We’ve covered almost everything in Java! Just a few more things to introduce, which will be covered from time to time.

Recursion: Look at Java Hypertext entry “recursion”.

Note: For next week, the tutorial you have to watch is about loop invariants. We’ll introduce it in this lecture. It’s important to master this material, because we use it a lot in later lectures.

You know about method specifications and class invariants. Now comes the loop invariant.

Next recitation: Loop invariants

In JavaHyperText, click on link Loop invariants in the horizontal navigation bar. Watch the videos on that page and the second page, 2. Practice on developing parts of loops.

There will be a short quiz on Loop invariants and a problem set to do during recitation.

We now introduce the topic.

Next recitation: Loop invariants

Next recitation: Loop invariants

// store in s the sum of the elements in array b.
int k= 0; s= 0;
while (k < b.length) {
    s= s + b[k];
k= k+1;
}
// R: s = sum of b[0..b.length-1]

3   2   5   1
0   1   2   3   4
when done, s = 11

3   2   5   1
0   1   2   3   4

This will be true before and after each iteration

Without giving meaning to variables, the only way you can tell this works is by executing it in your head, see what it does on a small array. A loop invariant will give that meaning.

Loopy question 1: Does init truthify P?

This will be true before and after each iteration

P: b
k
k
s is sum of these
b.length

This will be true before and after each iteration

P: b
k
k
s is sum of these
b.length
Loopy question 2: Is R true upon termination?

```java
int k = 0; s = 0;
// invariant P: s = sum of b[0..k-1]
while (k < b.length) {
    s = s + b[k];
    k = k + 1;
}
// R: s = sum of b[0..b.length-1]
```

This will be true before and after each iteration

<table>
<thead>
<tr>
<th>b</th>
<th>s</th>
<th>k</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

P: b is sum of these

Loopy question 3: Does repetend make progress toward termination?

```java
int k = 0; s = 0;
// invariant P: s = sum of b[0..k-1]
while (k < b.length) {
    s = s + b[k];
    k = k + 1;
}
// R: s = sum of b[0..b.length-1]
```

This will be true before and after each iteration

<table>
<thead>
<tr>
<th>b</th>
<th>s</th>
<th>k</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

P: b is sum of these

Loopy question 4: Does repetend keep invariant true?

```java
int k = 0; s = 0;
// invariant P: s = sum of b[0..k-1]
while (k < b.length) {
    s = s + b[k];
    k = k + 1;
}
// R: s = sum of b[0..b.length-1]
```

This will be true before and after each iteration

<table>
<thead>
<tr>
<th>b</th>
<th>s</th>
<th>k</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

P: b is sum of these

All four loopy questions checked. Loop is correct.

```java
int k = 0; s = 0;
// invariant P: s = sum of b[0..k-1]
while (k < b.length) {
    s = s + b[k];
    k = k + 1;
}
// R: s = sum of b[0..b.length-1]
```

Use of invariant allows us to break loop (and init) into parts and handle them independently.

Initialization? Look only at possible precondition of algorithm and loop invariant

Termination? Look only at loop invariant, loop condition, postcondition.

To Understand Recursion...
Recursion – Real Life Examples

<noun phrase> is <noun>, or
<noun phrase>, or
<adjective> <noun phrase>, or
<adverb> <noun phrase>

Example:
terrible horrible no-good very bad day

Recursion – Real Life Examples

<noun phrase> is <noun>, or
<noun phrase>, or
<adjective> <noun phrase>, or
<adverb> <noun phrase>

ancestor(p) is parent(p), or
parent(ancestor(p))
great great great great great great great great great great great great great great great great great great
great great grandmother.

