Recursion

ACS-1904 LECTURE 10

Recursion

- Recursion:
  - applies to methods and to data structures.
- A recursive method is a method that, directly or indirectly, calls itself.
- A recursive data structure is a structuring of classes so an instance can have a reference to another instance of the same class

Recursive methods

A recursive method :

- breaks a problem into smaller problems of the same problem type such that their solutions can be combined to form a solution.
- The decomposition into smaller problems must be constructed is such a way as to terminate in a finite number of steps, and to end in a basic form that has a known solution.

Example: Factorials

Examples

$0! = 1$

$1! = 1$

$2!= 2x1 = 2$

$3! = 3x2x1 = 6$

Definition:

$0! = 1$

$n! = n * (n-1)!$ for $n>0$

The base case

Instead of n!, now we have a smaller problem … (n-1)!

Example: Factorials

- First the logic of the recursive solution.

1. How can the problem be defined in terms of smaller instances of itself?
    - If **5! = 5 * 4 * 3 * 2 * 1** then it can also be said that

Example: Factorials

- First the logic of the recursive solution.

1. How can the problem be defined in terms of smaller instances of itself?
   - If **5! = 5 * 4 * 3 * 2 * 1** then it can also be said that
   - **5! = 5 * 4!** Because 4! Is 4 * 3 * 2 * 1

Example: Factorials

- First the logic of the recursive solution.

1. How can the problem be defined in terms of smaller instances of itself?
    - If **5! = 5 * 4 * 3 * 2 * 1** then it can also be said that
    - **5! = 5 * 4!** And
    - **4! = 4 * 3!**

Example: Factorials

- First the logic of the recursive solution.

1. How can the problem be defined in terms of smaller instances of itself?
   - If **5! = 5 * 4 * 3 * 2 * 1** then it can also be said that
   - **5! = 5 * 4!** And
   - **4! = 4 * 3!**
   - **3! = 3 * 2!**
   - **2! = 2 * 1!**
   - **1! = 1 * 0!**
   - And **0! = 1** (this is the base case)
   - So it is obvious that 5! Can be easily solved if we know 4! And so on .

Example: Factorials

Factorial.java

```
public class Factorial {

    public int factorial (int n){
        if (n==0)
            return 1;                 // base case is solved

        return n*factorial(n-1); // smaller problem
    }
}
```

When n is 0 there is an immediate return

Otherwise factorial is called again with a smaller value.

The current invocation of factorial is suspended and waits for this next one to complete.

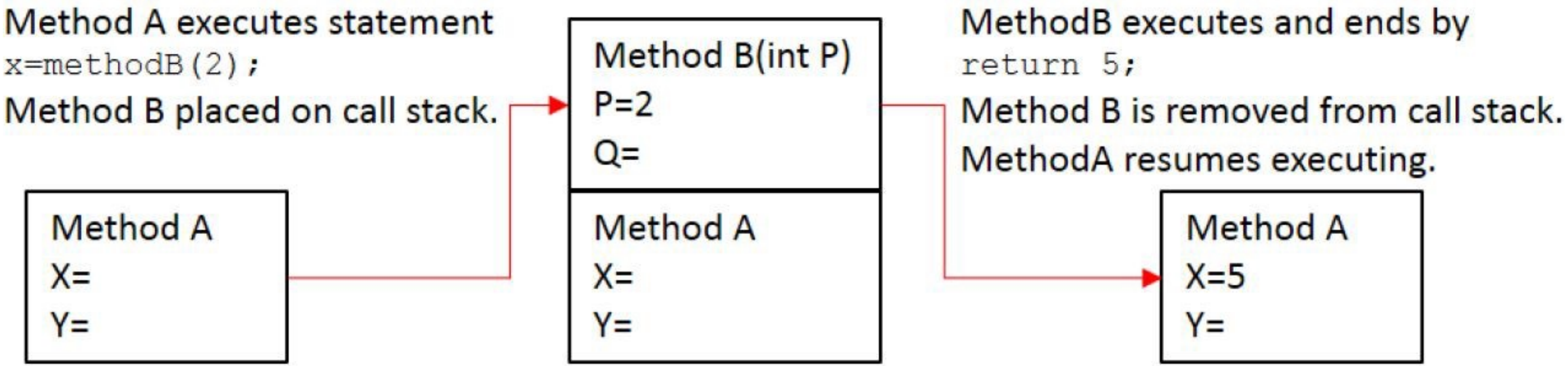
Call stack

To control the execution of methods the JVM uses a memory structure called the Call Stack.

