Exception handling

- An exception is generated (raised) with the *throw* statement:

  ```
  throw object;
  ```

  where `object` is an instance of a subclass of `Exception` or `Throwable`

- The *try-catch* statement:

  ```
  try {
    statements;
  }
  catch (ExceptionSubclass1 e) {
    statements1;
  }
  catch (ExceptionSubclass2 e) {
    statements2;
  }
  .
  .
  .
  ```

McGill

Exception handling

```
int some_method(int b)
{
  int a, c, d;
  a = 5;
  if (b != 0) c = a / b;
  else c = 0;
  d = c + 2;
  return d;
}
```

Exception handling

```
int some_method(int b)
{
  int a, c, d;
  try {
    a = 5;
    c = a / b;
    d = c + 2;
  }
  catch (ArithmeticException e) {
    d = 2;
  }
  return d;
}
```

Exception handling

```
class SomeClass {
  static int some_method(int b)
  {
    int a, c, d;
    try {
      a = 5;
      c = a / b;
      d = c + 2;
    }
    catch (ArithmeticException e) {
      d = 2;
    }
    return d;
  }
  static void yet_another_method()
  {
    int x = 5, y;
    y = some_method(x);
    System.out.println(y);
  }
}
```

Exception handling

```
class SomeClass {
  static int some_method(int b) throws ArithmeticE
  {
    int a, c, d;
    a = 5;
    c = a / b;
    d = c + 2;
    return d;
  }
  static void yet_another_method()
  {
    int x = 5, y;
    try {
      y = some_method(x);
    }
    catch (ArithmeticException e) {
      y = 2;
    }
    System.out.println(y);
  }
}
```

Exception handling

```
class Food {
  boolean fresh, smelly;
}
class FoulSmell extends Exception {
  public String toString() {
    return ‘‘Yuck’’;
  }
}
class FoodPoison extends Exception {
  public String toString() {
    return ‘‘Ouch’’;
  }
}
```

Exception handling

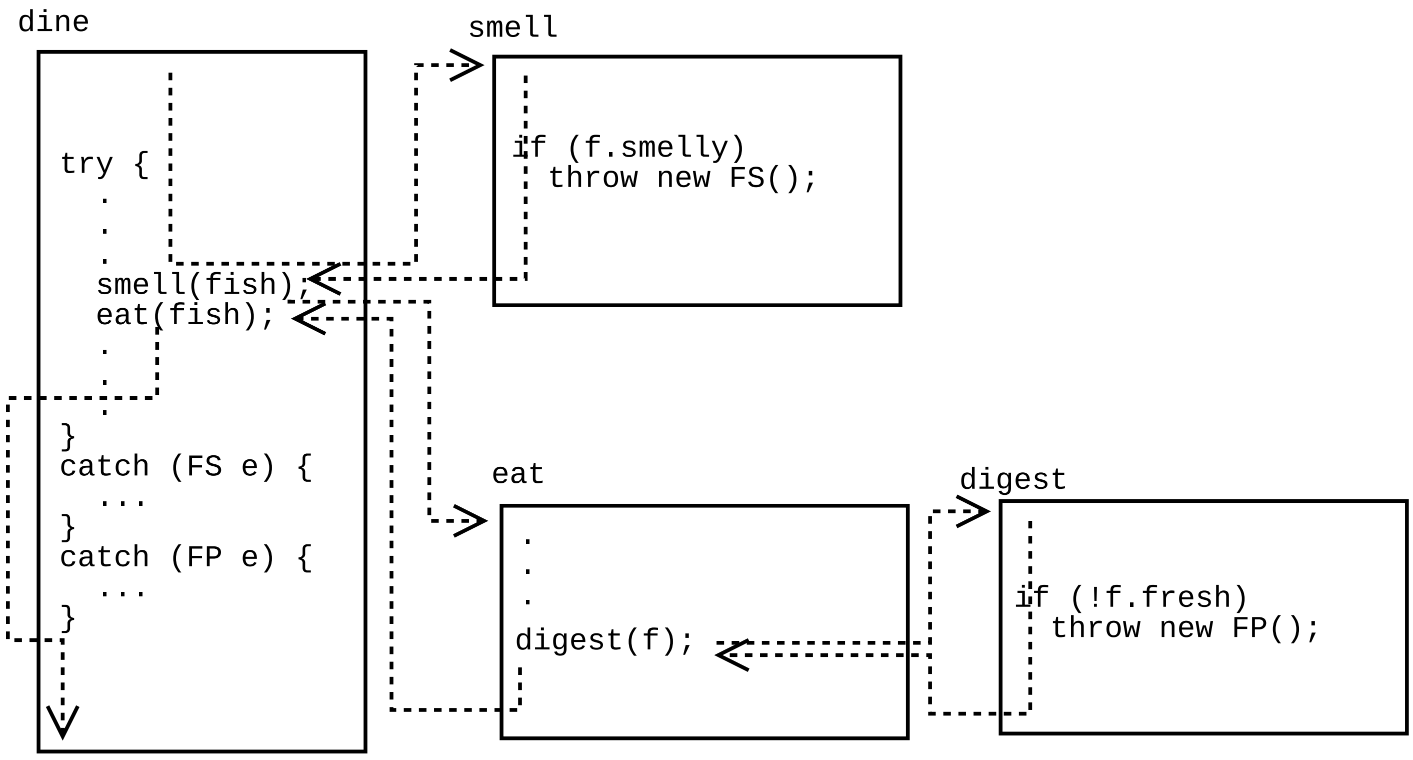
```
static void smell(Food f) throws FoulSmell
{
  if (f.smelly)
    throw new FoulSmell();
  System.out.println("Smells OK");
}
static void eat(Food f) throws FoodPoison
{
  System.out.println("Hmmm...");
  digest(f);
}
static void digest(Food f) throws FoodPoison
{
  if (!f.fresh)
    throw new FoodPoison();
}
```

Exception handling

```
static void dine()
{
  try {
    Food fish = new Food();
    fish.smelly = false;
    fish.fresh = true;
    smell(fish);
    eat(fish);
  }
  catch (FoulSmell e) {
    System.out.println(e);
  }
  catch (FoodPoison e) {
    System.out.println(e);
  }
}
```

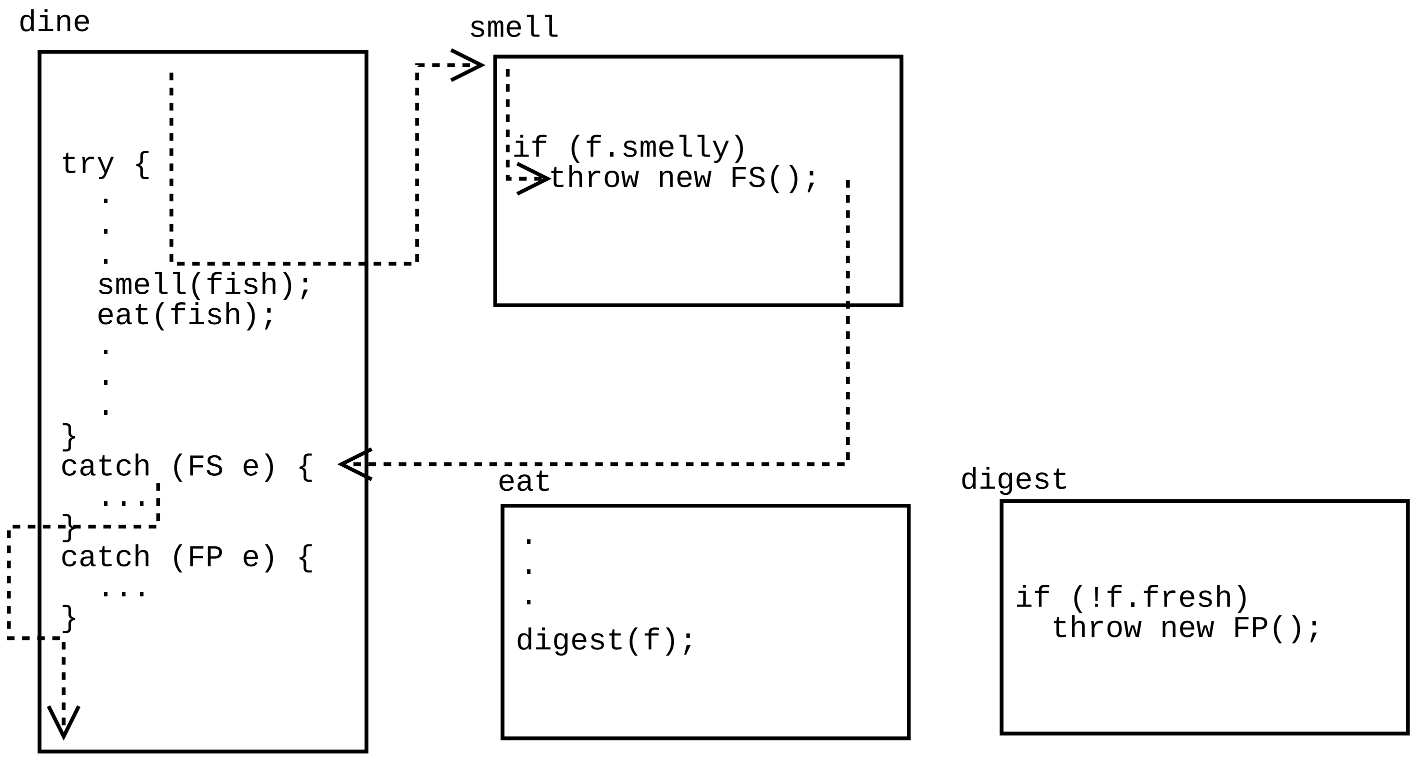
Exception handling

```
fish.smelly = false; fish.fresh = true;
```



