Overview references to sections in text

- Note: We've covered everything in JavaSummary.pptx!
- What is recursion? 7.1-7.39 slide 1-7
- Base case 7.1-7.10 slide 13
- How Java stack frames work 7.8-7.10 slide 28-32

Function equals

```java
public class Object {
    /** Return true iff this object is the same as ob */
    public boolean equals(Object b) {
        return this == b;
    }
}
```

This gives a null-pointer exception:

```
x.equals(y) is same as x == y except when x is null!
```

Overriding function equals

Override function equals in a class to give meaning to:
“these two (possibly different) objects of the class have the same values in some of their fields”

For those who are mathematically inclined, like any equality function, equals should be reflexive, symmetric, and transitive.

Reflexive: b.equals(b)
Symmetric: b.equals(c) = c.equals(b)
Transitive: if b.equals(c) and c.equals(d), then b.equals(d)

Function equals in class Animal

```java
public class Animal {
    /** = "h is an Animal with the same values in its fields as this Animal" */
    public boolean equals(Object h) {
        if (!(h instanceof Animal))
            return false;
        Animal ob = (Animal) h;
        return name.equals(ob.name) && age == ob.age;
    }
}
```

1. Because of h is an Animal in spec, need the test h instanceof Animal

2. In order to be able to reference fields in partition Animal, need to cast h to Animal
### Function equals in class Animal

```java
public class Animal {
    /** h is an Animal with the same values in its fields as this Animal */
    public boolean equals(Object h) {
        if (!(h instanceof Animal)) return false;
        Animal ob = (Animal) h;
        return name.equals(ob.name) && age == ob.age;
    }
    public String toString() {
        // ... return the string representation of the object.
    }
}
```

3. Use `String equals` function to check for equality of `String` values. Use `==` for primitive types.

### Why can’t the parameter type be Animal?

```java
public class Animal {
    /** h is an Animal with the same values in its fields as this Animal */
    public boolean equals(Animal h) {
        if (!(h instanceof Animal)) return false;
        Animal ob = (Animal) h;
        return name.equals(ob.name) && age == ob.age;
    }
    public String toString() {
        // ... return the string representation of the object.
    }
}
```

What is wrong with this?

### Sum the digits in a non-negative integer

```java
/** return 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 sum(n/10) + n%10;
}
```

E.g. `sum(7) = 7`

E.g. `sum(8703) = sum(870) + 3;`

### Two issues with recursion

1. Why does it work? How does the method executed?

2. How do we understand a given recursive method or how do we write/develop a recursive method?

### 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)

**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 parameters.
  - End of method call: pop its frame from the stack; if it is a function leave the return value on top of stack.
Frames for methods sum main method in the system

Frame for method in the system that calls method main

Example: Sum the digits in a non-negative integer

Example: Method main calls sum: main is then called

Example: Sum the digits in a non-negative integer

Example: Sum the digits in a non-negative integer

Example: Sum the digits in a non-negative integer
Example: Sum the digits in a non-negative integer

```java
public static int sum(int n) {
    if (n < 10) return n;
    return sum(n/10) + n%10;
}
public static void main(String[] args) {
    int r = sum(824);
    System.out.println(r);
}
```

Using return value 8 stack computes 8 + 2 = 10 pops frame from stack puts return value 10 on stack

Example: Sum the digits in a non-negative integer

```java
public static int sum(int n) {
    if (n < 10) return n;
    return sum(n/10) + n%10;
}
public static void main(String[] args) {
    int r = sum(824);
    System.out.println(r);
}
```

Using return value 10 stack computes 10 + 4 = 14 pops frame from stack puts return value 14 on stack

Example: Sum the digits in a non-negative integer

```java
public static int sum(int n) {
    if (n < 10) return n;
    return sum(n/10) + n%10;
}
public static void main(String[] args) {
    int r = sum(824);
    System.out.println(r);
}
```

Using return value 14 main stores 14 in r and removes 14 from stack

Summary of method call execution

- Memorize this!
- 1. A frame for a call contains parameters local variables and other information needed to properly execute a method call.
- 2. To execute a method call: push a frame for the call on the stack, assign values to parameters, execute method body, 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.

Questions about local variables

```java
public static void m(...) {
    ...
    while (...) {
        int d = 5;
        ...
    }
}
```

In a call `m()` when is local variable `d` created and when is it destroyed? Which version of procedure `m` do you like better? Why?

Recursion is used extensively in math

Math definition of n factorial

\[ n! = 1 \quad \text{for } n > 0 \]

\[ n! = n \cdot (n-1)! \quad \text{for } n > 0 \]

E.g. 3! = 3*2*1 = 6

Easy to make math definition into a Java function!

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

Math definition of b^c for c >= 0

\[ b^0 = 1 \]

\[ b^c = b \cdot b^{c-1} \quad \text{for } c > 0 \]

Lots of things defined recursively: expression grammars trees ….

We will see such things later
Two views of recursive methods

- How are calls on recursive methods executed? We saw that. Use this only to gain understanding/assurance that recursion works.
- How do we understand a recursive method — know that it satisfies its specification? How do we write a recursive method?
This requires a totally different approach. Thinking about how the method gets executed will confuse you completely! We now introduce this approach.

Understanding a recursive method

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

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

Example:

```java
public static int sum(int n) {
    if (n < 10) return n; // n has at least two digits
    return sum(n/10) + n%10;
}
```
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.

**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.

---

**Check palindrome-hood**

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

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
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));
}
```

A recursive definition!

---

**The Fibonacci Function**

**Mathematical definition:**

\[
\begin{align*}
\text{fib}(0) & = 0 \\
\text{fib}(1) & = 1 \\
\text{fib}(n) & = \text{fib}(n - 1) + \text{fib}(n - 2) \quad n \geq 2
\end{align*}
\]

Fibonacci sequence: 0 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-2) + fib(n-1);
}
```

**Statue in Pisa Italy**

Giovanni Paganucci 1863

**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));
}
```

isPal("racecar") returns true
isPal("pumpkin") returns false

---

**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;
    // { first character of s is c }
    if (s.charAt(0) != c) return countEm(c, s.substring(1));
    return 1 + countEm(c, s.substring(1));
}
```

**Example: The Fibonacci Function**

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

**Statue in Pisa Italy**

Giovanni Paganucci 1863