1 Course Overview

- Even though Moore's Law[1] is still valid, heat and power are of primary concerns.
  - These challenges can be overcome with smaller and more efficient processors or simply more processors
  - To make better use of the added computation power, parallelism is used.
- Parallel vs. Concurrent: In both cases, one of the difficulties is to actually determine which processes can overlap and which can't:
  - Concurrent: Focus on which activities may b executed at the same time (= overlapping execution)
  - Parallel: Overlapping execution on a real system with constraints imposed by the execution platform.
- Parallel/Concurrent vs. distributed: In addition to parallelism/concurrency, systems can actually be physically distributed (e.g. BOINC)
- Concerns in PP:
  - Expressing Parallelism
  - Managing state (data)
  - Controlling/coordinating parallel tasks and data

2 Parallel Architectures

- Turing machine:
  - Infinite tape
  - Head that reads/writes symbols on tape
  - State registers
  - Program is expressed as rules: (reg)(head) $\rightarrow$ (reg)(head)(movement)
- Today's computers:
  - Consist of CPU, memory and I/O
  - Stored Program: program instructions are stored in memory
  - Von Neumann Architecture: Program data and program instruction in the same memory
- Since accessing memory became slower than accessing CPU registers, CPUs now have caches which are closer (faster and smaller) to the CPU. Caches are:
  - Faster then memory
  - Smaller than memory

[1] "The number of transistors on integrated circuits doubles approximately every two years"

  - Organized in multi-level hierarchies (e.g. L1,L2,L3)

- To improve sequential processor performance, you can use the following parallelism techniques:
  - Vectorization
    For example, when adding vectors (load $\rightarrow$ operation(s) $\rightarrow$ store)
    - Normal: 1-at-a-time
    - Vectors: N-at-a-time (bigger registers)
  - Pipelining[2]

    maybe add diagram from slides?

    - Multiple stages (CPU Functional Units)
      - Instruction Fetch
      - Instruction Decode
      - Execution
      - Data access
      - Writeback
    - Each instruction takes 5 time units (cycles)
    - 1 instruction per cycle (not always possible though)
  - Instruction Level Parallelism (ILP)
    - Superscalar CPUs
      - Multiple instructions per cycle
      - multiple functional units
    - Out-of-Order (OoO) Execution
      - Potentially change execution order of instructions
      - As long as the programmer observes the sequential program order
    - Speculative execution
      - Predict results to continue execution

- Moore's Law
  - *"The number of transistors on integrated circuits doubles approximately every two years"* - Gordon E.Moore, 1965
  - Actually an observation
  - For a long time, CPU Architects improved sequential execution by exploiting Moore's Law and ILP
  - More transistors $\rightarrow$ more performance
  - Sequential programs were becoming exponentially faster with each new CPU
    - (most) programmers did not worry about performance
    - They waited for the next CPU model

[2]Think laundry: you can either wash, dry, fold and repeat, or while the n load is drying, the $n+1$ load can start washing

- Architects hit walls
  - Power dissipation wall: Making CPU faster $\rightarrow$ expensive to cool
  - Memory Wall: CPUs faster than memory access
  - ILP Wall: Limits in inherent program's ILP, complexity
  - **No longer affordable to increase sequential CPU performance**
- Multicore processors
  - Use transistors to add cores (instead of improving sequential performance)
  - Expose parallelism to software
  - Implication: Programmers need to write parallel programs to take advantage of new hardware
    - Past: Parallel programming was performed by a few
    - Now: Programmers need to worry about (parallel) performance
- Shared memory architectures

  Maybe add picture form slides, page 34 form 2 - parallel architectures; ADD PICTURES FOR EACH OF THE DIFFERENT TYPES OF SMA

