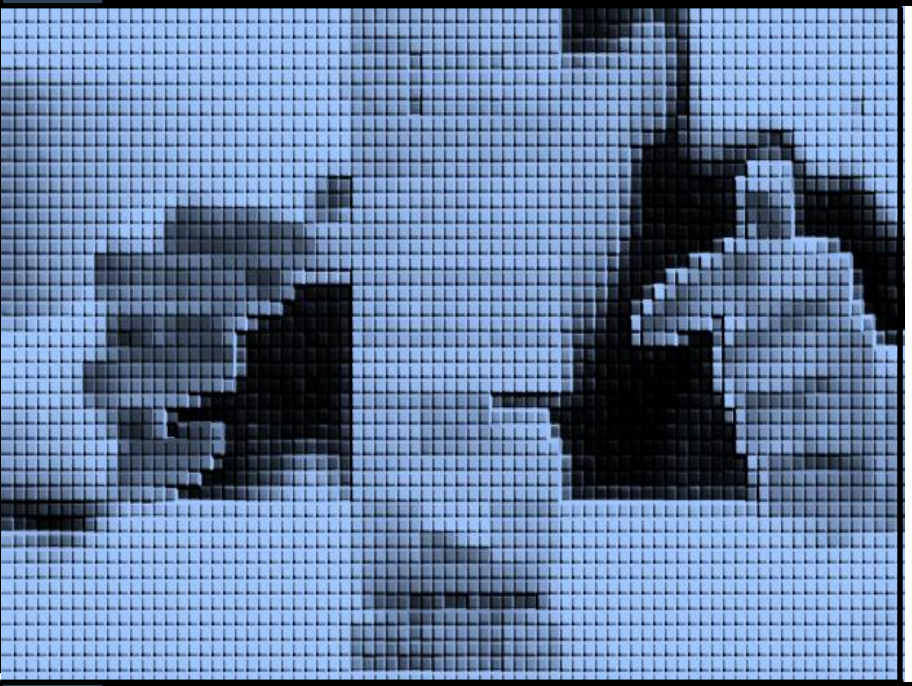
University of BRISTOL

School of Computer Science

Programming in Java

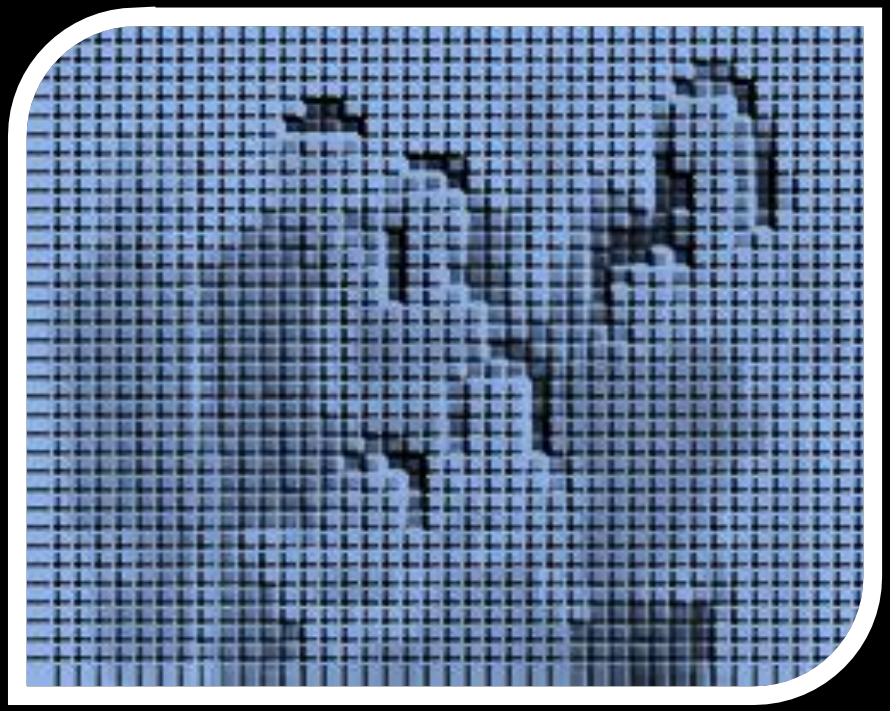


Programming in Java – Lecture 03

Decisions & Recursion

Sion Hannuna and Simon Lock,
based on Tilo Burghardt's C Unit

Relations



Relational Expressions

- We can put arithmetic expressions in *relation to each other* and interpret them as true or false, for instance: `(x < y)`
- A relational expression *produces* either the value 0 (false) or 1 (true).
- Any *non-zero* value is interpreted as *true*.
- Values of *exactly 0* are interpreted as *false*.
- Any relation can thus be interpreted as either true or false.