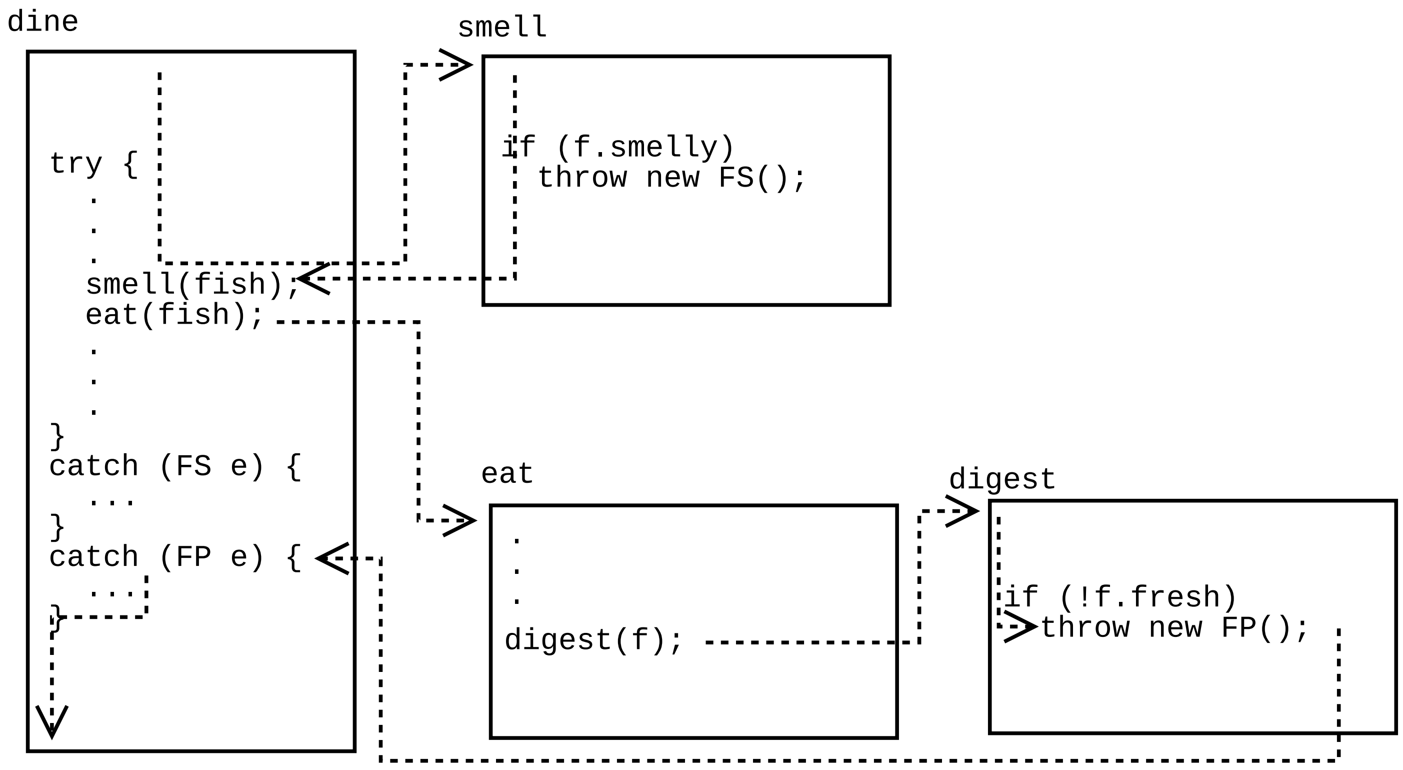
Exception handling

```
fish.smelly = true;
```



Exception handling

```
fish.smelly = false; fish.fresh = false;
```



Exceptions

```
public int book_index(String title)
{
  int i;
  i = 0;
  while (i < book_list.length)
  {
    Book m = book_list[i];
    if (m != null)
    {
      String s = m.title();
      if (s.equals(title))
      {
        return i;
      }
    }
    i++;
  }
  return -1;
}
```

Exceptions

```
public Book find_book(String title)
{
  int i = book_index(title);
  if (i != -1) return book_list[i];
  return null;
}
```

Exceptions

```
Library l = new Library(10000);
Book b;
b = new Book(...);
l.add_book(b);
b = new Book(...);
l.add_book(b);
//...
Book m = l.find_book("Hamlet");
if (m != null)
{
  System.out.println(m.author());
}
else
{
  System.out.println("Not found");
}
```

Exceptions

```
public class BookNotFound extends Exception
{
  private String title;

  public BookNotFound(String t)
  {
    title = t;
  }

  public String toString()
  {
    return ‘‘Book not found: ’’ + title;
  }
}
```

Exceptions

```
public Book find_book(String title)
throws BookNotFound
{
  int i = book_index(title);
  if (i == -1)
  {
    throw new BookNotFound(title);
  }
  return book_list[i];
}
```

Exceptions

```
Library l = new Library(10000);
Book b;
b = new Book(...);
l.add_book(b);
b = new Book(...);
l.add_book(b);
//...
Book m = l.find_book(“Hamlet”);
if (m != null)
{
  System.out.println(m.author());
}
else
{
  System.out.println(“Not found”);
}
```

Exceptions

```
Library l = new Library(10000);
Book b;
b = new Book(...);
l.add_book(b);
b = new Book(...);
l.add_book(b);
//...
try {
  Book m = l.find_book(“Hamlet”);
  System.out.println(m.author());
}
catch (BookNotFound e)
{
  System.out.println(e);
}
```

Exceptions

```
public int book_index(String title)
throws BookNotFound
{
  int i;
  i = 0;
  while (i < book_list.length)
  {
    Book m = book_list[i];
    if (m != null)
    {
      String s = m.title();
      if (s.equals(title))
      {
        return i;
      }
    }
    i++;
  }
  throw new BookNotFound(title);
}
```

Exceptions

```
public Book find_book(String title)
throws BookNotFound
{
  int i = book_index(title);
  if (i == -1)
  {
    throw new BookNotFound(title);
  }
  return book_list[i];
}
```

Exceptions

```
public Book find_book(String title)
throws BookNotFound
{
  int i = book_index(title);
  return book_list[i];
}
```

Exception handling

- Some exceptions arise without an explicit throw.
- Some standard exceptions

```
Exception
  RunTimeException
    IndexOutOfBounds
    StringIndexOutOfBounds
    ArithmeticException
    NullPointerException
  NoSuchMethodException
  ClassNotFoundException
```

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

Recursion

- 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)

Recursion

- For example: Define a set N as follows:
  - $* \in N$
  - If $x \in N$ then $\mathsf{S}x \in N$
- or equivalently:

$$N \overset{def}{=} \{*\} \cup \{\mathsf{S}x \,|\, x \in N\}$$

- What is N?

Recursion

- For example: Define a set N as follows:
  - $* \in N$
  - If $x \in N$ then $\mathsf{S}x \in N$

- By this definition we know that:
  - $* \in N$
  - $\mathsf{S}* \in N$
  - $\mathsf{SS}* \in N$
  - $\mathsf{SSS}* \in N$
  - $\mathsf{SSSS}* \in N$
  - $\dots$

Recursive functions

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

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

Recursive functions

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

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

So

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

Recursive functions

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

$$n! \quad = \quad \underbrace{1 \cdot 2 \cdot 3 \cdot ... \cdot (n-2) \cdot (n-1)}_{(n-1)!} \cdot n$$

Recursive functions

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

$$n! \quad = \quad (n-1)! \cdot n$$

Because

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

Recursive functions

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

$$n! \quad = \quad (n-1)! \cdot n$$

If

$$n > 0$$

Recursive functions

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

$$n! \quad = \quad (n-1)! \cdot n$$

If

$$n > 0$$

and the "base case" is

$$n! = 1$$

If

$$n = 0$$

McGill

Recursive functions (contd.)

Summarizing:

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

Recursive functions (contd.)