  - SMT (Hyperthreading)
    - Single core
    - Multiple instruction streams (threads); Virtual (phony) cores
    - Between ILP $\leftrightarrow$ multicore
      - ILP: Multiple units for one instruction stream
      - SMT: Multiple units for multiple instruction streams
    - Limited parallel performance
  - Multicores
    - Single chip, multiple cores
    - Dual-, Quad-, x8...
    - Each core has its own hardware units; computations un parallel perform well
    - Might share part of the cache hierarchy
  - SMP (Symmetric MultiProcessing)
    - Multiple CPUs on the same system
    - CPUs share memory: same cost to access memory
    - CPU caches coordinate: Cache coherence protocol
  - NUMA (Non-Uniform Memory Access)
    - Memory is distributed
    - Local/Remote (fast/slow)
    - Shared memory interface

- Flynn's Taxonomy

  Add diagram from page 49 of 2 - parallel architectures

- GPUs
  - Graphical Processing Units
    - Scene description $\rightarrow$ pixels
    - Highly data-parallel process
  - Massively parallel vector machines
  - Not a standard component up until recently
  - Started very specialized (rigid pipelines)
  - Driven by game industry success
- GP-GPUs
  - General programming using GPUs (CUDA, OpenCL)
  - Much research on "how to execute X on a GPU"
  - Generally GPUs are:
    - Well suited for data parallel programs
    - Not very well-suited for programs with random accesses
    - People are rethinking algorithms
  - GPUs are, currently, something of a standard in the HPC (High Performance Computing) domain

3 Basic Concepts

- Performance in sequential execution:
  - Computational complexity
    - Theoretical computer science
    - Asymptotic behavior: big O notation ($\mathcal{O}$), or big Θ
    - How many steps does an algortithm take
    - Complexity classes
  - Execution Time: The less time, the better
- Sequential programs are much easier to write, but if we care about performance we have to write parallel programs
- Parallel Performance
  - Sequential execution time: T_1
  - Execution time T_p on p CPUs:
    - $T_p = \frac{T_1}{p}$ (Perfection)
    - $T_p > \frac{T_1}{p}$ (Performance Loss, what normally happens)
    - $T_p < \frac{T_1}{p}$ (Sorcery!)

- (Parallel) Speedup
  - (Parallel) speedup S_p on p CPUs:

    $$S_p = \frac{T_1}{T_p}$$

    * $S_p = p \rightarrow$ linear speedup (Perfection)
    * $S_p < p \rightarrow$ sublinear speedup (Performance loss)
    * $S_p > p \rightarrow$ superlinear speedup (Sorcery!)
  - Efficiency: $\frac{S_p}{p}$
- Scalability: How well a system reacts to increased load
  - In Parallel Programming
    * Speedup when we increase processors
    * What will happen if number of processors $\rightarrow \infty$
- Performance loss ($S_p < p$) happens because:
  - Programs may not contain enough parallelism, e.g.:
    * pipeline with 4 stages on a 32-CPU machine
    * Some parts of the program might be sequential
  - Overheads introduced by parallelization; typically associated with coordination
  - Architectural limitations, e.g. memory contention
- Amdahl's Law
  - b: sequential part (no speedup)
  - $1 - b$: parallel part (linear speedup)

    $$T_p = T_1 \left(b + \frac{1-b}{p} \right) \qquad S_p = \frac{p}{1 + b\,(p-1)}$$

  - Remarks About Amdahl's Law:
    * It concerns maximum speedup (Optimistic approach). Architectural constrains will make factors worse
    * Takeaway: **All non-parallel parts of a program (no matter how small) can cause problems!**
    * Law of diminishing returns[3]
- Gustafson's Law
  - An Alternative (optimistic) view to Amdahl's Law
  - Observations:

[3]The law of diminishing returns (also law of diminishing marginal returns or law of increasing relative cost) states that in all productive processes, adding more of one factor of production, while holding all others constant, will at some point yield lower per-unit returns [Taken from Wikipedia]

