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 → operation(s) → 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 → 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 → 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 ↔ 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

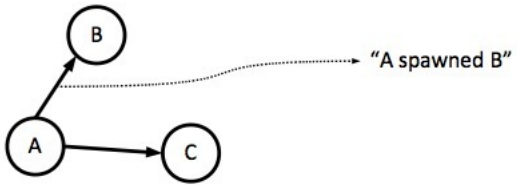
- Coarse vs. Fine Granularity
  - Fine granularity
    - More portable (can be executed in machines with more processors)
    - Better for scheduling
    - Parallel slackness (Expressed parallelism ¿¿ machine parallelism)
    - BUT: if scheduling overhead is comparable to a single task → overhead dominates
  - 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: $T_p = \frac{T_1}{P} + O\left(T_\infty\right)$
    * 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

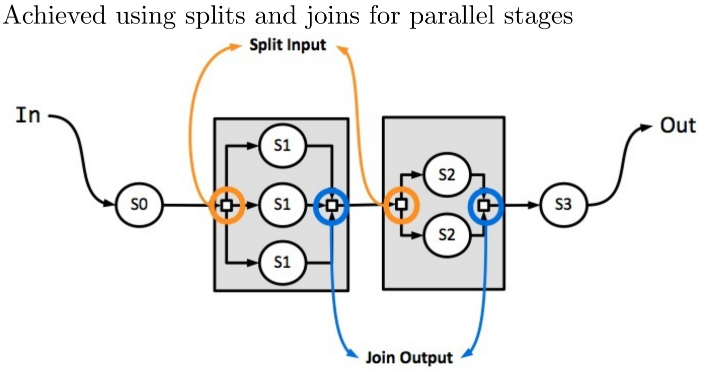
– Example: Sum the elements of a list, using D&C[4]

```
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



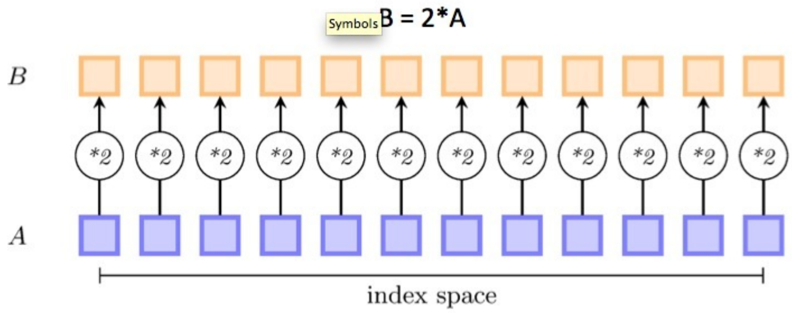
[4]D&C: Divide and Conquer

        * 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 → 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 → 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

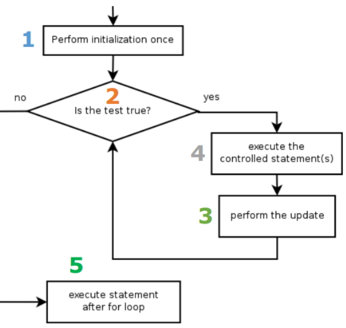
- The *for* loop
  - The for loop statement performs a task many times
  - syntax:

```
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



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.

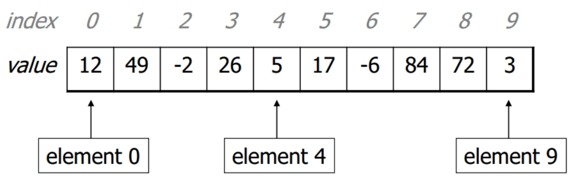
- Sentinel values:
  - **Sentinel loop**: it repeats until a sentinel value is seen
  - Sentinel code example:

```
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;
```