OPERATION	OPERATOR
less than	<
greater than	>
less than or equal to	<=
greater than or equal to	>=
equal to	==
not equal to	!=
not	!
logical AND	&&
logical OR	||

Evaluation of Expressions as True/False

```
                0 //false
          (0 || 1) //true
        (15 < 18) //true
((15 + 4) < 18) //false
           (37) //true
          (!21) //false
((1 - 1) && 21) //false
      (11 != 11) //false
((1 - 1) || 11) //true
 ((1 - 1) == 0) //true

        (x = 5) //usually a bug, but true
        (x = 0) //usually a bug, but false
```

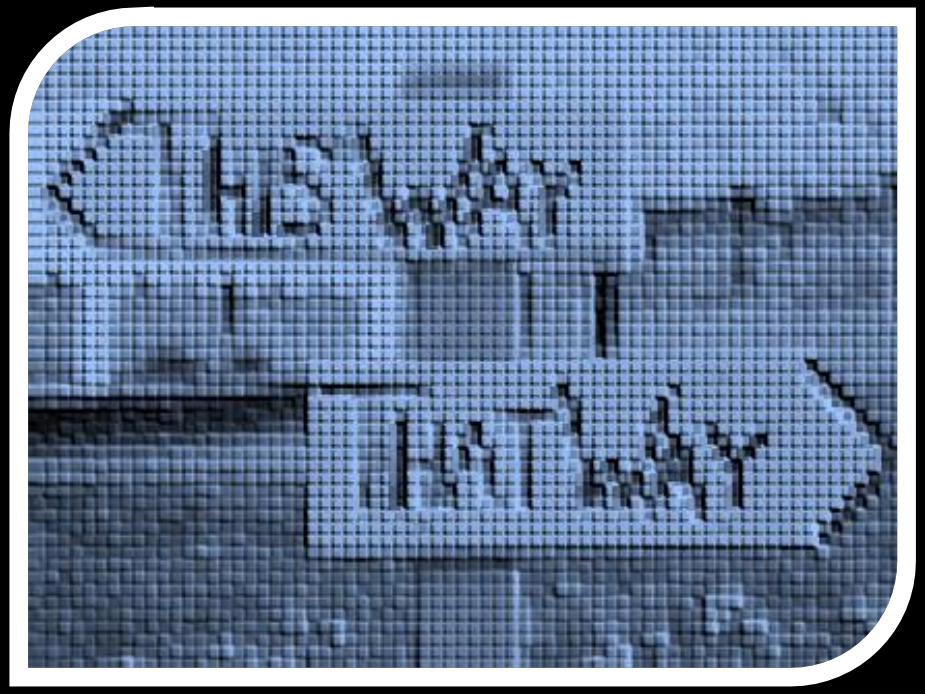
RULES:

A relational expression *produces* either the value 0 (false) or 1 (true).

Any *non-zero* value is interpreted as *true*.

Values of *exactly 0* are interpreted as *false*.

Decisions



Conditionals

- Decisions can be made using `if` and `else` where following statements are executed dependent on the true/false evaluation of an expression (the `else`-branch is optional):

```
if (EXPRESSION) STATEMENT (executed if true)
else STATEMENT (executed if false)
```

- Example A:

```
...
// Return the smaller integer.
public int minimum(int x, int y) {
  int min;
  if (x < y) min = x;
  else min = y;
  return min;
} ...
```

- Example B using a block {…}:

```
...
// Get next hailstone number.
public int nextHailstone(int x) {
  int next;
  if (x % 2 == 1) {
    int base = 3 * x;
    next = base + 1;
  }
  else next = x / 2;
  return next;
} ...
```

Conditionals with Blocks

```
...
  if (x % 2 == 1) {
    int base = 3 * x;
    next = base + 1;
  }
...
```

- An `if` statement (or an `else` statement for that matter) can be followed by a *block* instead of a single statement.
- Remember: a block is a sequence of statements between curly braces, the same as we know from a function body.
- Note that `else` is optional – if not used then the default is 'else do nothing'.
- Note that the *scope* of the local variable `base` is the block – thus, it doesn't exist outside!