        - Consider problem size
        - Run-time, not problem size, is constant
        - More processors allows to solve larger problems in the same time
        - Parallel part of a program scales with the problem size
    - Formula:
        - b: sequential part (no speedup)

$$T_1 = p(1-b)T_p + bT_p$$
$$S_p = p - b(p-1)$$

- Concurrency vs. Parallelism
    - Concurrency is:
        - A programming model
        - Programming via independently executing tasks
        - About structuring a program
        - A concurrent program does not have to be parallel
    - Parallelism:
        - About execution
        - Concurrent programming is suitable for parallelism

Add views of different architectures

- Concerns in Parallel programming
    - Expressing parallelism
        - Work partitioning (Split up work for a single program into parallel tasks). Can be done:
            - Manually (task parallelism): User explicitly expresses tasks
            - Automatically by the system (e.g. in data parallelism): User expresses an operation and the system takes care of how to split it up
        - Scheduling
            - Assign task to processors (usually done by the system)
            - goal: full utilization (no processor is ever idle)
    - Managing state (data)
        - Shared vs. distributed memory architectures (in the latter, data needs to be distributed)
        - Which parallel tasks access which data, and how (e.g. READ or WRITE access)
        - (Potentially) split up data. Ideal: each task exclusively accesses its own data
        - Depending on the application:
            - Tasks, then data
            - Data, then tasks

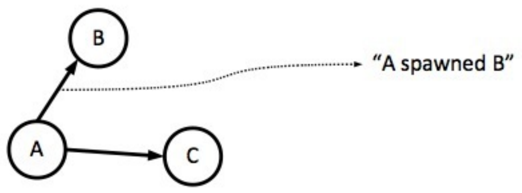
    - Controlling/Coordinating parallel tasks and data
        - Distributed data
            - No coordination (e.g. embarrassingly parallel)
            - Messages
        - Shared data: controlling concurrent access
            - Concurrent access may cause inconsistencies
            - Mutual exclusion to ensure data consistency

- Coarse vs. Fine Granularity
    - Fine granularity
        - More portable (can be executed in machines with more processors)
        - Better for scheduling
        - Parallel slackness (Expressed parallelism ¿¿ machine parallelism)
        - BUT: if scheduling overhead is comparable to a single task $\rightarrow$ overhead dominates
    - Guidelines:
        - As small as possible
        - but, significantly bigger than scheduling overhead; system designers strive to make overheads small
    - Coordinating tasks:
        - Enforcing ordering between tasks, e.g.:
            - Task X uses result of task A
            - Task X needs to wait for task A to finish
        - Example primitives:
            - *barrier*
            - *send()/receive()*
        - All tasks need to reach the barrier before they can proceed

4 Parallel programming models

- Parallel Programming is not uniform
    - Similar to sequential programming
    - Many different approaches to solve problems
    - Many are equivalent under certain conditions, it depends on the application
    - More of an art than a science
- Task Parallel: Programmer explicitly defines parallel tasks (generic, not always productive)
    - Tasks:
        - Execute code

    - Spawn other tasks
    - wait for results from other tasks
  - A graph is formed based on spawning tasks



  - Example: Fibonacci function

```
public class Fibonacci {
   public static long fib(int n) {
      if (n < 2)
         return n;
      spawn a task to execute fib(n-1);
      spawn a task to execute fib(n-2);
      wait for the tasks to complete
      return the addition of the task results
      }
}
```

  - Tasks can execute in parallel
    - But they don't have to
    - Assignment of tasks to CPU is up to the scheduler
  - Task graph is dynamic: unfolds as execution proceeds
  - Intuition: wide task graph → more parallelism
  - Time:

    Check for a better explanation somewhere else

  - Scheduling
    - algorithm for assigning tasks to processors
    - There exists a scheduling with

$$T_p \leq \frac{T_1}{p} + T_\infty$$

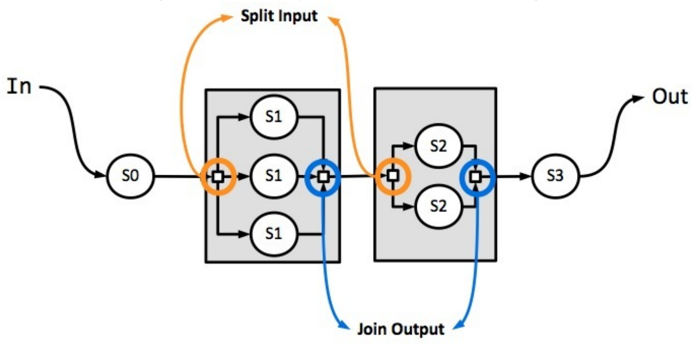
    - This upper bound can be achieved with a greedy scheduler
      1. if $\geq p$ tasks exist, p tasks execute
      2. if $< p$ tasks exist, all execute
    - optimal with a factor of 2
    - linear speedup for $\frac{T_1}{T_\infty} \geq P$
  - Work stealing scheduler
    - First used in CIlk
    - provably: $T_p = \frac{T_1}{P} + O(T_\infty)$
    - empirically: $T_p \approx \frac{T_1}{P} + T_\infty$

    * $\frac{T_1}{T_\infty} >> P \rightarrow$ linear speedup
    * Parallel slackness: granularity
    * Why should the programmer care? A: guideline for parallel programs
- Example: Sum the elements of a list, using D&C[4]

```
public static long sum_rec(List<Long> Xs){
    int size = Xs.size();
    if (size == 1)
        return Xs.get(0);
    int mid = size / 2;
    long sum1 = sum_rec(Xs.subList(0, mid));
    long sum2 = sum_rec(Xs.subList(mid, size));
    return sum1 + sum2;
}
```

```
Divide and Conquer:
    if cannot divide:
        return unitary solution (stop recursion)
    divide problem in two
    solve first (recursively)
    solve second (recursively)
    combine solutions
    return result
```

- So far: Dynamic task graph, but the graph can also be static, i.e. does not change with time
    * Pipeline
        · Think Laundry as an example
        · In full utilization, output rate is 1 item per time unit
        · Time unit is determined by the slower stage: a slower stage stalls the pipeline
        · Hence, we try to create pipelines where each stage takes (roughly) the same amount of time
        · Achieved using splits and joins for parallel stages



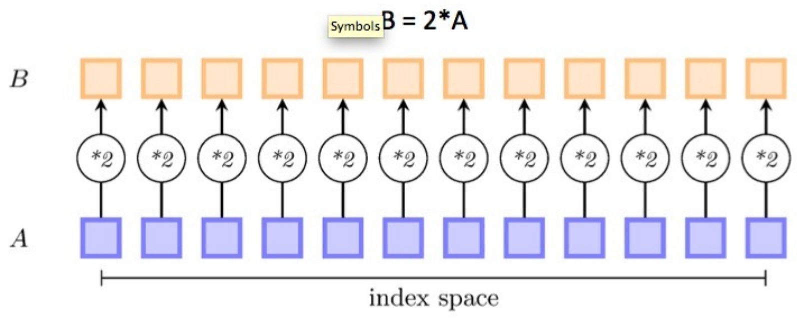
[4]D&C: Divide and Conquer

        * Streaming

          There isn't a single thing in the slides about streaming

        * Dataflows
            - Programmer defines: what each task does, and how the tasks are connected
            - Scheduling: Assigning nodes (tasks) into processors
            - $n < p$: cannot utilize all processors
            - $n == p$: one node per processor
            - $n > p$: need to combine tasks; portability, flexibility (parallel slackness); balancing, minimize communication (graph partitioning)
            - Dataflow programming is a good match for parallel programming since the programmer is not concerned with low-level details; and the same program is used for different platforms (e.g. shared/distributed memory $\rightarrow$ different edge impl.)