Summarizing:

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

This can be implemented as a static recursive method:

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

Computation of a recursive function

Given

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

Compute 4!

$$4! \quad = \quad (4-1)! \ \cdot 4$$

Computation of a recursive function

Given

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

Compute $4!$

$$4! \quad = \quad 3! \quad \cdot 4$$

Computation of a recursive function

Given

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

Compute $4!$

$$\begin{array}{rcl} 4! & = & 3! \cdot 4 \\ & = & ((3-1)! \cdot 3) \cdot 4 \end{array}$$

Computation of a recursive function

Given

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

Compute 4!

$$\begin{aligned} 4! &= 3! \cdot 4 \\ &= (2! \cdot 3) \cdot 4 \end{aligned}$$

Computation of a recursive function

Given

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

Compute 4!

$$\begin{aligned} 4! &= 3! \cdot 4 \\ &= (2! \cdot 3) \cdot 4 \\ &= (((2-1)! \cdot 2) \cdot 3) \cdot 4 \end{aligned}$$

Computation of a recursive function

Given

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

Compute 4!

$$\begin{aligned} 4! &= 3! \cdot 4 \\ &= (2! \cdot 3) \cdot 4 \\ &= ((1! \cdot 2) \cdot 3) \cdot 4 \end{aligned}$$

Computation of a recursive function

Given

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

Compute 4!

$$\begin{aligned} 4! &= 3! \cdot 4 \\ &= (2! \cdot 3) \cdot 4 \\ &= ((1! \cdot 2) \cdot 3) \cdot 4 \\ &= ((((1-1)! \cdot 1) \cdot 2) \cdot 3) \cdot 4 \end{aligned}$$

Computation of a recursive function

Given

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

Compute 4!

$$\begin{aligned} 4! &= 3! \cdot 4 \\ &= (2! \cdot 3) \cdot 4 \\ &= ((1! \cdot 2) \cdot 3) \cdot 4 \\ &= (((0! \cdot 1) \cdot 2) \cdot 3) \cdot 4 \end{aligned}$$

Computation of a recursive function

Given

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

Compute 4!

$$\begin{aligned} 4! &= 3! \cdot 4 \\ &= (2! \cdot 3) \cdot 4 \\ &= ((1! \cdot 2) \cdot 3) \cdot 4 \\ &= (((0! \cdot 1) \cdot 2) \cdot 3) \cdot 4 \\ &= (((1 \cdot 1) \cdot 2) \cdot 3) \cdot 4 \end{aligned}$$

Computation of a recursive function

Given

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

Compute 4!

$$\begin{aligned} 4! &= 3! \cdot 4 \\ &= (2! \cdot 3) \cdot 4 \\ &= ((1! \cdot 2) \cdot 3) \cdot 4 \\ &= (((0! \cdot 1) \cdot 2) \cdot 3) \cdot 4 \\ &= (((1 \cdot 1) \cdot 2) \cdot 3) \cdot 4 \\ &= ((1 \cdot 2) \cdot 3) \cdot 4 \end{aligned}$$

Computation of a recursive function

Given

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

Compute $4!$

$$\begin{aligned} 4! &= 3! \cdot 4 \\ &= (2! \cdot 3) \cdot 4 \\ &= ((1! \cdot 2) \cdot 3) \cdot 4 \\ &= (((0! \cdot 1) \cdot 2) \cdot 3) \cdot 4 \\ &= (((1 \cdot 1) \cdot 2) \cdot 3) \cdot 4 \\ &= ((1 \cdot 2) \cdot 3) \cdot 4 \\ &= (2 \cdot 3) \cdot 4 \end{aligned}$$

Computation of a recursive function

Given

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

Compute 4!

$$\begin{aligned} 4! &= 3! \cdot 4 \\ &= (2! \cdot 3) \cdot 4 \\ &= ((1! \cdot 2) \cdot 3) \cdot 4 \\ &= (((0! \cdot 1) \cdot 2) \cdot 3) \cdot 4 \\ &= (((1 \cdot 1) \cdot 2) \cdot 3) \cdot 4 \\ &= ((1 \cdot 2) \cdot 3) \cdot 4 \\ &= (2 \cdot 3) \cdot 4 \\ &= 6 \cdot 4 \end{aligned}$$

Computation of a recursive function

Given

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

Compute 4!

$$\begin{aligned} 4! &= 3! \cdot 4 \\ &= (2! \cdot 3) \cdot 4 \\ &= ((1! \cdot 2) \cdot 3) \cdot 4 \\ &= (((0! \cdot 1) \cdot 2) \cdot 3) \cdot 4 \\ &= (((1 \cdot 1) \cdot 2) \cdot 3) \cdot 4 \\ &= ((1 \cdot 2) \cdot 3) \cdot 4 \\ &= (2 \cdot 3) \cdot 4 \\ &= 6 \cdot 4 \\ &= 24 \end{aligned}$$

Execution of recursive methods

- Consider the following client for this factorial function:

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

Execution of recursive methods

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

- Execution proceeds as follows:

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

Execution of recursive methods

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

- Execution proceeds as follows:

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

Execution of recursive methods

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

- Execution proceeds as follows:

```
factorial(4)
return factorial(3) * 4
return (factorial(3-1) * 3) * 4
```

Execution of recursive methods

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

- Execution proceeds as follows:

```
factorial(4)
return factorial(3) * 4
return (factorial(2) * 3) * 4
```

Execution of recursive methods

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

- Execution proceeds as follows:

```
factorial(4)
return factorial(3) * 4
return (factorial(2) * 3) * 4
return ((factorial(2-1) * 2) * 3) * 4
```

Execution of recursive methods

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

- Execution proceeds as follows:

```
factorial(4)
return factorial(3) * 4
return (factorial(2) * 3) * 4
return ((factorial(1) * 2) * 3) * 4
```

Execution of recursive methods

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

- Execution proceeds as follows:

```
factorial(4)
return factorial(3) * 4
return (factorial(2) * 3) * 4
return ((factorial(1) * 2) * 3) * 4
return (((factorial(1-1) * 1) * 2) * 3) * 4
```

Execution of recursive methods

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

- Execution proceeds as follows:

```
factorial(4)
return factorial(3) * 4
return (factorial(2) * 3) * 4
return ((factorial(1) * 2) * 3) * 4
return (((factorial(0) * 1) * 2) * 3) * 4
```

Execution of recursive methods

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

- Execution proceeds as follows:

```
factorial(4)
return factorial(3) * 4
return (factorial(2) * 3) * 4
return ((factorial(1) * 2) * 3) * 4
return (((1 * 1) * 2) * 3) * 4
```

Execution of recursive methods

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

- Execution proceeds as follows:

```
factorial(4)
return factorial(3) * 4
return (factorial(2) * 3) * 4
return (((1) * 2) * 3) * 4
```

Execution of recursive methods

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

- Execution proceeds as follows:

```
factorial(4)
return factorial(3) * 4
return ((2) * 3) * 4
```

Execution of recursive methods

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

- Execution proceeds as follows:

```
factorial(4)
return (6) * 4
```

Execution of recursive methods

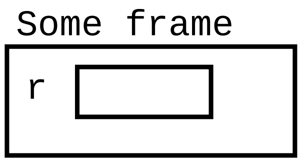
```
static int factorial(int n)
{
    if (n == 0)
    {
        return 1;
    }
    return factorial( n - 1 ) * n;
}
```

- Execution proceeds as follows:

```
factorial(4)
return 24
```

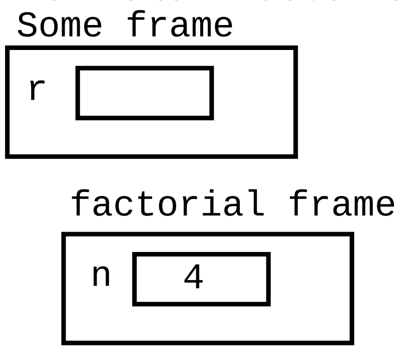
Execution of recursive methods

This is executed in some frame:



Execution of recursive methods

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



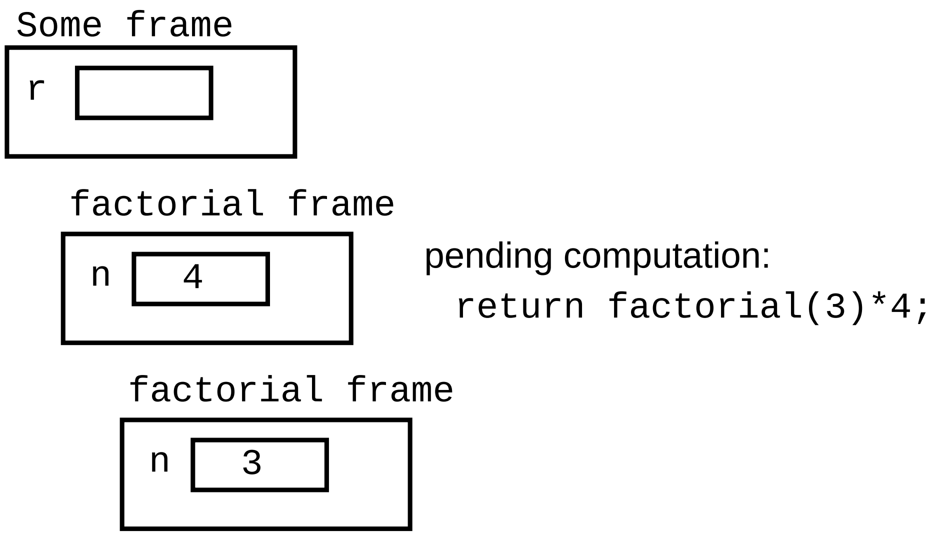
We execute the body of factorial; n is not 0 so we execute