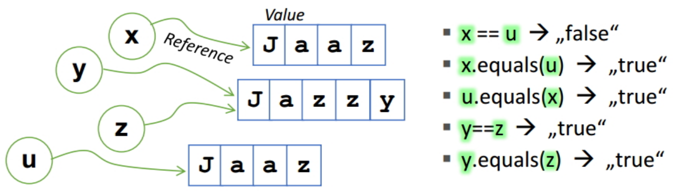
- Indexes
  - Characters of a string are numbered with 0-based *indexes*:
    - First character's index: 0
    - Last character's index: 1 less than th string's length
    - The individual characters are values of type chars

- 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:



- 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 strings 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.
    * **Parameters**;
      str - the String which is concatenated to the end of this String
    * **Returns**: A string that represents the 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

- **Object**: An entity that combines data and behavior.
  - **object-oriented programming (OOP)**:
    Programs that perform their behavior as interactions between objects.
    * data: variables inside the object
    * behavior: methods inside the object
      · You interact with the methods; the data is hidden in the object
    * Constructing an object:
      `Type objectName = new Type (parameters);`
    * Calling an object's method:
      `objectName.methodName (parameters);`

- Inheritance
  Eventhought there are systematic diferences, different kinds of objects often have a certain amount in common with each other.
  Classes *inherit* commonly used state and behavior from other classes, but specialized behavior and states further WTF??

- References and objects
  - Arrays and objects use reference semantics, because of:
    * *efficiency*: Copying large objects slows down a program
    * *sharing*: It's useful to share an object's data among methods
    * Example:

```
DrawingPanel1 panel1 = new DrawingPanel (80, 50);
DrawingPanel panel2 = panel1; //same window
panel2.setBackground(Color.CYAN);
```

- Objects as parameters
  - When an object is passed as a parameter, th object is *not* copied. The parameter refers to the same object.

        * If the parameter is modified, it *will* affect the original object.

```
public static void main(String[] args) {
        DrawingPanel window = new DrawingPanel(80, 50);
        window.setBackground(Color.YELLOW);
        example(window);
}

public static void example(DrawingPanel panel) {
        panel.setBackground(Color.CYAN);
        ..
}
```

- Arrays pass by reference
    - Arrays are passed as parameters by *reference*.
        * Changes made in the method are also seen by the caller

```
public static void main(String[] args) {
        int[] iq = {126, 167, 95};
        increase(iq);
        System.out.println(Arrays.toString(iq));
}
public static void increase(int[] a) {
        for (int i = 0; i < a.length; i++) {
                a[i] = a[i] * 2;
        }
}

Output:
[252, 334, 190]
```

- Arrays of objects
    - `null`: A value that does not refer to any object.
        * The elements of an arays of objects are initialized to `null`.

- Things you can do with null:
    - store `null` in a variable or an array element
    - print a `null` reference
    - ask whether a variable or array element is `null`
    - pass `null` as a parameter to a method
    - return `null` from a method (often to indicate failure)
- Null pointer exception
    - *dereference*: To access data or methods of an object with the dot notation, such as `s.length()`.
        * It is illegal to dereference `null` (causes an exception)

    - `null` is not any object, so it has no methods or data.
  - One can avoid raising an exception by simply using a if case statement before calling an object's method:

```
if (words[i] != null){
        ...
}
```

- Encapsulation
  - Encapsulation is a very important O-O concept
    - Each object has 2 views. An internal view and an external view
  - Encapsulation is a form of protection
    - Also called *Information Hiding*
    - The outside world does not have direct access to the internal implementation or representation of an object
    - As long as the external view does not change, the internal view can take on any form without affecting the outside world
    - Methods have the responsibility of maintaining data integritiy
  - Private visibility offer full encapsulation
    - protected and default offer limited encapsulation
    - public offers no encapsulation
  - Benefits
    - Abstraction between object and clients
    - Protects object from unwanted access
      - Example: One can't fraudulently increase an `Account`'s balance
    - Can change the class implementation later
      - Example: `Point` could be rewritten in polar coordinates with the same methods,
    - Can constrain object's state (**invariants**)
      - Example: Only allow `Accounts` with non-negative balance.
      - Example: Only allow `Dates` with a month 1-12-

  Un po' ripetitivo...???