- Data parallel: An operation is applied simultaneously to an aggregate of individual items (e.g. arrays) (productive, not general)
    - In task parallelism, programmer describes what each task does and the task graph (dynamic or static)
    - In data parallelism, the programmer describes an operation on an aggregate of data items.
        * Data partitioning is done by the system
        * D.P. is declarative: programmer describes what, not how
    - Example: Map



        * Each operation can be performed in parallel
        * Work partitioning $\rightarrow$ partition the index space
    - Reductions
        * Simple examples: sum, max
        * Reductions over programmer-defined operations
            - Operation properties (associativity/commutativity) define the correct executions
            - Supported in most parallel languages/frameworks
            - powerful constru

Word is chopped

        - Other data types than arrays
        - similar operation: prefix scan
    - Parallel Loops
        - So far: work partition → partition object (e.g. array) index space
        - Iterations can (but do not have to) perform in parallel: work partitioning → partition iteration space
        - Add generality
        - Potential source of bugs if thought of as a sequential loop due to data races

- Managing State: Main challenge for parallel programs. There are different approaches:
    - Immutability
        - Data does not change
        - best option, should be used when possible
    - Isolated mutability
        - Data can change, but only one execution context can access them
        - message passing for coordination
        - State is not shared
        - Each task (actor) holds its own state
        - (Asynchronous) messages
        - Models:
            - Actors
            - Communicating Sequential processes (CSP)
    - Mutable/shared data
        - Data can change/all execution contexts can potentially access them
        - Enabled in shared memory architectures
            - **However**: concurrent accesses may lead to inconsistencies
            - **Solution**: protect state by allowing only one execution context to access it at a time
        - State needs to be protected
            - Exclusive access
            - intermediate inconsistent states should not be observed[5]
        - Methods
            - **Locks**: Mechanisms to ensure exclusive access/atomicity; ensuring good performance/correctness with locks can be hard
            - **Transactional Memory**: Programmer describes a set of actions that need to be atomic; easier for the programmer, but getting good performance might be challenging

[5]Think two people at the blackboard example

5 Introduction to (Parallel) Programming

Not really needed??

6 Introduction to Java

Only the technical things are discussed, no basic java syntax!

- Java is an interpreted language
  - Platform independence via bytecode interpretation
  - Java programs run (in theory) on any computing device (PC, mobile, linux, etc.)
  - Java compiler translates source to byte code
  - Java Virtual Machine (JVM) interprets compiled program
- Compiling/Running
  1. Write it
     - Code or source code: The set of instructions in a program
  2. Compile it
     - Compile: Translate a program from one language to another
     - Byte code: The java compiler converts your code into a format named *byte code* that runs on many computer types
  3. Run (execute) it
     - Output: The messages printed to the user by a program
- JVM Bytecode interpretation
  - JVM interprets compiled Bytecode
  - Simulates a virtual CPU (or rather an entire machine)
  - Translates bytecode into architecture dependent machine code at runtime
  - Loss in performance due to interpretation?
    * Yes and no
    * Some overhead due to interpretation step but JVM is highly optimized
    * Other language constructs impact performance more
- Structure of a Java program

```java
public class name {
        public static void main(String[] args){
                statement;
                statement;
                ...
                statement;
        }
}
```

Where:

- **class**: a program
- **method (main)**: a named group of statements
- **statement**: a command to be executed

Also:

- **import** statement makes classes and methods from other packages (API) available
- The **class body** contains:
  * Instance and class variables (attributes)
  * Names constants
  * Class specific methods (**static** methods)
- **Methods** are what we called functions or procedures so far
  * **Constructor** is a special method that gets called automatically on object creation
- **Methods** must have a name and optionally have:
  * List of parameters
  * Local variables
  * Instructions (statements)
- Each **class** can be compiled independently

```
import ...

class A {
       class body

       constructor {
           ...
       }
        method_m1 {
           ...
       }
       method_m2 {
           ...
       }
}

class B {
     ...
}
```

- In Java, the main method is called at runtime automatically, it serves as the entry point. Methods are then called from here.