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

which in this frame is the same as

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

Execution of recursive methods



pending computation:

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

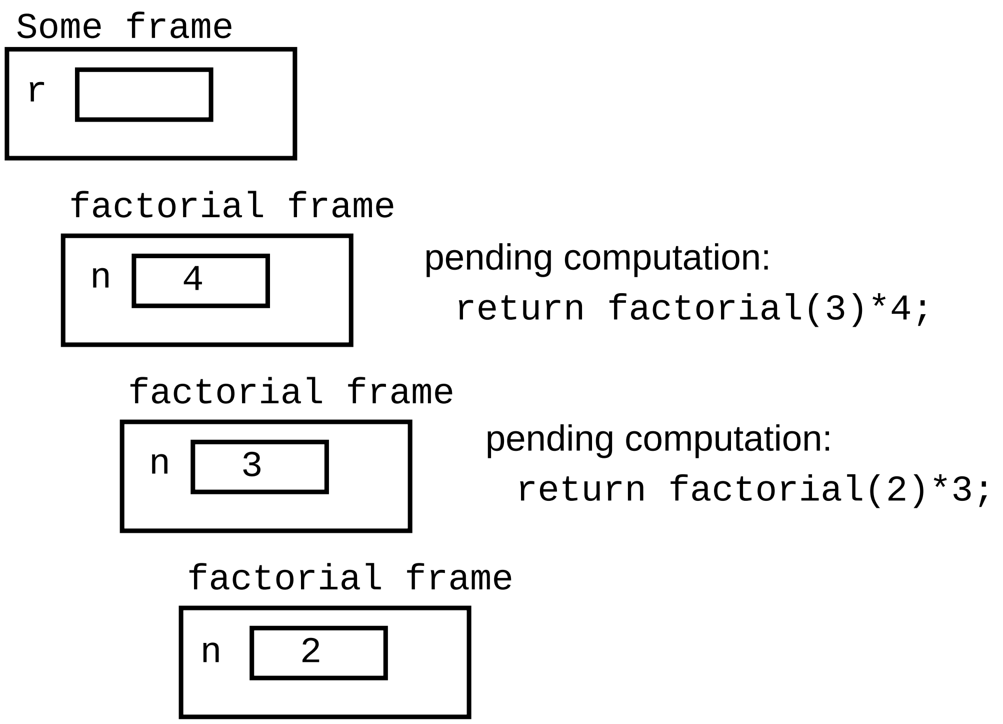
Again, we execute the body of factorial;
again, n is not 0 so we execute

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

which in this frame is the same as

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

Execution of recursive methods



pending computation:

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

pending computation:

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

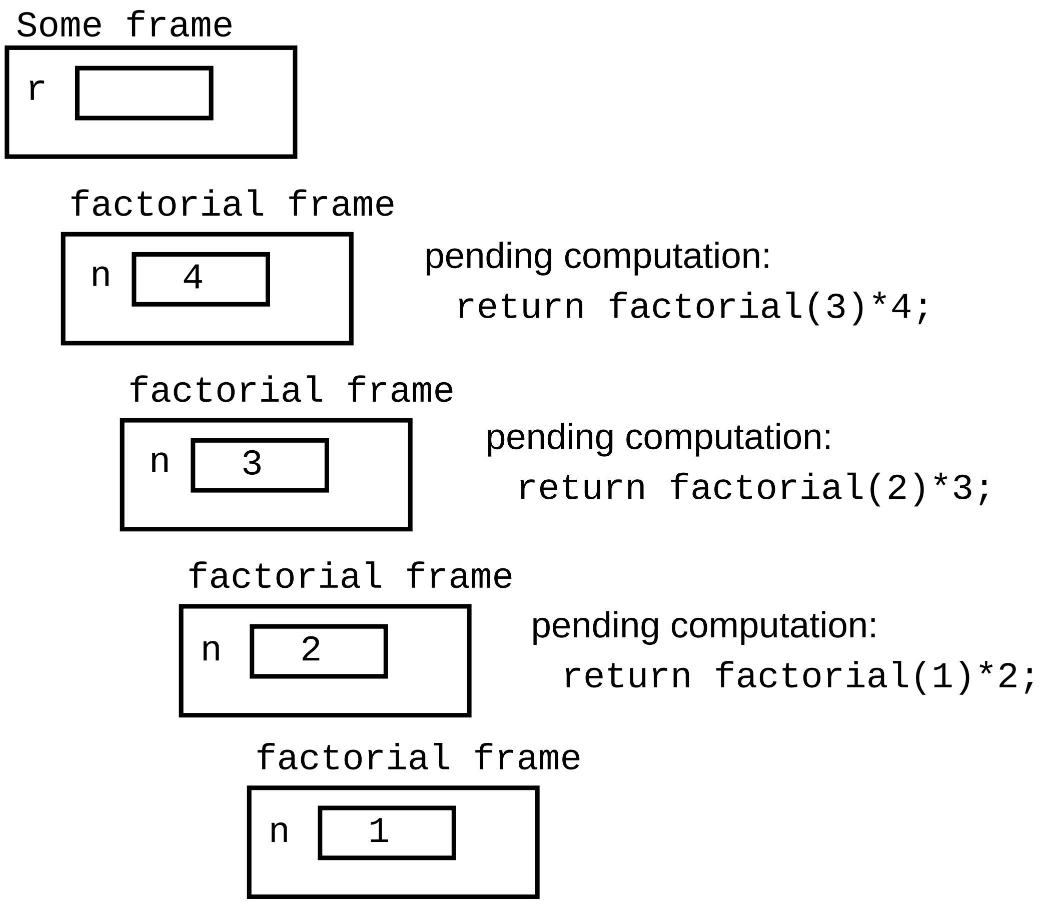
Again, we execute the body of factorial;
again, n is not 0 so we execute