- Java access restrictions
  - **Packages**: defin a name space
  - Default package used in the absence of a package declaration
  - For class members you can specify which other objects can access (read/write or invoke) them
    - (default)[nothing]: accessible in current package
    - `public`: everywhere
    - `private`: only from this class

        * `protected`: accessible in current package and all subclasses, regardless of their package

- Private fields
    - A field that cannot be accessed from outside the classe
      `private type name;`
- The `this` keyword
    - `this`: Refers to the implicit parameter inside your class.
      (a variable that stores the object a method is called)
        * Refer to a field `this.field`
        * Call a method: `this.method(parameters);`
        * One constructor can call another: `this(parameters);`
- Class versus Instance Methods
    - **Class methods** are marked by the `static` keyword.
        * They are not called via an object reference but directly via the class
        * Can be called from a static context.
        * often serve as utility function.
        * They are generic and need no access to object variables and methods

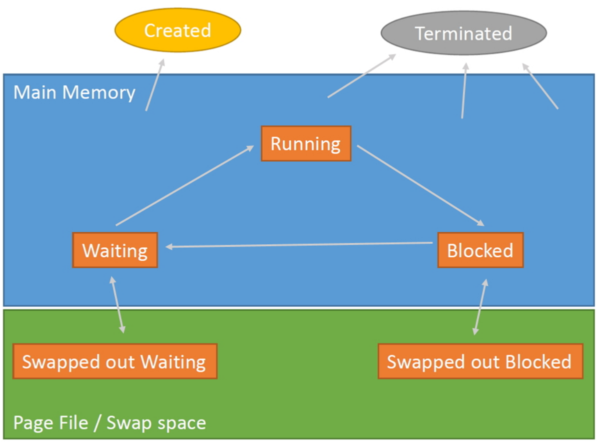
```
classDatum {
        privateintday, month, year;
        staticString monthName(Datum d)
                {
                        if(d.month== 1) return "January";
                        if...                return
                             "February";
                        ...
                }
        ...
}
```

8 Threads

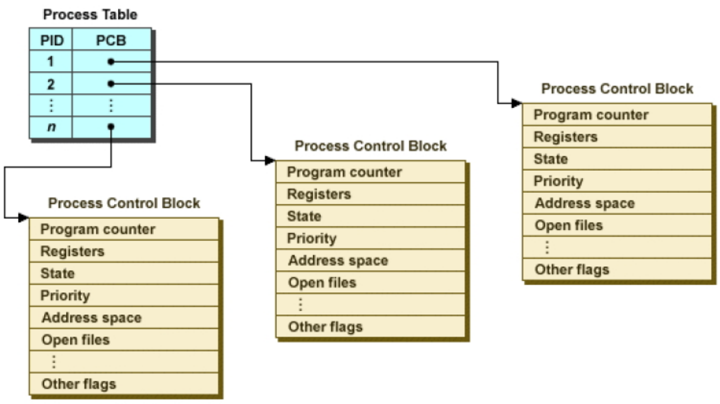
- Multitasking
    - Concurrent execution of multiple tasks
    - Time multiplexing of CPU
        * Creates impression of parallelism, even on single core/CPU system
    - Allows for asynchronous I/O
        * I/O devices and CPU are truly parallel

    * 10ms waitinf for HDD allows other proceses to execute $> 10^{10}$ instructions

- Process ontext
  - Multiple concurrent instances of **same program** (e.g., mutiple browser windows)
  - Multiple applications (=processes) in parallel
  - Each process has a **context**
    * Instruction counter
    * Register content
    * Variable vaues
    * Stack content
    * Resourcing (device access, open files)
- Process States