When method A calls method B the JVM :

- ensures the field values of method A are in A's area of the stack,
- notes the location in method A to which control will return,
- allocates a new area on top for the newly called method B.

Call stack



Method A executes statement
`x=methodB(2);`
Method B placed on call stack.

Method A
X=
Y=

Method B(int P)
P=2
Q=

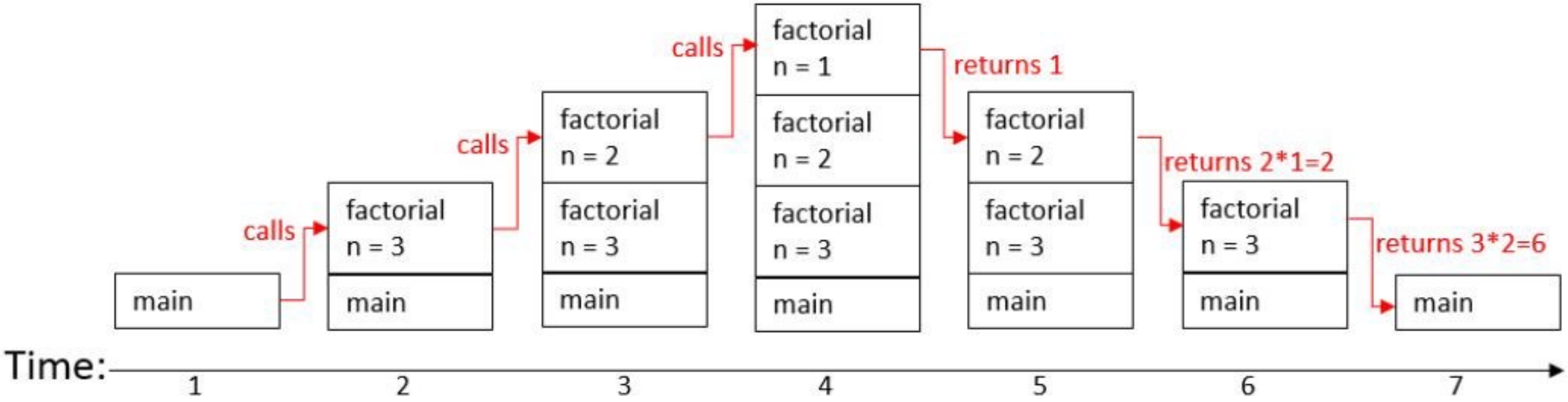
Method A
X=
Y=

MethodB executes and ends by
`return 5;`
Method B is removed from call stack.
MethodA resumes executing.