Minimum

```java
import java.util.Scanner;

public class MinimumNumber {

   public static int minimum(int x, int y) {
      int min;
      if (x < y) {
         min = x;
      } else {
         min = y;
      }
      return min;
   }

   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      System.out.println("Enter x:");
      int x = scanner.nextInt();
      System.out.println("Enter y:");
      int y = scanner.nextInt();
      System.out.println("The smaller number is: " + minimum(x, y));
      scanner.close();
   }
}
```

MinimumNumber.java

This code reads two integers from the user and prints the smaller one. It uses the Scanner class for input, and the minimum method to compare the integers. The main method handles input, calls minimum, prints the result, and closes the Scanner to free resources.

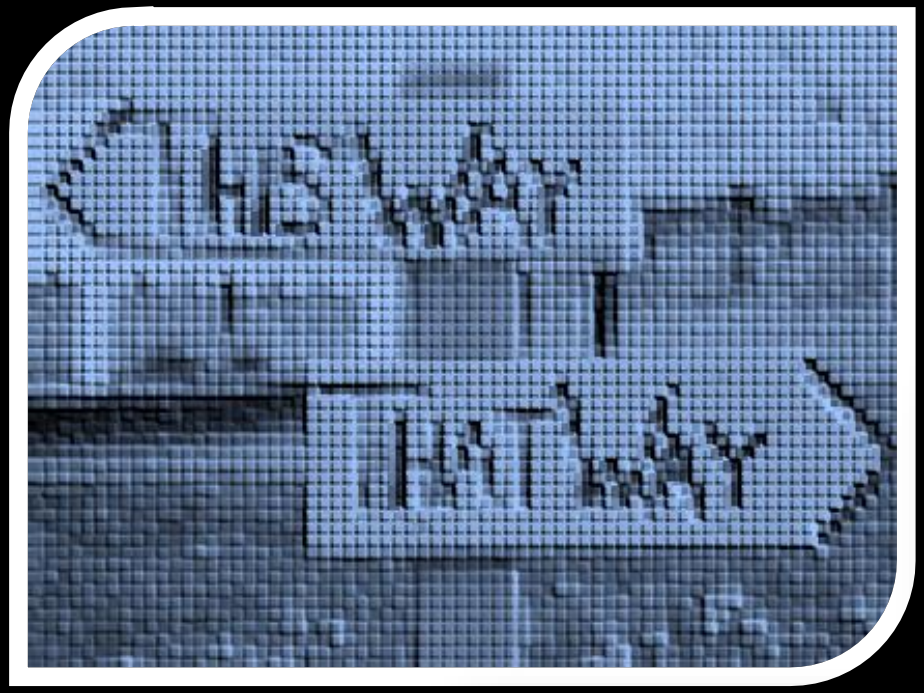
Next Hailstone

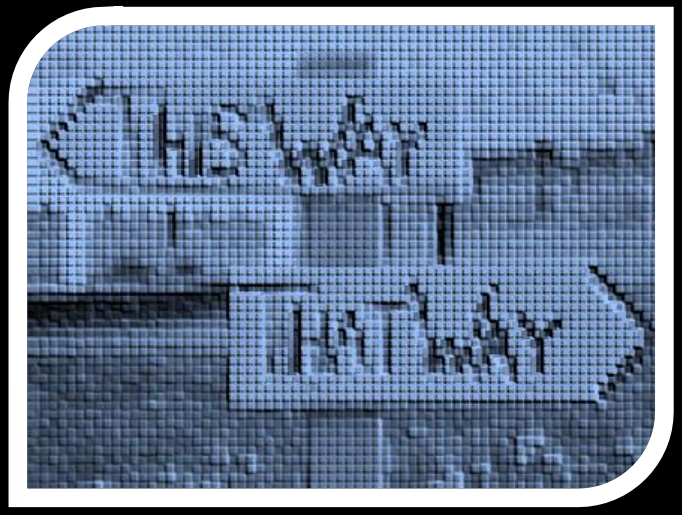
- For a given integer seed n, the next hailstone number is n/2 if n is even, or 3n+1 otherwise:

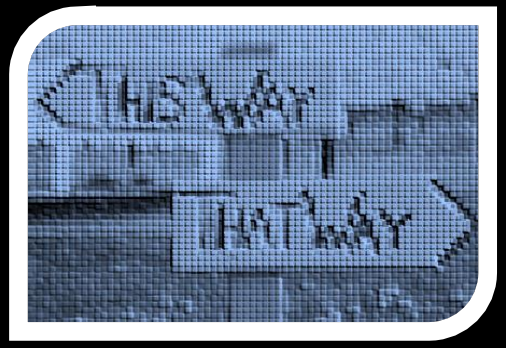
```
public class Hailstone {
   public static int nextHailstone(int x) {
      int next;
      if (x % 2 == 1) {
         int base = 3 * x;
         next = base + 1;
      } else {
         next = x / 2;
      }
      return next;
   }

   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      System.out.println("Enter seed:");
      int n = scanner.nextInt();
      System.out.println("The next hailstone number is: " + nextHailstone(n));
      scanner.close();
   }
}
```

Hailstone.java

Multiple Decision Points







Conditional Chains

- More complicated decisions can be made using `else if` statements with various decision points:

```
if (EXPRESSION1) STATEMENT
else if (EXPRESSION2) STATEMENT ...
else STATEMENT
```

- Example where a mark is interpreted to calculate a grade:

```
...
/* Transform mark into grade. */
public static int grade(int mark) {
  int grade;
  if (mark >= 70) grade = 1;
  else if (mark >= 50) grade = 2;
  else if (mark >= 40) grade = 3;
  else grade = 4;
  return grade;
} ...
```

Expression examples: Using ! (Logical not)

The **‘!’** operator **inverts** the value of a boolean expression:

```
public class LogicalNotExample {
    public static void main(String[] args) {
        boolean isRaining = false;

        // Using ! to invert the boolean value
        if (!isRaining) {
            System.out.println("It is not raining. You can go outside without an umbrella.");
        } else {
            System.out.println("It is raining. You need an umbrella.");
        }
    }
}
```

LogicalNotExample.java

Expression examples: Using || (Logical OR)

The **||** operator returns true if **at least one** of the conditions is true.

```java
public class LogicalOrExample {
    public static void main(String[] args) {
        int age = 16;
        boolean hasParentalConsent = true;

        // Using || to check if at least one condition is true
        if (age >= 18 || hasParentalConsent) {
            System.out.println("You can watch the movie.");
        } else {
            System.out.println("You cannot watch the movie.");
        }
    }
}
```

LogicalOrExample.java

Expression examples: Using && (Logical AND)

The **&&** operator returns true if **all** of the conditions is true.

```
public class LogicalAndExample {
    public static void main(String[] args) {
        int age = 20;
        boolean hasID = true;

        // Using && to check if both conditions are true
        if (age >= 18 && hasID) {
            System.out.println("You are allowed to enter the club.");
        } else {
            System.out.println("You are not allowed to enter the club.");
        }
    }
}
```

LogicalAndExample.java

Expression examples: Combined Example

Combining **!**, **||**, and **&&** in a single conditional statement.

```
public class CombinedLogicalExample {
    public static void main(String[] args) {
        boolean isWeekend = true;
        boolean hasHomework = false;
        boolean isHoliday = false;

        // Using !, ||, and && together
        if (isWeekend && (!hasHomework || isHoliday)) {
            System.out.println("You can relax and enjoy your day.");
        } else {
            System.out.println("You need to finish your homework.");
        }
    }
}
```

CombinedLogicalExample.java

Expression examples - explanations

- **Logical NOT (`!`)**: Inverts the boolean value.
  - `!isRaining` is true if `isRaining` is false.
- **Logical OR (`||`)**: Returns true if at least one condition is true.
  - `age >= 18 || hasParentalConsent` is true if either the age is at least 18 or there is parental consent.
- **Logical AND (`&&`)**: Returns true only if both conditions are true.
  - `age >= 18 && hasID` is true if the age is at least 18 and there is an ID.
- **Combined Example**: Demonstrates using all three operators together to create more complex conditions.
  - `isWeekend && (!hasHomework || isHoliday)` is true if it is the weekend and either there is no homework or it is a holiday.