7 Loops/Objects/Classes

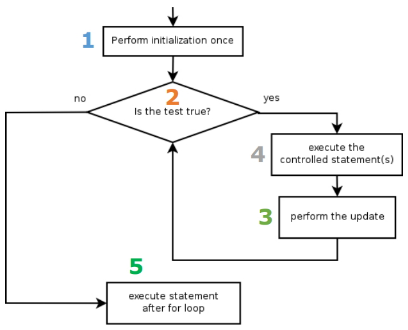
- The ***for*** loop
  - The for loop statement performs a task many times
  - syntax:

```
for (initialization;test;update){
  statement;
  statement;
    ...
  statement;
}
```

  - It performs initialization once
  - Repeat the following:
    - ∗ Check if the test is true. If not, stop.
    - ∗ Execute the statements.
    - ∗ Perform the update
- Loop walkthrough

```
for (int i=1; i<=4; i++){
    System.out.println(i + ‘‘ squared = ’’ + (i*i));
  }
  System.out.println(‘‘Whoo!’’);
```

Output:
1 squared = 1
2 squared = 4
3 squared = 9
4 squared = 16


1
Perform initialization once
no
2
Is the test true?
yes
4
execute the controlled statement(s)
3
perform the update
5
execute statement after for loop

Whoo!

- Categories of loops
  - **Definite loop**: Executes a known number of times.
    - ∗ The **for** loops we have seen are definite loops
      - · Print “Hello” 10 times.
      - · FInd all the prime numbers up to an integer n.
      - · Print each odd number between 5 and 127
  - **Indefinite loop**: One where the number of times its body repeats is not known in advance.
    - ∗ Prompt the user until they type a non-negative number.

        * Print random numbers until a prime number is printed.
        * Repeat until the user has types “q” to quit.

- The ***while*** loop:
    - It repeatedly executes its body as long as a logical test is true.
    - Syntax:

      ```
      while (test){
          statement;
      }
      ```

    - **While** is better than **for** because we don't know how many times we will need to increment to find the factor.

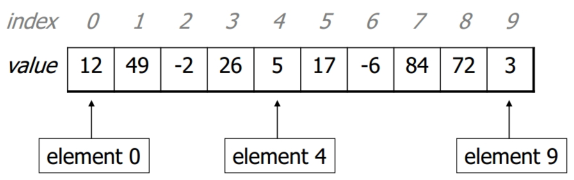
- Sentinel values:
    - **Sentinel loop**: it repeats until a sentinel value is seen
    - Sentinel code example:

      ```
      Scanned console = new Scanner(System.in);
      int sum = 0
      //pull one prompt/read out of the loop
      System.out.print(``Enter a number (-1 to quit): '');
      int number = console.nextInt();

      while (number != -1){
        sum=sum+number;  //moved to top of loop in order to avoid
                         //fencepost problems
        System.out.print(``Enter a number (-1 to quit): '');
        number = console.nextInt();
      }

      System.out.println(``The total is '' + sum);
      ```

- The **do/while** loop
    - It performs its test at the end of each repetition.
        * It guarantees that the loop's body will run at least once.
        * Syntax:

          ```
          do{
            statement;
          } while (test);
          ```

- Arrays
    - **Array**: object that stores many values of the same type.
        * **element**: One value in an array.
        * **index**: A 0-based integer to access an element from an array



  - Accessing elements

    ```
    name[index]          //access
    name[index] = value; //modify
    ```

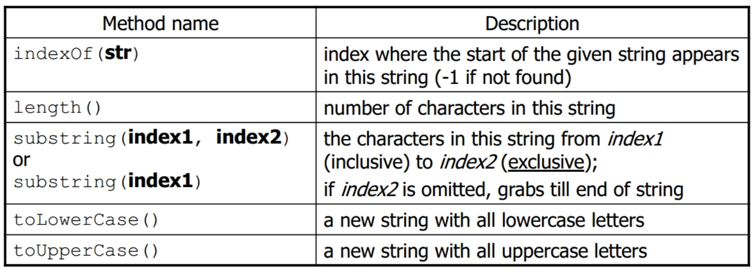
  - One can use **for loops** to insert elements in the array
  - Arrays are reference types
  - Out-of-bounds exception
    - Legal indexes: between **0** and the **array's length -1**.
      - Reading or writing any index outisde this range will throw an *ArrayIndexOutOfBoundsException*