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

which in this frame is the same as

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

Execution of recursive methods



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

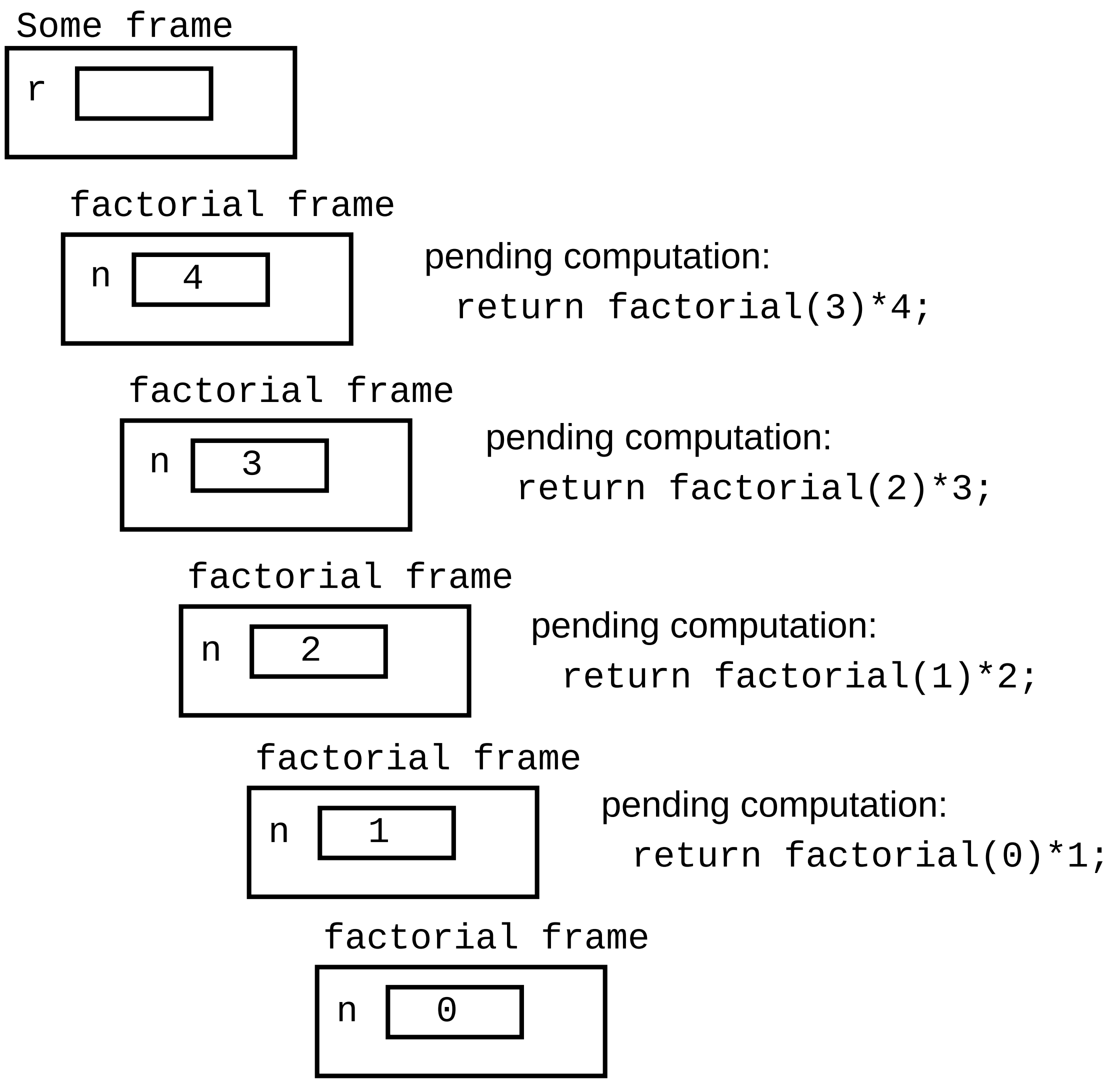
```
return factorial(n-1)*n;
```

which in this frame is the same as

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

McGill

Execution of recursive methods



pending computation:

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

pending computation:

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

pending computation:

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

pending computation:

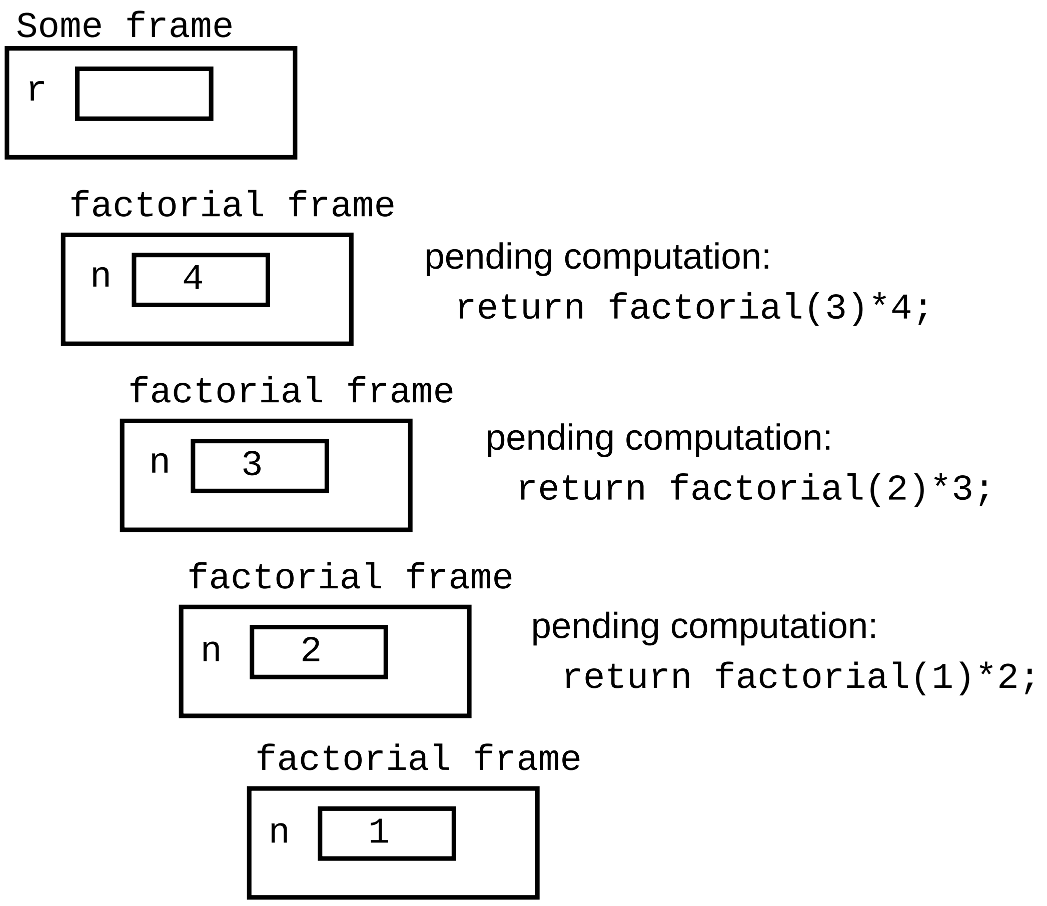
```
return factorial(0)*1;
```

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

Execution of recursive methods



The pending computation here was:

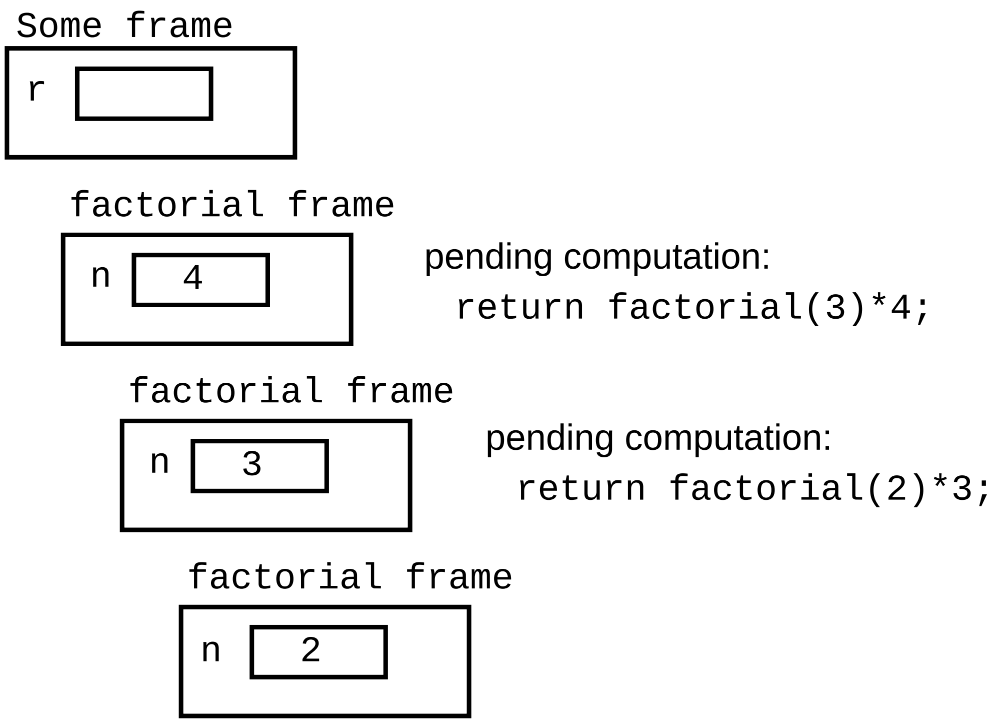
```
return factorial(0)*1;
```

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

Execution of recursive methods



The pending computation here was:

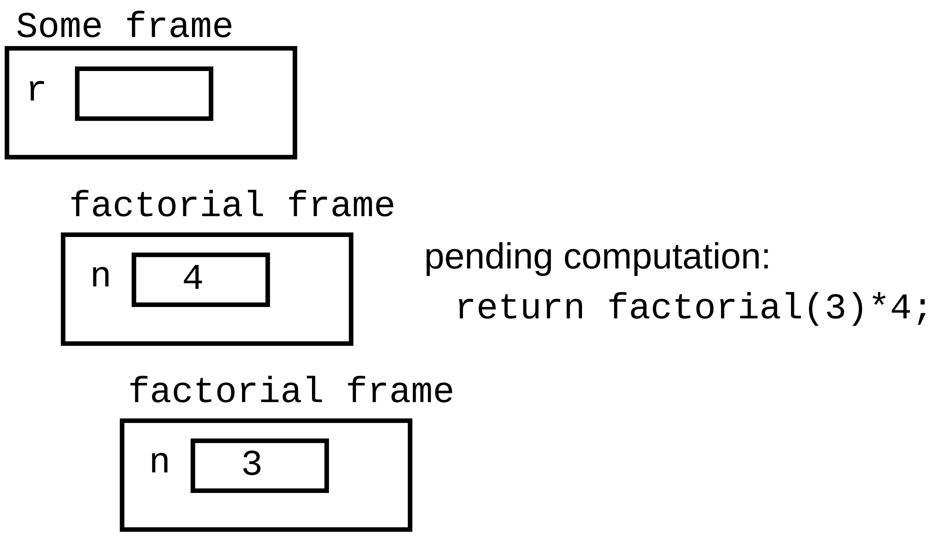
```
return factorial(1)*2;
```

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

Execution of recursive methods



The pending computation here was:

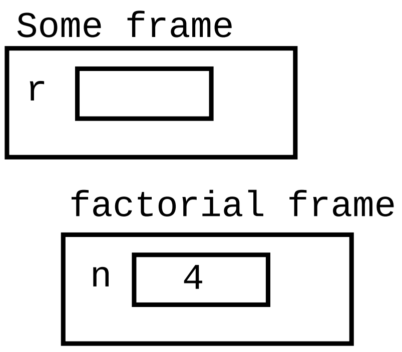
```
return factorial(2)*3;
```

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

Execution of recursive methods



The pending computation here was:

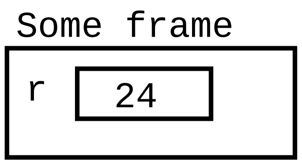
```
return factorial(3)*4;
```

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

Execution of recursive methods



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.)

```
static String reverse(String s)
{
  if (s.equals(“”)) {
    return “”;
  }
  return reverse(rest(s))+s.charAt(0);
}
```

Reverse (contd.)

```
static String rest(String s)
{
  String result =‘“’;
  int i = 1;
  while (i < s.length()) {
    result = result + s.charAt(i);
    i++;
  }
  return result;
}
```

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:

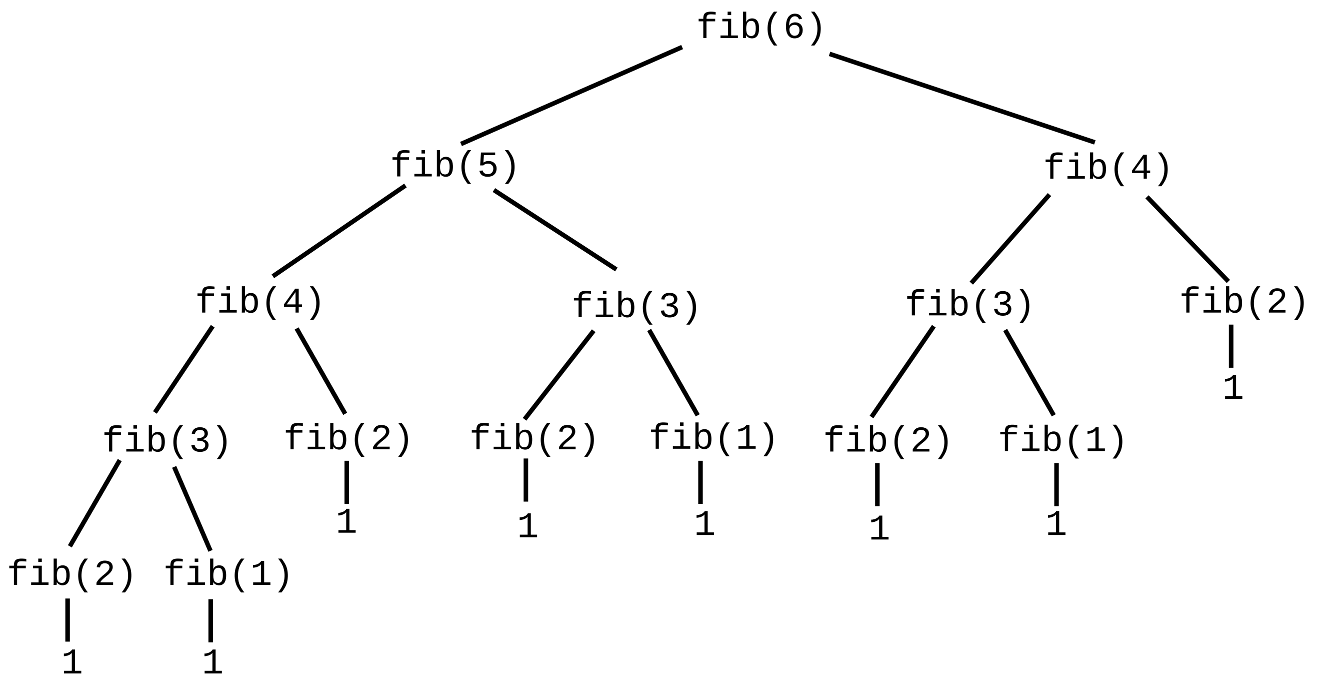
```
static int fib(int n)
{
    if (n <= 2) {
        return 1;
    }
    return fib(n - 1) + fib(n - 2);
}
```

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



Divide and Conquer

- To solve a problem:

1. Divide it into smaller sub-problems
2. Solve each subproblem separately
3. Combine the solution of the subproblems

Divide and Conquer

Problem: prepare dinner

Divide and Conquer

Problem: prepare dinner

- Subproblem 1: Decide what to eat
- Subproblem 2: Get recipie
- Subproblem 3: Get ingredients
- Subproblem 4: Carry out recipie

Divide and Conquer

Problem: prepare dinner

- Subproblem 1: Decide what to eat
- Subproblem 2: Get recipie
- Subproblem 3: Get ingredients
  - Subproblem 3.1: Go to the supermarket
  - Subproblem 3.2: Select ingredients
  - Subproblem 3.3: Pay
- Subproblem 4: Carry out recipie

Divide and Conquer

Problem: prepare dinner

- Subproblem 1: Decide what to eat
- Subproblem 2: Get recipie
- Subproblem 3: Get ingredients
  - Subproblem 3.1: Go to the supermarket
  - Subproblem 3.2: Select ingredients
  - Subproblem 3.3: Pay
- Subproblem 4: Carry out recipie
  - Subproblem 4.1: Cut vegetables
  - Subproblem 4.2: Season meat
  - Subproblem 4.3: Mix
  - Subproblem 4.4: Stir-fry

Divide and Conquer

- To solve a problem:

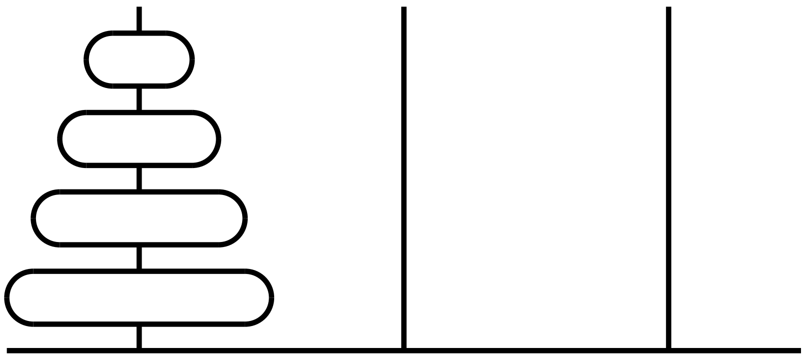
1. Divide it into smaller sub-problems
2. Solve each subproblem separately
   (a) The solution of each subproblem should be written in its own method
   (b) Those methods should be invoked from the method solving the main problem
3. Combine the solution of the subproblems

Divide and Conquer

- When the subproblems are "smaller" instances of the original problem, then use recursion:
  - To compute $n!$ you need to ...
  - ... compute $(n-1)!$
  - ... which is a "smaller" instance of the original problem

Divide and Conquer

- The Towers of Hanoi puzzle:



- Goal: To move the tower from the first peg to the last according to the following rules

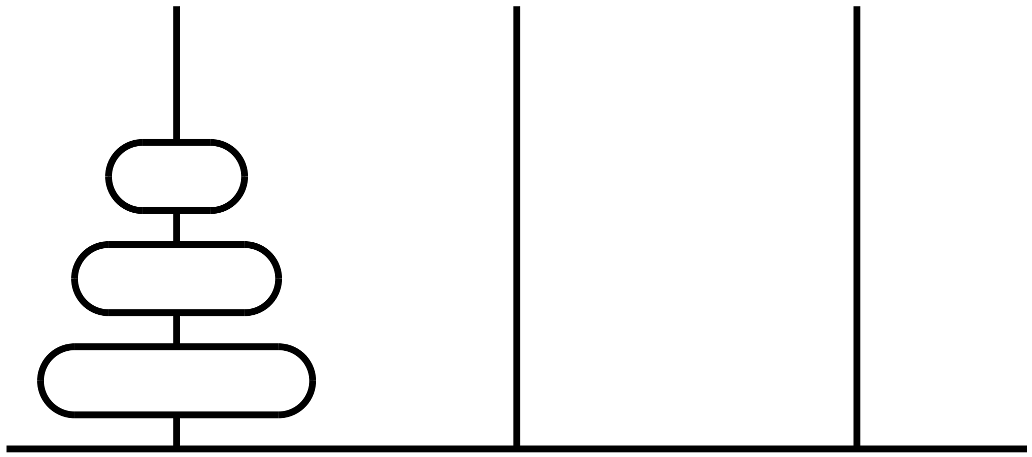
- Rules:

1. Only one disk can be moved at a time

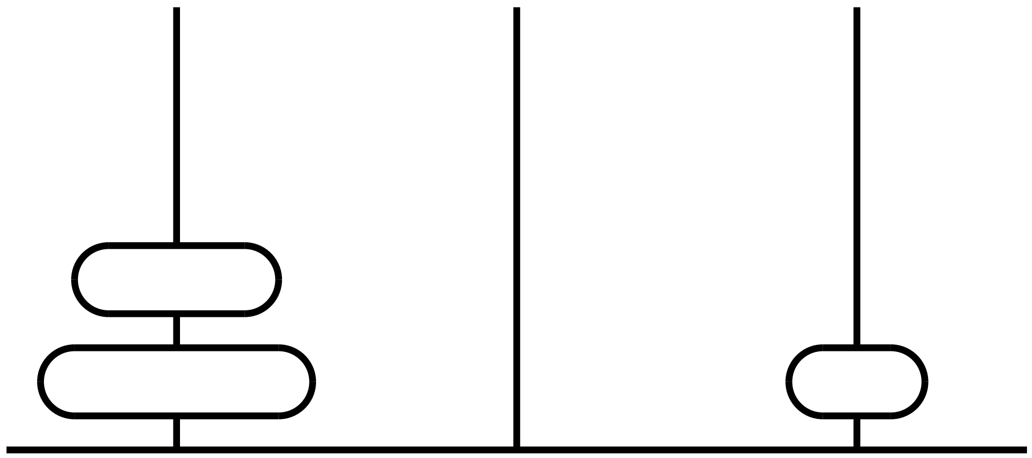
2. A disk cannot be placed on top of a smaller disk