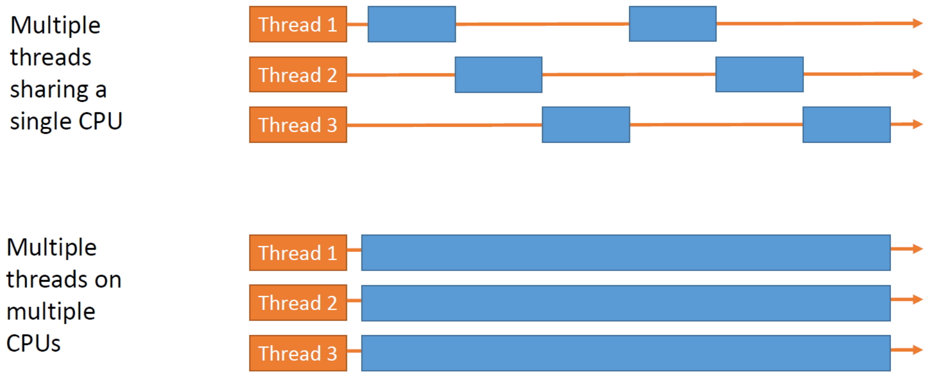


- Process management
  - Proceses need resources
    * CPU time, Memory, etc.
  - OS manages processes:
    * Starts processes
    * Terminates processes (frees resources)
    * Controls resource usage (prevents monopolizing CPU tim)
    * Schedules CPU time
    * Synchronizes processes if necessary
    * Allows for inter process communication
- Process control blocks (PCB)



Java Threads

- Threads (light weight processes)
  - **Threads** (of control) are independent sequences of execution, but multiple threads **share the same address space**
    - Threads are not shielded from eachother
    - Threads share resources and can communicate moer easily
  - Context switching between threads is much more efficient for threads
    - No change of address space
    - No autmoatic scheduling
    - No saving / (re-)loading of PCB state
  - Many more thread changes possible than process switches per unit of time
- Advantages of Multithreading
  - Reactive systems-constantly monitoring
  - More responsive to user input-GUI application can interrupt a time-consuming task
  - Server can handle multiple clients simultaneously
  - Can take advantage of parallel processing
- Multithreading conceptually



- Two options to create Java Threads
  - Extend `java.lang.Thread` class
    * Override run method (must be overridden
    * run() is called when execution of the thread begins
    * A thread terminates when run() returns
    * start() method invokes run()
    * Calling run() does not create a new thread
  - Implement `java.an.Runnable` thread
    * If already inheriting another class (i.e., JApplet)
    * Single method: public void run()
    * Thread classh implements Runnable

11 Synchronization: Beyond Locks

- Locks provide means to enforce atomicity via mutual exclusion
- They lack means for threads to communicate about changes, e.g. changes in state
- Example: producer/consumer (p/c) queues (think: bakery)
  - can be used for data-flow parallel programs, e.g. pipelines where a mean to transfer X from the producer to the consumer is needed
  - There might be multiple producers or (not xor) multiple consumers
  - For an implementation a circular buffer (with a fixed size) can be used with simple **dequeue()/enqueue()** and an in and out counter. Both functions use a shared (reentrant) lock and rely on helper functions to check for full/empty queue[6]
  - If you use a busy wait (while loop) there is a chance of a deadlock (and CPU running high). Using sleep for synchronization as another approach is generally discouraged

[6]Note: If you have a try-catch-finally block and there is a return statement (assume it will be called no matter what) in the "try" part and an unlock() in the "finally" part, the finally part will always be executed (and thus also the lock released!).