Grade Program

```java
import java.util.Scanner;

public class GradeCalculator {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter your mark:");
        int mark = scanner.nextInt();
        System.out.println("Your grade is: " + grade(mark));
    }
    public static int grade(int mark) {
        int grade;
        if (mark >= 70) {
            grade = 1;
        } else if (mark >= 50) {
            grade = 2;
        } else if (mark >= 40) {
            grade = 3;
        } else {
            grade = 4;
        }
        return grade;
    }
}
```

GradeCalculator.java

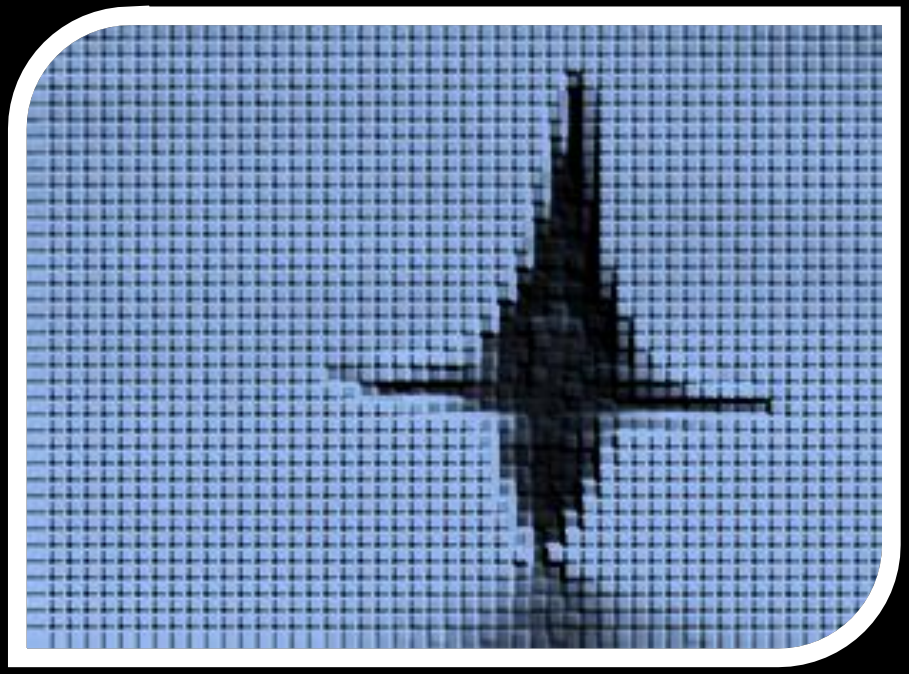
Shadowing

- In our `grade` procedure we declare a second identifier of the same name (i.e. the local variable `grade`):

```
...
public static int grade(int mark) {
  int grade;  //cannot call grade(…) in this procedure
  ...
```

- The procedure identifier has *global scope*, whilst the variable grade has a *local scope* limited to this procedure only.
- In such situations the identifier declared last (innermost scope) takes precedence and all other identifiers of the same name are temporarily not accessible or *shadowed*.
- We are usually *not* allowed to *declare* the same identifier name *twice* in exactly the same scope (although see overloading later).

Recursion



Triangle Numbers

- The n^{th} triangle number is the sum of numbers from 1 to n:

```
import java.util.Scanner;

public class TriangleNumber {
   public static int sum(int n) {
      if (n == 1) return 1;
      else return n + sum(n - 1);
   }
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      System.out.println("Enter n:");
      int n = scanner.nextInt();
      System.out.printf("The %dth triangle number is: %d\n", n, sum(n));
   }
}
```

TriangleNumber.java

A Self-Calling Procedure

- The important part of the program is the `sum` function:

```
...
public static int sum(int n) {
  if (n == 1) return 1;
  else return n + sum(n - 1);
}
...
```

- It has one argument variable n.
- Arguments have local scope: they are *created* at the start of a call, and *destroyed* when the function returns.
- Intriguingly, the function is *recursive*, i.e. it calls itself.
- It decreases the argument by 1 every time the self-call happens.
- Once the argument is down to 1 – as checked in the `if` statement – it *stops* calling itself and just `return`s 1.