0! = 1
n! = n * (n - 1)!
1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 39916800, 479001600…

Sum the digits in a non-negative integer

```java
/** = sum of digits in n.
 * Precondition: n >= 0 */
public static int sum(int n) {
    if (n < 10) return n;
    // { n has at least two digits }
    // return first digit + sum of rest
    return n%10 + sum(n/10);
}
```

sum(7) = 7
sum(8703) = 3 + sum(870)
= 3 + 8 + sum(70)
= 3 + 8 + 7 + sum(0)

Two different questions, two different answers

1. How is it executed?
(or, why does this even work?)

2. How do we understand recursive methods?
(or, how do we write/develop recursive methods?)

Stacks and Queues

Stack: list with (at least) two basic ops:
* Push an element onto its top
* Pop (remove) top element

Last-In-First-Out (LIFO)

Like a stack of trays in a cafeteria

Queue: list with (at least) two basic ops:
* Append an element
* Remove first element

First-In-First-Out (FIFO)

Americans wait in a line. The Brits wait in a queue!

Stack Frame

A “frame” contains information about a method call:
At runtime Java maintains a stack that contains frames for all method calls that are being executed but have not completed.
Method call: push a frame for call on stack. Assign argument values to parameters. Execute method body. Use the frame for the call to reference local variables and parameters.
End of method call: pop its frame from the stack; if it is a function leave the return value on top of stack.
Memorize method call execution!

A frame for a call contains parameters, local variables, and other information needed to properly execute a method call.

To execute a method call:
1. push a frame for the call on the stack,
2. assign argument values to parameters,
3. execute method body,
4. pop frame for call from stack, and (for a function) push returned value on stack

When executing method body look in frame for call for parameters and local variables.

Frames for methods sum main method in the system

Example: Sum the digits in a non-negative integer

public static int sum(int n) {
    if (n < 10) return n;
    return n%10 + sum(n/10);
}

class Main {
    public static void main(String[] args) {
        int r = sum(824);
        System.out.println(r);
    }
}

Memorize method call execution!

Example: Sum the digits in a non-negative integer

public static int sum(int n) {
    if (n < 10) return n;
    return n%10 + sum(n/10);
}

class Main {
    public static void main(String[] args) {
        int r = sum(824);
        System.out.println(r);
    }
}

Example: Sum the digits in a non-negative integer

public static int sum(int n) {
    if (n < 10) return n;
    return n%10 + sum(n/10);
}

class Main {
    public static void main(String[] args) {
        int r = sum(824);
        System.out.println(r);
    }
}
Example: Sum the digits in a non-negative integer

```java
public static int sum(int n) {
    if (n < 10) return n;
    return n%10 + sum(n/10);
}

public static void main(String[] args) {
    int r = sum(824);
    System.out.println(r);
}
```

Poll time!
Two different questions, two different answers

1. How is it executed? (or, why does this even work?)
   It’s not magic! Trace the code’s execution using the method call algorithm, drawing the stack frames as you go.
   Use only to gain understanding / assurance that recursion works.

2. How do we understand recursive methods? (or, how do we write/develop recursive methods?)
   This requires a totally different approach.

---

How to understand what a call does

Make a copy of the method spec, replacing the parameters of the method by the arguments

```
/** = sum of the digits of n.
* Precondition: n >= 0 */
public static int sumDigs(int n) {
    if (n < 10) return n;  // n has at least two digits
    return n%10  + sumDigs(n/10);
}
```

Understanding a recursive method

Step 1. Have a precise spec!
Step 2. Check that the method works in the base case(s). That is, cases where the parameter is small enough that the result can be computed simply and without recursive calls.

```
/** = sum of the digits of n.
* Precondition: n >= 0 */
public static int sumDigs(int n) {
    if (n < 10) return n;  // n has at least two digits
    return n%10  + sumDigs(n/10);
}
```

Step 3. Look at the recursive case(s). In your mind replace each recursive call by what it does according to the method spec and verify that the correct result is then obtained.

```
return n%10 + sum(n/10);
return n%10 + (sum of digits of n/10);  // e.g. n = 843
```

Factorial function:

```
0! = 1
n! = n \times (n-1)! for n > 0
(e.g.: 4! = 4\times3\times2\times1=24)
```

Easy to make math definition into a Java function!