3. All disks must be on a peg, except for the disk being moved

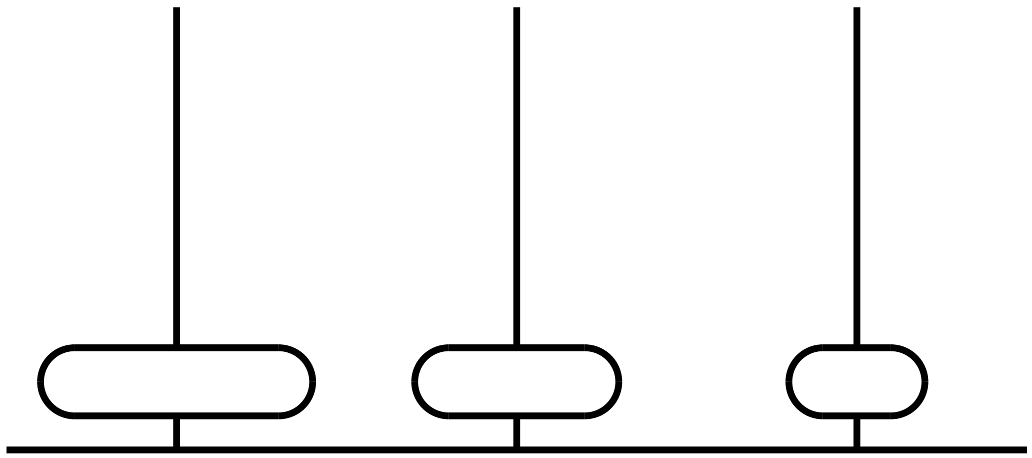
Divide and Conquer



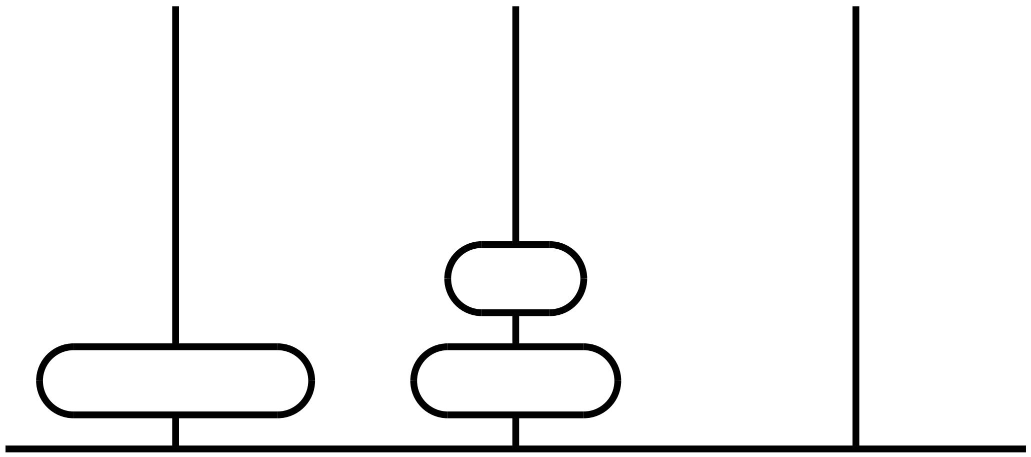
Divide and Conquer



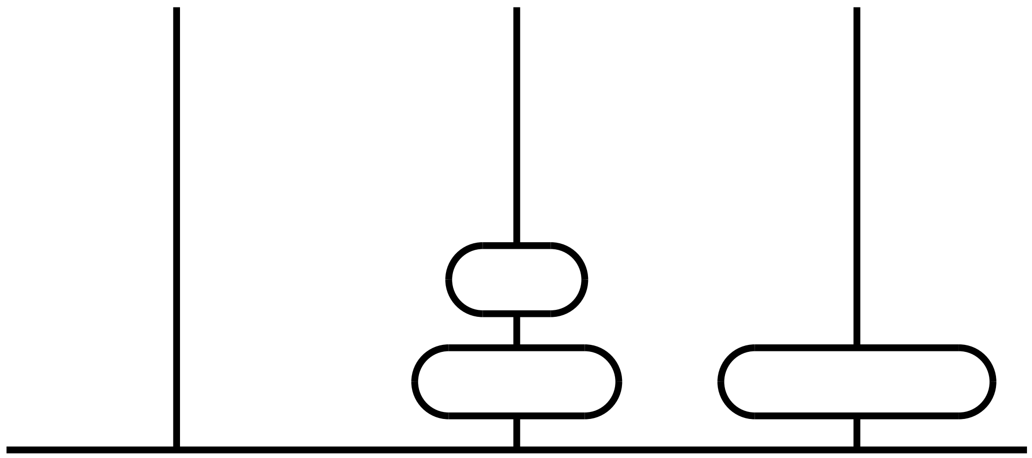
Divide and Conquer



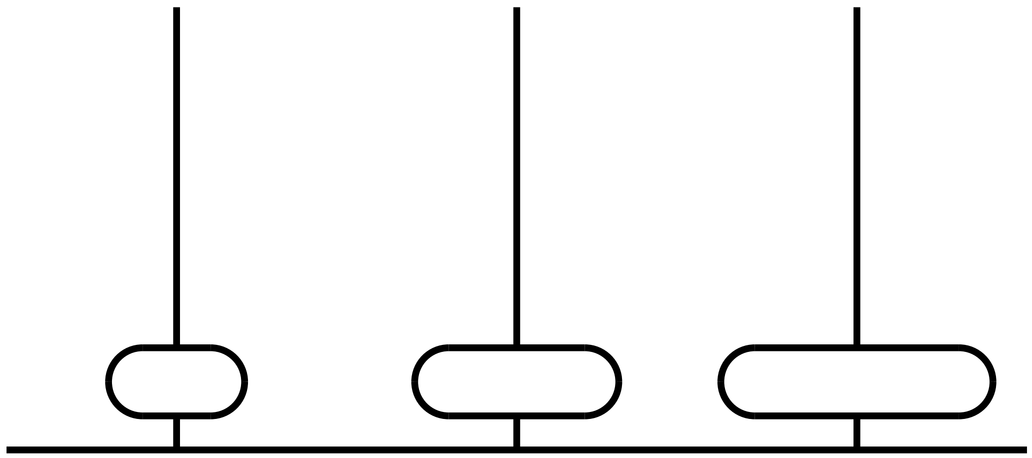
Divide and Conquer



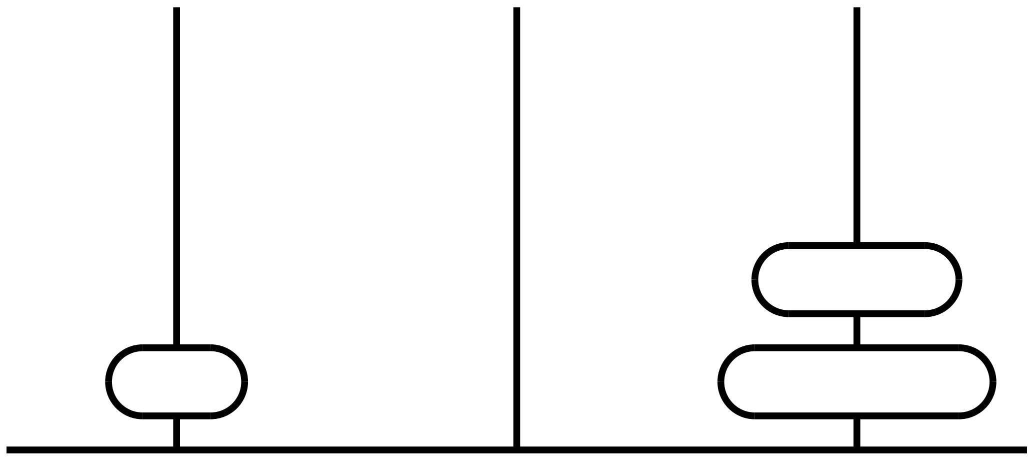
Divide and Conquer



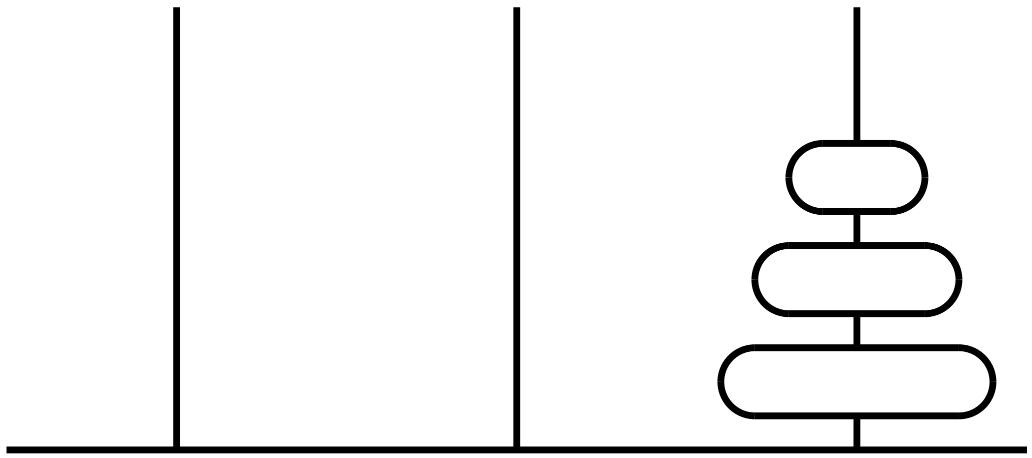
Divide and Conquer



Divide and Conquer



Divide and Conquer



Divide and Conquer

To move a stack of n disks from the original peg to the destination peg:

1. Move the topmost $n-1$ disks from the original peg to the extra peg
2. Move the largest disk from the original peg to the destination peg
3. Move the $n-1$ disks from the extra peg to the destination peg

Divide and Conquer

```
public class TowerOfHanoiSolver
{
  public static void main(String[] args)
  {
    HanoiTowers towers = new HanoiTowers(4);

    towers.solve();
  }
}
```

Divide and Conquer

```
public class HanoiTowers
{
  private int total_disks;

  public HanoiTowers(int disks)
  {
    total_disks = disks;
  }

  public void solve()
  {
    move_tower(total_disks, 1, 3, 2);
  }

  private void move_one_disk(int origin, int dest)
  {
    System.out.println(‘‘Move one disk from ’’
                     + origin + ‘‘ to ’’ + dest);
  }

  // Continues below
  private void move_tower(int disks,
                          int origin,
                          int dest,
                          int extra)
  {
    if (disks == 1)
    {
      move_one_disk(origin, dest);
    }
    else
    {
      move_tower(disks - 1, origin, extra, dest);
      move_one_disk(origin, dest);
      move_tower(disks - 1, extra, dest, origin);
    }
  }
}
```

Recursion

Problem: write a method that given a double b and a positive integer n, computes b^n

Recursion

Problem: write a method that given a double b and a positive integer n, computes b^n

```
double pow(double b, int n)
{
  double product;
  int i;

  product = 1.0;
  i = 1;
  while (i <= n)
  {
    product = product * b;
    i++;
  }

  return product;
}
```

Recursion

$$b^n = \underbrace{b \cdot b \cdot b \cdot \cdots \cdot b \cdot b}_{n \text{ times}}$$

Recursion

$$b^n \quad = \quad b \cdot \underbrace{b \cdot b \cdot \cdots \cdot b \cdot b}_{n-1 \text{ times}}$$

Recursion

$$b^n = b \cdot b^{n-1}$$

Recursion

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{if } n > 0 \end{cases}$$

Recursion

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{if } n > 0 \end{cases}$$

```
double pow(double b, int n)
{
  if (n == 0) return 1;
  return b * pow(b, n - 1);
}
```

Recursion

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{if } n > 0 \text{ and } n \text{ is odd} \\ (b^{n/2})^2 & \text{if } n > 0 \text{ and } n \text{ is even} \end{cases}$$

Recursion

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{if } n > 0 \text{ and } n \text{ is odd} \\ (b^{n/2})^2 & \text{if } n > 0 \text{ and } n \text{ is even} \end{cases}$$

```
double fastpow(double b, int n)
{
  if (n == 0) return 1;
  if (n % 2 == 1) return b * fastpow(b, n - 1);
  double p = fastpow(b, n/2);
  return p * p;
}
```

McGill

Recursion

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{if } n > 0 \text{ and } n \text{ is odd} \\ (b^2)^{n/2} & \text{if } n > 0 \text{ and } n \text{ is even} \end{cases}$$

```
double fastpow(double b, int n)
{
  if (n == 0) return 1;
  if (n % 2 == 1) return b * fastpow(b, n - 1);
  return fastpow(b * b, n / 2);
}
```

Recursion and termination

```
static void f(int n)
{
  System.out.println(n);
  f(n);
}
```

Recursion and termination

```
f(5)
f(5)
f(5)
f(5)
.
.
.
```

Recursion and termination

```
static int f(int n)
{
  System.out.println(n);
  return f(n) + 1;
}
```

Recursion and termination

```
f(5)
return f(5) + 1
return (f(5) + 1) + 1
return ((f(5) + 1) + 1) + 1
.
.
.
```

Recursion and termination

```
static int f(int n)
{
  System.out.println(n);
  return f(n-1);
}
```

Recursion and termination

```
f(5)
return f(4)
return f(3)
return f(2)
return f(1)
return f(0)
return f(-1)
.
.
.
```

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  System.out.println(n);
  return f(n);
}
```

Recursion and termination

```
f(5)
return f(5)
return f(5)
return f(5)
return f(5)
return f(5)
return f(5)
.
.
.
```

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  System.out.println(n);
  return f(n-1);
}
```

Recursion and termination

```
f(5)
return f(4)
return f(3)
return f(2)
return f(1)
return f(0)
return 1
```

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  System.out.println(n);
  return f(n - 2) + 1;
}
```

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  System.out.println(n);
  return f(n / 2) + 1;
}
```

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  System.out.println(n);
  return f(n + 2);
}
```

Recursion and termination

```
f(5)
returns f(7)
returns f(9)
returns f(11)
returns f(13)
.
.
.
```

Recursion and termination

- For a recursive method to terminate it is necessary:
  - to have at least one base case
  - the recursive method call must change the parameters so that the subproblem is “smaller” than the original problem

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  else if (n % 2 == 0) return n / 2;
  else return f( f( 3 * n + 1 ) );
}
```

Recursion and termination

If n is odd then the third case applies, and we have to compute

$$f(f(3n+1))$$

But, since n is odd, then $n = 2k+1$ for some $k \geq 0$, which implies that

$$\begin{aligned} 3n+1 &= 3(2k+1)+1 \\ &= 6k+3+1 \\ &= 6k+4 \\ &= 2(3k+2) \end{aligned}$$

so $3n+1$ is even!

Therefore

$$f(3n+1) = \frac{3n+1}{2}$$

so we can rewrite

$$f(f(3n+1))$$

as

$$f(\frac{3n+1}{2})$$

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  else if (n % 2 == 0) return n / 2;
  else return f( (3 * n + 1) / 2 );
}
```

Recursion and termination

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n/2 & \text{if } n \text{ is even} \\ f(\frac{3n+1}{2}) & \text{if } n \text{ is odd} \end{cases}$$

Question: for all $n > 0$, is $\frac{3n+1}{2} < n$?

Answer: no! For $n > 0$

$$3n > 2n$$

so

$$3n + 1 > 2n$$

therefore

$$\frac{3n+1}{2} > n$$

Hence the argument of the recursive call increases!

Does this mean, then that f cannot terminate?

No! eventually $\frac{3n+1}{2}$ will be even, thus reaching a base case (but the proof is hard!)

Recursion and termination

Determining whether a program might terminate is in general hard

Furthermore, it is impossible to write a program that takes as input any program P and decides whether P terminates or not.

Normal Recursion

```
void p(...)
{
   ...
   p(...);
   ...
}
```

Indirect Recursion

```
void p(...)
{
  ...
  q(...);
  ...
}
void q(...)
{
  ...
  r(...);
  ...
}
void r(...)
{
  ...
  p(...);
  ...
}
```

Mutual Recursion

```
void p(...)
{
  ...
  q(...);
  ...
}
void q(...)
{
  ...
  p(...);
  ...
}
```

The end