- The solution is a condition variable which (ideally) notifies the threads upon change

- A condition interface provides the following methods:
  - `await()`: the current thread waits until it is signaled.
    * Called with the lock held
    * Releases the lock atomically and waits for thread to be signaled
    * When returns, it is guaranteed to hold the lock
    * Thread always needs to check condition
  - `signal()`: wakes up one waiting thread. Called with the lock held
  - `signalAll()`: wakes up all waiting threads. Is called with the lock held

  Conditions are always associated with a lock

- Check then act!
- Conditions can also be used with intrinsic locks where each object can act as a condition, implementing `.notify()`, `.notifyAll()`, `.wait()`.
  - They do not allow for multiple conditions (e.g. different condition for Full/Empty in P/C queues)
- `Object.wait` and `Condition.await`:
  - always have a condition predicate
  - always test the condition predicate: before calling wait and after returning from wait
  - always call wait in a loop
  - Ensure state is protected by lock associated with condition
- Semaphores[7]
  - Invented by Dijkstra
  - Operations:
    * **Initialize** to an integer value
      · After initialization only wait/signal operations are allowed
    * **Acquire**:
      · Integer value is decreased by one
      · If $< 0 \rightarrow$ thread suspends execution
    * **release**
      · Integer value is decreased by one
      · If there is at least a thread waiting, one of the waiting threads resumes execution
  - Notes on Semaphores:

[7]Language background: semaphore is fancy for traffic light in English (see also Spanish).

    * A thread cannot know the value of a semaphore, and thus cannot know whether its execution will be suspended if it calls `acquire()`
    * There is no rule about what thread will continue its operation after `release()`
    * Operations are traditionally also referred to as
      · *P()*: acquire (also wait)
      · *V()*: release (also signal)
  – Building a lock with a semaphore:
    * mutex[8] = Semaphore(1); Initialize mutex using a semaphore, and set it to unlocked (1)
    * lock mutex → mutex.acquire(); only one thread is allowed into the critical section
    * unlock mutex → mutex.release(); One other thread will be woken up
    * Semaphore number:
      · 1 = unlocked
      · 0 = locked
      · $-x$ = locked and x threads are waiting to enter
  – You can (of course) also use semaphore for p/c queues, however you need to use two semaphores to order the operations (and to prevent a deadlock).

- Barriers
  – Rendezvous for arbitrary number of threads i.e. every thread has to wait up for all other threads to arrive at a certain point
  – Can be implemented for n threads with:
    * two semaphores (and one count variable):
      · One as a mutex (used to atomically increment the counter) with default = 1
      · One as a barrier with default = 0 (which is released if count == n and otherwise only acts as acquire-and-release-immediately).

```
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)

mutex.acquire()
   count = count + 1
mutex.release()

if (count == N){
   barrier.release()
}

barrier.acquire()
barrier.release()
```

[8]Mutual Exclusion Locks: make sure that at most one thread owns the lock

- If you want a reusable barrier for n threads (aka 2-phase barrier) with semaphores, you need a count, a mutex and two barriers for it to be thread-safe.

```
count=0; mutex = Semaphore(1); bar0 = Semaphore(0); bar1 =
    Semaphore(1);

mutex.acquire()
count +=1
if (count == N){
  bar1.acquire()
  bar0.release()
}
mutex.release()

bar0.acquire()
bar0.release()

mutex.acquire()
count -= 1
if (count == 0){
  bar0.acquire()
  bar1.release()
}
mutex.release()

bar1.acquire()
bar1.release()
```

12 Advanced (and other) Topics

- Locks can be implemented with low-level atomic operations (basic operations that are guaranteed to be atomic) and busy wait loops (a thread continuously checks a condition)
  - Example: Peterson Lock

```
AtomicBoolean t0 = new AtomicBoolean(false);
AtomicBoolean t1 = new AtomicBoolean(false);
AtomicInteger victim = new AtomicInteger(0);
lock:
    my_t.set(true)
    victim.set(me);
    while (other_t.get() == true &&victim.get() == me)
        ;

        unlock:
        my_t.set(false);
```

Two AtomicBooleans (one per thread) and an AtomicInteger which decides which thread will be selected.