Method A
X=5
Y=

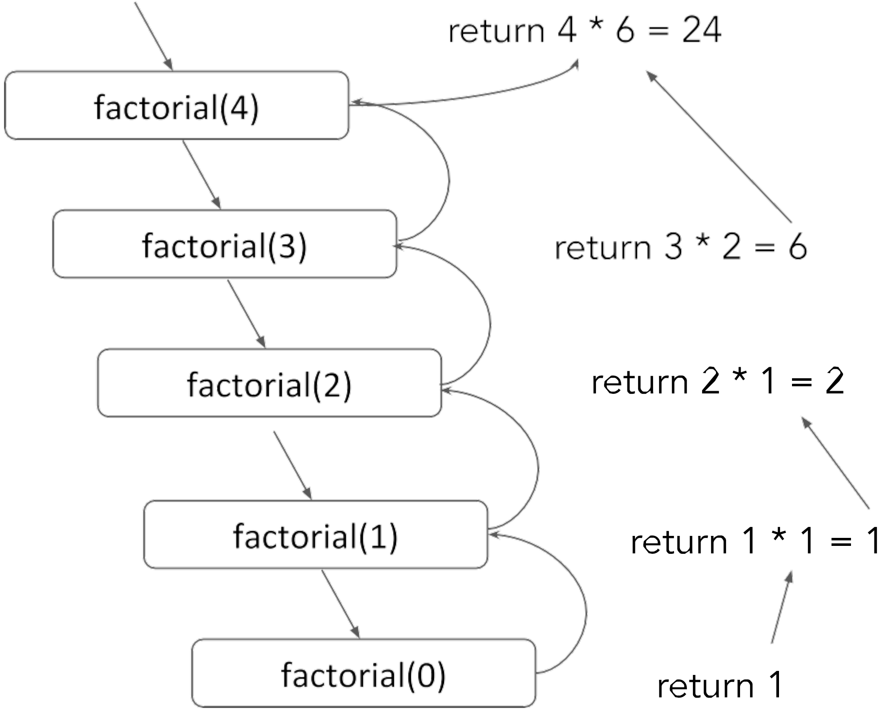
Call stack

Suppose method main calls factorial(3)



Visualizing recursion

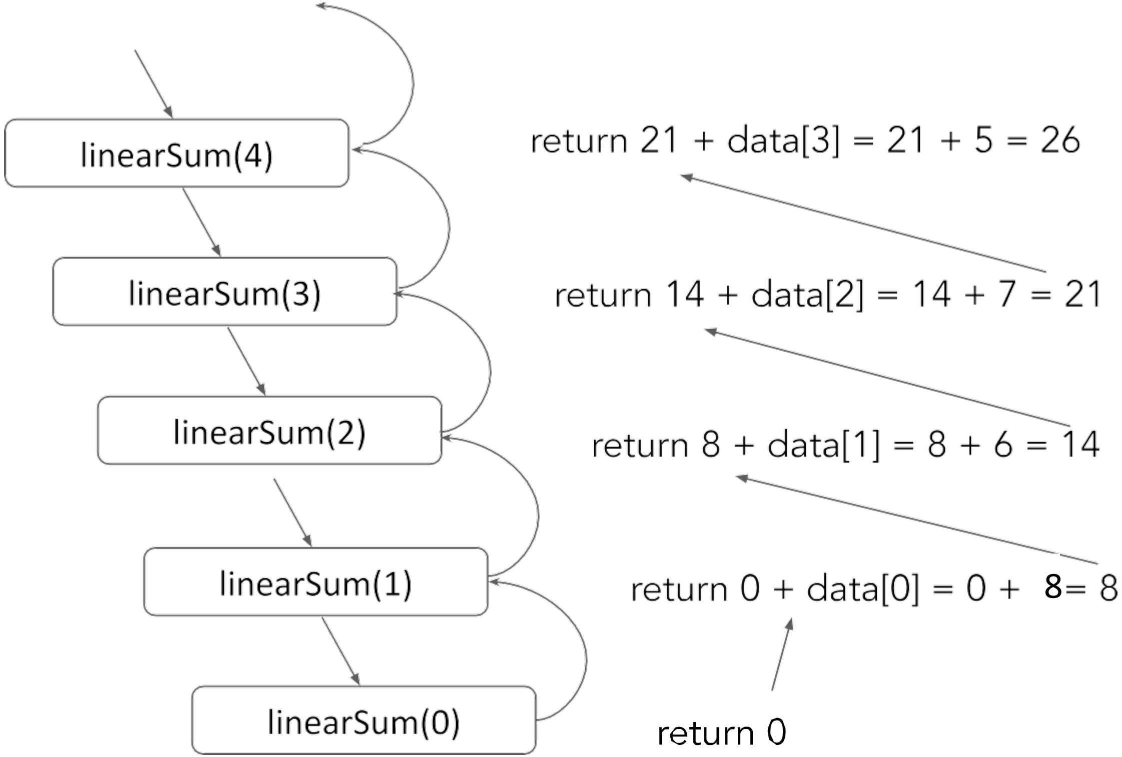
- Recursion Trace



Summing the elements of an array

0	1	2	3	4	5	6
8	6	7	5	3	0	9

```java
public static int linearSum (int[] data, int n){
    if (n==0)
        return 0;
    else
        return linearSum(data, n-1) + data[n-1];
}
```



linearSum(4)

linearSum(3)

linearSum(2)

linearSum(1)

linearSum(0)

