ICS 111
Recursive Methods

- Example of recursive call
- Recursive methods
- Recursive thinking

ICS 111
Making Problems Smaller

- Suppose you had the task of eating a stack of pancakes
- It might seem like a huge task
- In the spirit of stepwise refinement, you might say “I’ll eat the first pancake, then I’ll eat the rest”
- So first you call the method to eat the first pancake, then you call another method to eat the rest of the pancakes

ICS 111
Making Problems Smaller: Java methods

```
public static void eatStackOfPancakes() {
  if (pancakesExist()) {
    eatTopPancake();
    eatStackOfPancakes();
  }
}
```

- the method to eat the rest of the pancakes is the same as the method to eat all the pancakes!
- Here we have a method calling itself – known as a **recursive** call
- Java (and most other programming languages) supports recursive calls
- a method calling itself recursively is essentially the same as a method calling another method

ICS 111
Making problems smaller

- As we solve part of a problem, sometimes the rest of the problem is just a smaller instance of the original problem
- Examples in the book:
  - cleaning a house
  - printing a triangle
- To solve the smaller problem, why not use the same method as we used to solve the bigger problem?

ICS 111
converting an integer to a string

- in Java, concatenating an integer to a string automatically converts the integer to a string
- what if we didn’t have that?
- it’s easy enough to convert the last digit (the integer modulo 10) to a string
- so all we need is to convert to a string the integer divided by 10

ICS 111
converting an integer to a string

```
public static String intToString (int value) {
  String higherDigits = "";
  if (value >= 10) {
    higherDigits = intToString (value / 10);
  }
  return higherDigits + digitToString(value % 10);
}
```

- the integer stored in `value` is different for each of the recursive calls
  - if my `value` is 4567, in the intToString that I call, `value` will be 456
- we still need to define digitToString

ICS 111
converting a digit to a string

```
public static String digitToString (int value) {
  switch (value) {
  case 0: return "0";
  case 1: return "1";
  case 2: return "2";
  case 3: return "3";
  case 4: return "4";
  case 5: return "5";
  case 6: return "6";
  case 7: return "7";
  case 8: return "8";
  case 9: return "9";
  }
  return "illegal parameter " + value + " to digitToString";
}
```

- each case ends with return, no need for break statements

ICS 111
tracing intToString

```
public static String intToString (int value) {
  String higherDigits = "";
  if (value >= 10) {
    higherDigits = intToString (value / 10);
  }
  return higherDigits + digitToString(value % 10);
}
```

- we call `intToString(543)`
  - value is 543 >= 10, so we call `intToString(54)`
    - value is 54 >= 10, so we call `intToString(5)`
      - value is 5, not >= 10, so we return `"5"`
    - higherDigits "5" + digitToString(4) returns `"54"`
  - higherDigits "54" + digitToString 3 returns `"543"`
- calls and returns are nested like russian dolls

ICS 111
designing recursive methods

- Like a loop, a recursive method does “the same thing” over and over again
- Like a loop, a recursive method must stop at some point
  - infinite recursion causes stack overflow!
- Each recursive call should solve a smaller version of the problem
  - and stop once the problem is small enough to be solved directly (such as `digitToString`)
  - the “size” of a problem is defined by the programmer’s understanding
  - but in any case, there must be at least one stopping condition
  - and each recursive call must get “closer” to this stopping condition
- So every recursive call must be executed conditionally, that is, only until the problem is small enough to solve directly

ICS 111
A mathematical example

- The factorial function (written “!”) is defined as:
  - 1! = 1
  - n! = n * (n – 1)! for n > 1
- this is a recursive definition
  - like many mathematical definitions
- so the recursive implementation is straightforward:

```
public static int factorial(int n) {
  if (n <= 1) {
    return 1;
  }
  return n * factorial(n – 1);
}
```

- Note that factorial can also be implemented as a loop
  - implementing it as a loop would be described as an **iterative** (rather than recursive) implementation

ICS 111
Recursive Thinking

- As we solve part of a problem, sometimes the rest of the problem is just a smaller instance of the original problem
- So we have to figure out:
  - which part of the work do we take care of here, vs which part of the work is in the smaller instance of the problem?
  - when do we stop the recursion?
- This is not too different from thinking about loops
- In fact, loops and recursion are theoretically equivalent

Summary

- Recursion is indicated when, in stepwise refinement, one of the subproblems is a smaller instance of the bigger problem
- A method can call another method, or it can call itself – there is no difference between the two types of calls
- If a method calls itself, it must do so conditionally in order to avoid infinite recursion