System.out.println(“two: ”+x)
  }
}
public class B {
  void g()
  {
    A u = new A();
    u.f(5);
    u.f(false);
  }
}
```

Same for static methods

```
public class A {
  static void f(int x)
  {
    System.out.println("one: "+x)
  }
  static void f(boolean x)
  {
    System.out.println("two: "+x)
  }
}
public class B {
  void g()
  {
    A.f(5);
    A.f(false);
  }
}
```

The end