- Strings
  - **String**: An object storing a sequence of text characters
    - Unlike most other objects, a *String* is not create with *new.*
    - Syntax:

      ```
      String name = ‘‘text’’;
      String name = expression;
      ```

- Indexes
  - Characters of a string are numbered with 0-based *indexes*:
    - First character's index: 0
    - Last character's index: 1 less than th string's length
    - The individual characters are values of type chars

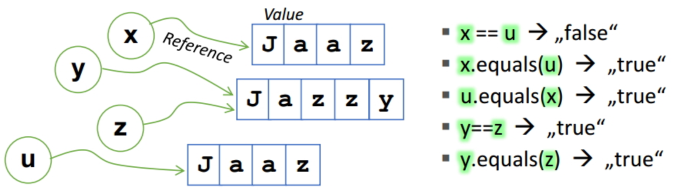
- String methods



Method name	Description
`indexOf(`**str**`)`	index where the start of the given string appears in this string (-1 if not found)
`length()`	number of characters in this string
`substring(`**index1**`, `**index2**`)` or `substring(`**index1**`)`	the characters in this string from *index1* (inclusive) to *index2* (exclusive); if *index2* is omitted, grabs till end of string
`toLowerCase()`	a new string with all lowercase letters
`toUpperCase()`	a new string with all uppercase letters

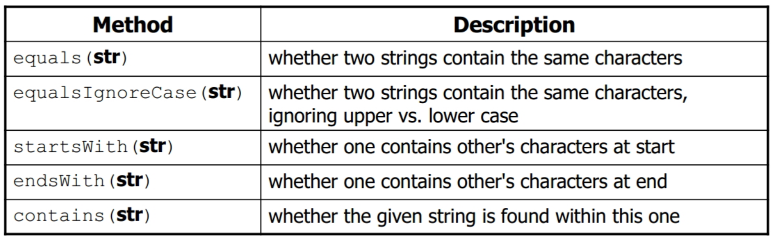
This methods are called using the dot notation

- Comparing strings
  - "==" compares objects by *references*, so it often gives *false* even when two *Strings* have the same letters
  - Example:



- x == u → „false"
- x.equals(u) → „true"
- u.equals(x) → „true"
- y==z → „true"
- y.equals(z) → „true"

- String test methods



Method	**Description**
equals(**str**)	whether two strings contain the same characters
equalsIgnoreCase(**str**)	whether two strings contain the same characters, ignoring upper vs. lower case
startsWith(**str**)	whether one contains other's characters at start
endsWith(**str**)	whether one contains other's characters at end
contains(**str**)	whether the given string is found within this one

```
String name = console.next ();
  if (name.startsWith(‘‘Prof’’)){
      System.out.println(‘‘When are your office hours?’’);
  } else if (name.equalsIgnoreCase(‘‘STUART’’)){
      System.out.println(‘‘Lets talk about meta!‘‘);
  }
```

- Extract from Java API
  - **compareTo**
    - **public int** compareTo(String anotherString)
      Compares two strings lexicographically
    - **Parameters**:
      anotherString - the String to be compared
    - **Returns**:
      The value 0 if the argument strins is equal to this string, less tha 0 if it is lexicographically less than the string argument, value greater than 0 otherwise
  - **concat**
    - **public** String concat(String str)
      Concatenates the string argument to the end of this string.
    - **Paramters**;
      str - the String which is concatenated to the end of this String
    - **Returns**: A string that represents th concatenation of this object's characters followed by the string argument's characters

    - **copyValueOf**
        * **public static** String copyValueOf(**char** data[])
        * **Parameters**:
          data - the character array
        * **Returns**:
          A string that contains the characters of the array
- Classes and objects
    - **Class**:
        * A program entity that represents either:
            1. A program/module, or
            2. A type of objects
        * A class is a blueprint or template for constructing objects