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

Exponentiation:

```
b^0 = 1
b^c = b \times b^{c-1} for c > 0
```

Easy to make math definition into a Java function!

```
public static int exp(int b, int c) {
    if (c == 0) return 1;
    return b * exp(b, c-1);
}
```

Understanding a recursive method

Step 1. Have a precise spec!
Step 2. Check that the method works in the base case(s).
Step 3. Look at the recursive case(s). In your mind replace each recursive call by what it does acc. to the spec and verify correctness.

```
/** = sum of the digits of n.
* Precondition: n >= 0 */
public static int sumDigs(int n) {
    if (n < 10) return n;  // n has at least two digits
    return n%10 + sumDigs(n/10);
}
```

Step 4. (No infinite recursion) Make sure that the args of recursive calls are in some sense smaller than the pars of the method.

```
n/10 < n, so it will get smaller until it has one digit
```
Understanding a recursive method

Step 1. Have a precise spec!

Step 2. Check that the method works in the base case(s).

Step 3. Look at the recursive case(s). In your mind replace each recursive call by what it does according to the spec and verify correctness.

Step 4. (No infinite recursion) Make sure that the args of recursive calls are in some sense smaller than the parameters of the method

Writing a recursive method

Step 1. Have a precise spec!

Step 2. Write the base case(s): Cases in which no recursive calls are needed. Generally for “small” values of the parameters.

Step 3. Look at all other cases. See how to define these cases in terms of smaller problems of the same kind. Then implement those definitions using recursive calls for those smaller problems of the same kind. Done suitably, point 4 (about termination) is automatically satisfied.

Step 4. (No infinite recursion) Make sure that the args of recursive calls are in some sense smaller than the parameters of the method

Two different questions, two different answers

2. How do we understand recursive methods? (or, how do we write/develop recursive methods?)

Step 1. Have a precise spec!

Step 2. Check that the method works in the base case(s).

Step 3. Look at the recursive case(s). In your mind replace each recursive call by what it does according to the spec and verify correctness.

Step 4. (No infinite recursion) Make sure that the args of recursive calls are in some sense smaller than the parameters of the method

Examples of writing recursive functions

For the rest of the class we demo writing recursive functions using the approach outlined below. The java file we develop will be placed on the course webpage some time after the lecture.

Step 1. Have a precise spec!

Step 2. Write the base case(s).

Step 3. Look at all other cases. See how to define these cases in terms of smaller problems of the same kind. Then implement those definitions using recursive calls for those smaller problems of the same kind.

Step 4. Make sure recursive calls are “smaller” (no infinite recursion).

Check palindrome-hood

A String palindrome is a String that reads the same backward and forward:

```java
isPall("racecar") -> true  
isPall("pumpkin") -> false
```

A String with at least two characters is a palindrome if:

- (0) its first and last characters are equal and
- (1) chars between first & last form a palindrome:
  ```java
e.g.  AMANAPLANACANALPANA
```

A recursive definition!
Example: Is a string a palindrome?

```java
/** = "s is a palindrome" */
public static boolean isPal(String s) {
    if (s.length() <= 1)
        return true;
    // { s has at least 2 chars }
    int n = s.length() - 1;
    return s.charAt(0) == s.charAt(n) && isPal(s.substring(1, n));
}
```

Example: Count the e's in a string

```java
/** = number of times c occurs in s */
public static int countEm(char c, String s) {
    if (s.length() == 0)
        return 0;
    // { s has at least 1 character }
    if (s.charAt(0) != c)
        return countEm(c, s.substring(1));
    // { first character of s is c }
    return 1 + countEm(c, s.substring(1));
}
```

The Fibonacci Function

Mathematical definition:
- fib(0) = 0
- fib(1) = 1
- fib(n) = fib(n-1) + fib(n-2)  n ≥ 2

Fibonacci sequence: 0 1 1 2 3 5 8 13 ...

```java
/** = fibonacci(n). Pre: n >= 0 */
static int fib(int n) {
    if (n <= 1)
        return n; // { 1 < n }
    return fib(n-1) + fib(n-2);
}
```

Example: Count the e’s in a string

- countEm('e', "it is easy to see that this has many e’s") = 4
- countEm('e', "Mississippi") = 0