return 21 + data[3] = 21 + 5 = 26

return 14 + data[2] = 14 + 7 = 21

return 8 + data[1] = 8 + 6 = 14

return 0 + data[0] = 0 + 8= 8

return 0

Binary recursion

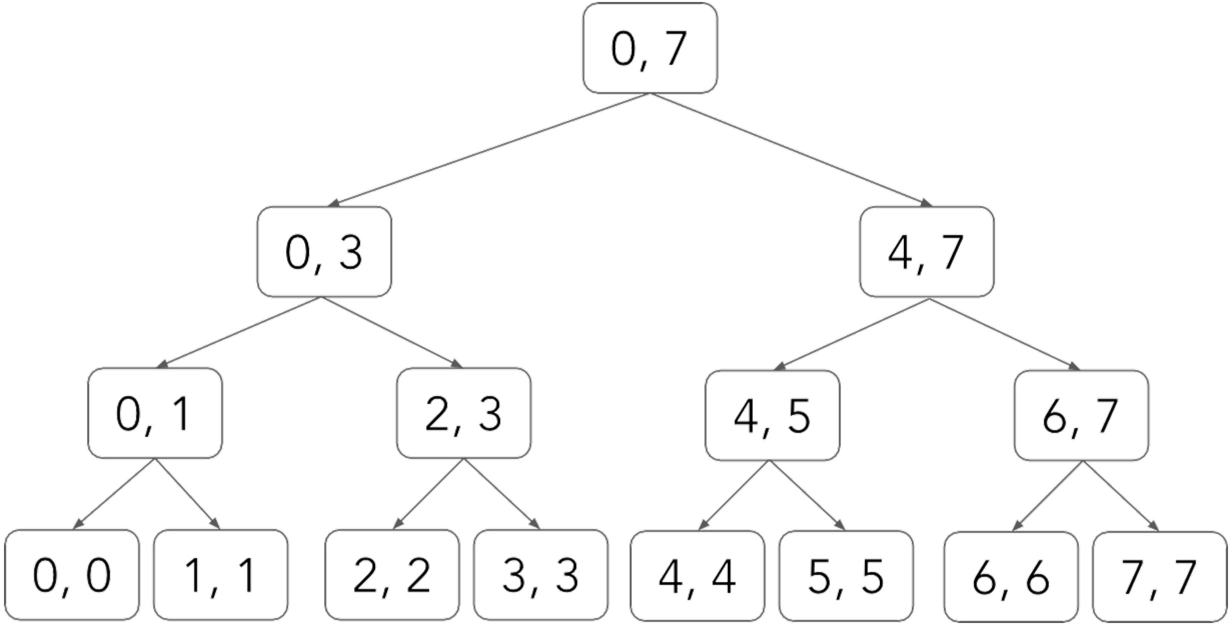
```
public static int binarySum(int[] data, int low, int high){
    if (low > high)
        return 0;
    else if (low == high)
        return data[low];
    else{
        int mid = (low + high) / 2;
        return binarySum(data, low, mid) + binarySum(data, mid+1,
    high);
    }
}
```

Binary recursion

0	1	2	3	4	5	6	7
8	6	7	5	3	0	9	10

```
public static int binarySum(int[] data, int low, int high){
    if (low > high)
        return 0;
    else if (low == high)
        return data[low];
    else{
        int mid = (low + high) / 2;
        return binarySum(data, low, mid) + binarySum(data, mid+1,
    high);
    }
}
```

Recursion trace



Example: Fibonacci Numbers

$$
\begin{aligned}
f_0 &= 0 \\
f_1 &= 1 \\
f_n &= f_{n-1} + f_{n-2}
\end{aligned}
$$

the infinite sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, . . .

Example: Fibonacci Numbers

Fibonacci.java

```
public class Fibonacci
{
    public int fibonacci (int n){
        if (n==0) return 0;     // first base case
        if (n==1) return 1;     // second base case
        return fibonacci(n-1) + fibonacci(n-2);
                               // two simpler problems
    }
}
```