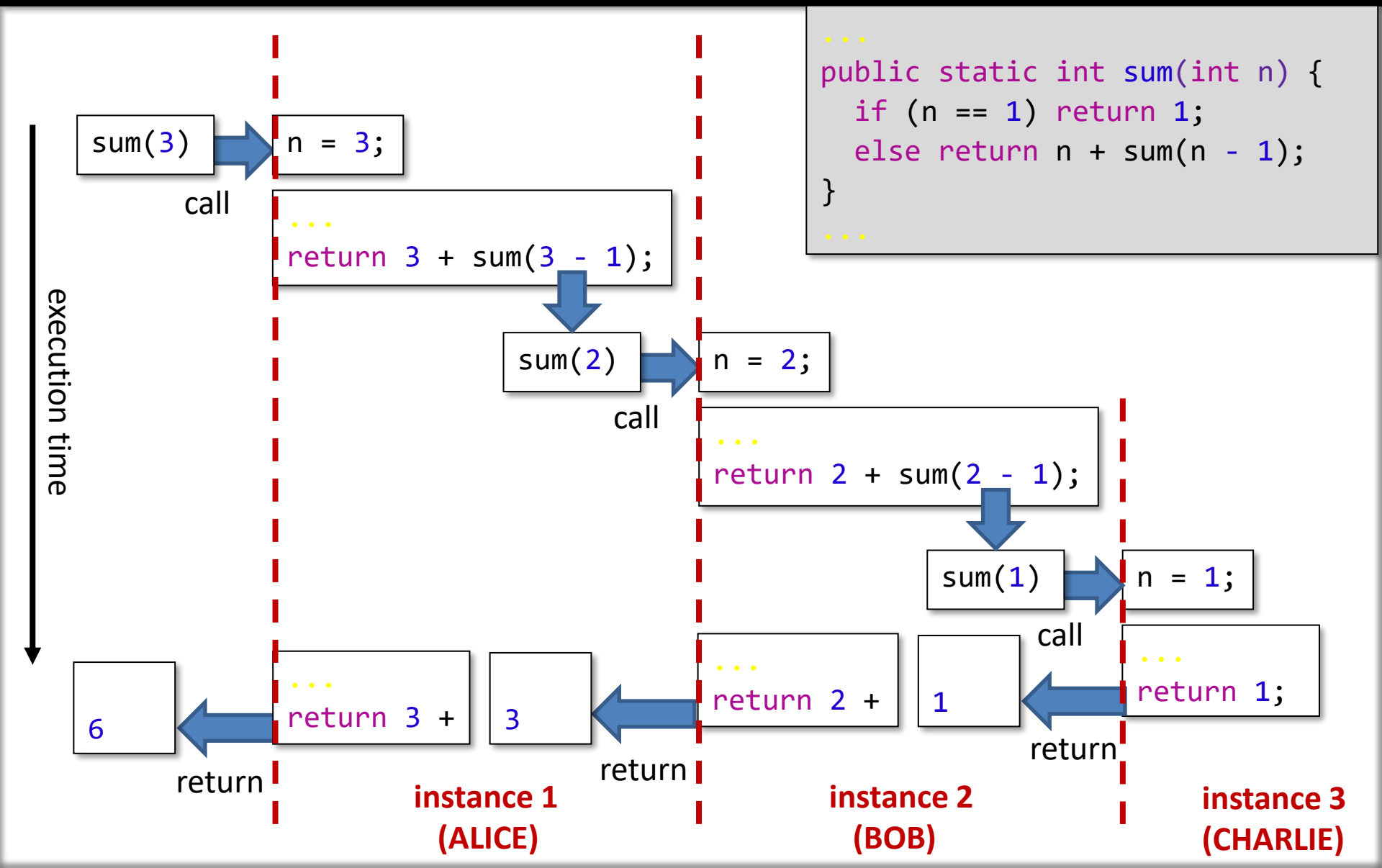
Analogy: A Row of Friends

- To get the hang of recursion, imagine a *row of friends* who co-operate in solving the problem.
- Each friend *keeps track of their own copy* of local variables and arguments (i.e. just n) on paper and has a *copy of the instructions*:

```
public static int sum(int n) {
  if (n == 1) return 1;
  else return n + sum(n - 1);
}
```