- In order to make our life easier, we need rich(er) atomic operations for AtomicInteger:
  - `getAndSet(val)` (atomically set to val, return old value)
  - `getAndAdd(val)`
  - `getAndIncrement`
  - `getAndDecrement`
  - `CompareAndSet` (CAS for short)
- Lock using `getAndSet`: mutex is an AtomicBoolean which is set to either true or false on lock or unlock (resp.)

```
mutex = new AtomicBoolean(false);
lock:
  while (mutex.getAndSet(true));
unlock:
  mutex.set(false);
```

- CAS; performs atomically the following (optimistically):

```
atomic_int.compareAndSet(int old, int new)

atomically {
  if (current_val == old) {
    current_val=new
    return true
  }
  else {
     return false
  }
}
```

- Lock using `getAndSet`:

```
mutex = new AtomicBoolean(false);

lock:
  while (mutex.getAndSet(true));

unlock:
  mutex.set(false);
```

- Busy-wait
  - Check continuously for a value
  - Wastes CPU-time
  - Alternative: exponential backoff
  - Ideally we would like some sort of notification mechanism
- Mutexes:

  - Locks that suspend the execution of threads while they wait are typically called mutexes (vs spinlocks)
  - Scheduler (typically from the OS) support is required
  - They do not waste CPU time but they have higher wakeup latency
  - Hybrid approach: spin and then sleep

- Locks performance:
  - Uncontended case:
    * When threads do not compete for the lock
    * Lock implementations try to have minimal overhead, typically just the cost of an atomic operation
  - Contended case:
    * When threads do compete for the lock
    * Can lead to significant performance degradation, as well as starvation

- Disadvantages of locking
  - locks are pessimistic by design, they assume the worse/worst and enforce mutual exclusion
  - Performance issues:
    * Overhead for each lock taken even in uncontended case
    * Contended case leads to significant performance degradation
  - blocking semantics (wait until acquire lock)
    * if a thread is delayed for a reason (e.g., scheduler) when in a critical section → all threads suffer
    * Leads to deadlocks (and also livelocks)

- Non-blocking algorithms:
  - Locks: a thread can indefinitely delay another thread
  - Non-blocking: failure or suspension of one thread cannot cause failure or suspension of another thread
  - Lock-free: at each step, some thread can make progress
  - Typically built using CAS operations(more powerful than plain-atomic)
    * see lecture slides for stack example (Page 18)
    * Non-blocking counter example:

```java
public class CasCounter {
  private AtomicInteger value;

  public int getVal() {
    return value.get();
  }

  public int inc() {
    int v;
```

```
    do {
      v = value.get();
    } while (!value.compareAndSet(v, v+1));
    return v+1;
  }
}
```

12.1 An overview of java.util.concurrent

- Lock interface:

```
void lock()
void lockInterruptibly()

boolean tryLock()
boolean tryLock(long time, TimeUnit unit)

void unlock()
Condition newCondition()
```

  Interface implemented by ReentrantLock

- Readers-Writer Lock
  - Observation: multiple readers can concurrently access state
  - Different roles:
    * Readers: Allow to co-exist with other readers
    * Writers: Exclusive access
  - Beneficial for “read-mostly” workloads
  - Can be implemented with semaphores but fairness might be an issue leading to starvation[9] unless prevented by means to notify the read lock about waiting writers

- Java collections
  - Aggregate Objects: Objects that group multiple elements into a single unit
  - Interfaces (e.g. Collection, List, Set, SortedSet,...)
  - Implementations (e.g. ArrayList and LinkedList for List)
  - Algorithms (e.g. Sort)

  → Based on Java Generics

- Synchronized Collections:
  - Vector, HashTable (JDK 1.0)
  - java.util.collections (JDK 1.2):

[9]Starvation: when a particular thread cannot resume execution; different from deadlock, where all the threads are unable to proceed