When n is 0 there is an immediate return

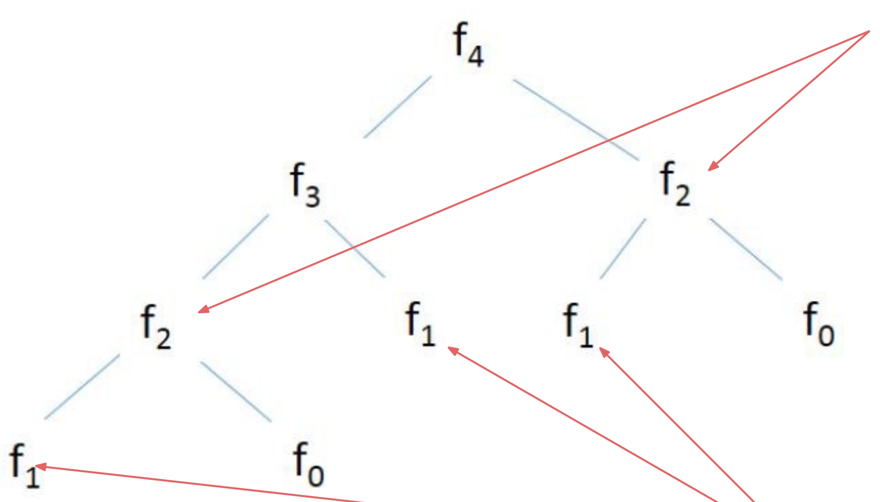
When n is 1 there is an immediate return

Otherwise fibonacci is called again (twice) with a smaller values.

The current invocation of fibonacci is suspended and waits for both fibonacci(n-1) and fibonacci(n-2) to complete.

Recursion trace

fibonacci(4) results in these calls / calculations



Values are recalculated e.g. f_2 twice

e.g. f_1 three times

Recursive data structures

Classes have associations with other classes

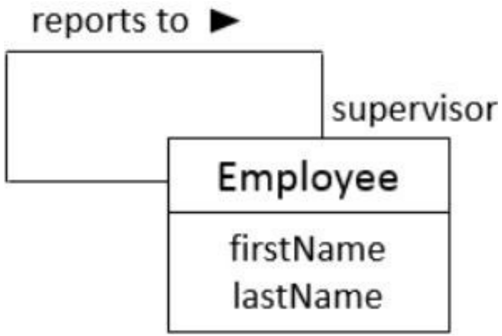
If an association involves the same class more than once we say the association is recursive (or reflexive)

Examples:

- a team plays against another team
- a person is a parent of another person
- an employee reports to another employee

Example: recursive association

An employee reports to another employee



The association can be read two ways

1. an employee reports to another employee, and
2. an employee may supervise many other employees.

We will implement this one

Example: Employee reports to an Employee

```
public class Employee{
    private Employee supervisor;
    private String firstName;
    private String lastName;
    public Employee() {
        firstName="unknown";
        lastName="unknown";
        supervisor = null;
    }
    // getters
    public String getFirstName(){
        return firstName;
    }
    public String getLastName(){
        return lastName;
    }
    public Employee getSupervisor (){
        return supervisor;
    }
```

The supervisor field implements the reports to association
A supervisor is another Employee object → recursive association

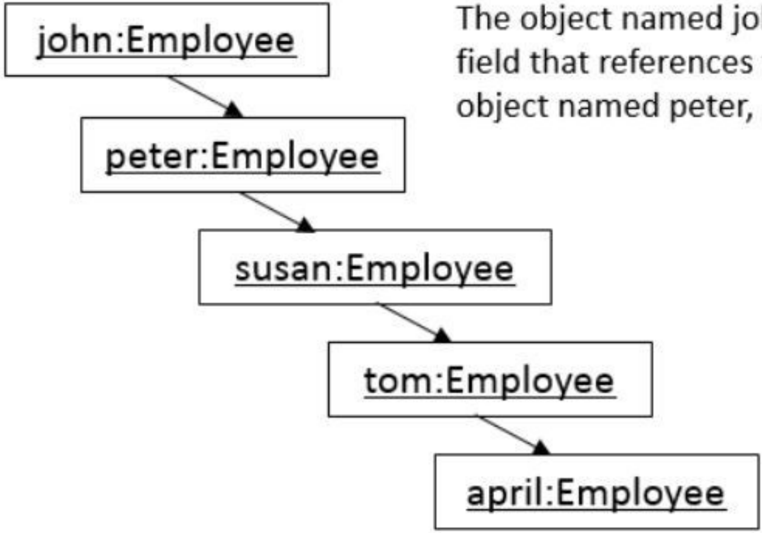
```
    // setters
    public void setFirstName(String first){
        firstName = first;
    }
    public void setLastName(String last){
        lastName = last;
    }
    public void setSupervisor (Employee s){
        supervisor = s;
    }
}
```

Example: Employee reports to an Employee

We can easily implement this reporting structure

A chain of objects

- where each references the next



The object named john has a field that references the object named peter, ...