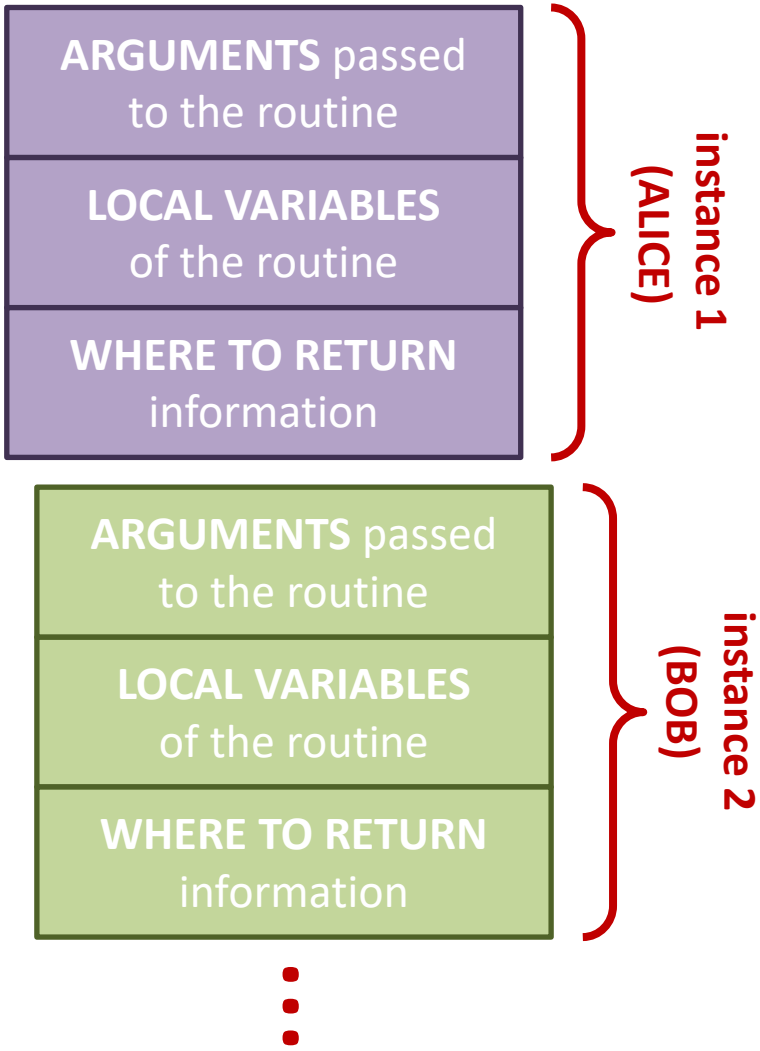
- Say, the main function calls `sum(3)`, which is like handing the problem to one friend, let's say Alice, who writes `n = 3` on her piece of paper...
- Alice then checks that she has to `return n + sum(n - 1);` thus she has to add `3` to whatever the solution to the problem `sum(3-1)` is, which she hands to Bob, who writes `n = 2` on his piece of paper...

Recursive Call Sequence



Call Stack

- A processor has access to a *call stack*, containing stack frames, like a pile of pieces of paper in our example with local variables written on, one for each function call which is in progress (full details will follow later)
- The call stack is very *efficient*, especially since call and return instructions are built into the processor



Termination

```
...
public static int sum(int n) {
  if (n == 1) return 1;
  else return n + sum(n - 1);
}
...
```

- It is a good thing the sum procedure *doesn't always* call itself.
- Otherwise, there would be an unlimited chain of calls (often called an '*infinite loop*') and the program would keep going until it ran out of memory.
- Recursion always needs a termination condition, which is also known as the *anchor* of the recursion.

Hailstone Sequence

- For a given seed integer and step number n, calculate the n^{th} number in the associated hailstone sequence:

```
import java.util.Scanner;

public class Hailstone {
   // Return the n'th hailstone number given a seed
   public static int hailstone(int seed, int n) {
      int next;
      if (seed % 2 == 1) next = 3 * seed + 1;
      else next = seed / 2;
      if (n == 1) return next;
      else return hailstone(next, n - 1);
   }
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      System.out.println("Enter seed:");
      int seed = scanner.nextInt();
      System.out.println("Enter how many steps to go:");
      int n = scanner.nextInt();
      System.out.printf("The %dth hailstone number for seed %d is: %d\n", n, seed, hailstone(seed, n));
   }
}
```

Hailstone.java

Lecture 03 Summary

IN THIS LECTURE WE COVERED:

- *types, relational expressions, if-else statement, blocks,*
  else-if statements, tracing, switch statemen
  break, default case, fall through, pro
  shadowing, recursion, call stack, anc
  basic program design rules

NEXT LECTURE: Loops and Arrays