Example: Employee reports to an Employee

Once we instantiate the five objects : `john, peter, …, april`

We just execute:

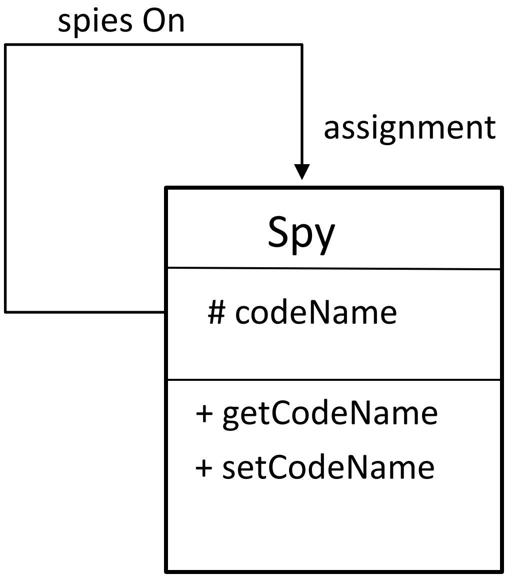
```
john.setSupervisor(peter);
peter.setSupervisor(susan);
susan.setSupervisor(tom);
tom.setSupervisor(april);
```

Example: Employee reports to an Employee

To display the name of someone's supervisor, say the supervisor for employee john:

```
Employee aSuper = peter.getSupervisor();
System.out.println( aSuper.getFirstName()
     +" "+ aSuper.getLastName() );
```

Example: Spy spies on a spy



spies On

assignment

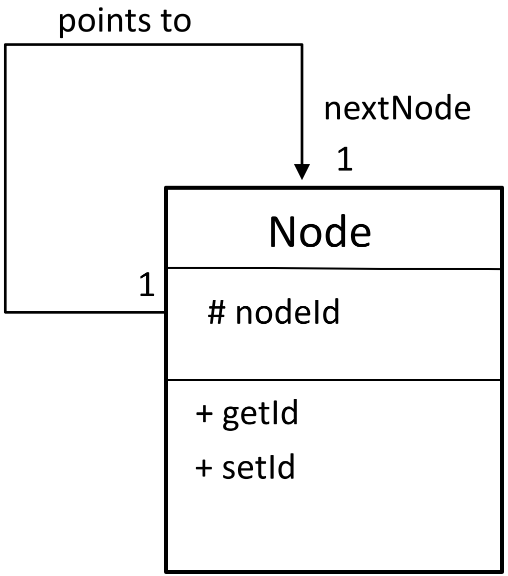
Spy

# codeName

+ getCodeName

+ setCodeName

Example: Node points to a Node



points to

nextNode

1

Node

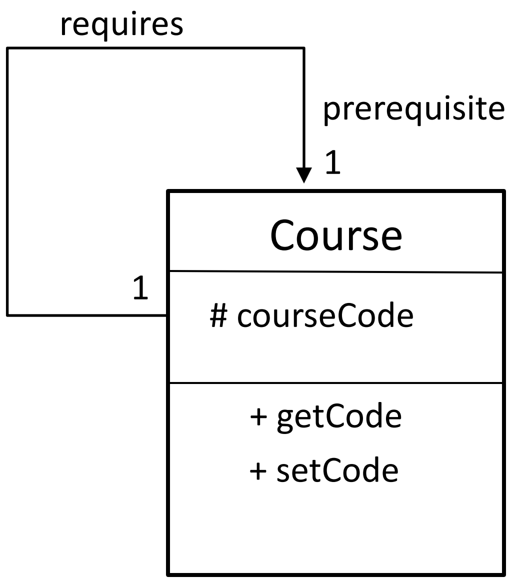
1

# nodeId

+ getId

+ setId

Example: Course requires a Course


requires
prerequisite
1
Course
1
# courseCode
+ getCode
+ setCode