        - synchronizedList(List<T> list)
        - synchronizedMap(Map<K,V> m)
        - synchronizedSet(Set<T> s)
        - synchronizedSortedMap(SortedMap<K,V> m)
        - synchronizedCollection(Collection<T> c)
    - They are synchronized wrapper classes
    - Wrap every public method in a synchronized block
    - Thread-safe
    - Client-side locking for compound actions (e.g. iteration, put-if-absent)
    - Poor concurrency: single, collection-wide lock

- Concurrent Collections:
    - Thread-safe
    - Not governed by a single lock
        - Cannot be locked for exclusive access
        - No client-side locking
        - Common compound operations added (e.g. put-if-absent, replace-if-equal,...)
    - Examples:
        - ConcurrentHashMap
            - As a replacement for synchronized hash-based map
        - ConcurrentSkipListMap, ConcurrentSkipListSet
            - As a replacement for synchronized tree-based impls.
        - CopyOnWriteArrayList, CopyOnWriteArrayList
            - Frequent reads, infrequent writes

- Queues:
    - BlockingQueue
        - ArrayBlockingQueue, LinkedBlockingQueue (FIFO)
        - PriorityBlockingQueue (ordered)
    - TransferQueue: allows to wait until a consumer receives item;
    - SynchronousQueue: hand-of, no internal capacity
    - Dequeue/BlockingDeque:
        - Allows efficient removal/insertion at both ends (head/tail)
        - Work stealing pattern

- Synchronizers:
    - Semaphores
    - CyclicBarrier (Java's barrier)
    - CountDownLatch (thread wait until countdown reaches zero)
        - `CountDownLatch(int count)` - initialize latch

        * `.await()` - wait for event
        * `.countDown()` - decrement count

- Future:
  interface Future<V>
    - Represents a result (type T) for an asynchronous computation
    - `.get()` - wait for result
    - `.isDone()` - check if task is completed
    - `.cancel()` - attempt to cancel computation

13 Parallel Tasks

Example for most of this section: $\sum x_i$

```
public static int sum(int[] xs) {
        int sum = 0;
        for (int x: xs)
                sum += x;
        return sum;
}
```

- When writing a parallel program, write a sequential version first! This is useful for knowing the results are correct and evaluate the performance of the parallel program
- Divide-and-conquer approach: recursive sum with the lower and upper half of the remaining part which cuts off at size $= 1$ is a lot slower ($\times 10$).
- Task parallel model, basic operations:
    - create a parallel task and wait for the parallel tasks to complete
    - when using D&C a task for the first and second part (one each) are created, and upon finishing the operations, their results are combined
- One thread per task model:
    - expensive to create
    - consumes many resources
    - Scheduled by the OS
    - Generally Inefficient!
- **ExecutorService**: A (huge) amount of tasks is handled by an interface which assigns a thread from a thread pool to each task and returns a Future
- Future (`interface Future<V>`):
    - represents a result (Type `T`) for an asynchronous computation
    - `.get()` - wait for result

  - `.isDone()` - check if task is completed
  - `.cancel()` - attempt to cancel computation

- Note: Callable vs Runnable:
  - Interface Runnable: `void run()`
  - Interface Callable<V>: `V cal()`
  - Runnable doesnt return a result, Callable does

- Executor service for recursive sum[10]:
  - Task is described as
    * The array to be summed
    * The region for which the task is responsible for
  - Additionally, an instance to the ExecutorService is passed so that the task can spawn other tasks
  - Problems (observation: no result returned):
    * Tasks create other tasks and then wait for results
    * When they are waiting they are keeping threads busy
    * Other tasks need to run so that the recursion reaches its bottom
    * System does not know that tasks waiting need to be removed so that other tasks can run.
  - Problems with this approach:
    * Tasks create other tasks (which is not supported)
    * Work partitioning (splitting up work) is part of the task
    * We can decouple work partitioning from solving the problem

- Fork/Join framework:
  - `ForkJoinTask<V>`: implements `Future<V>`
  - `ForkJoinPool`: implements `ExecutorService`
  - `.fork()`: creates a new task
  - `.join()`: returns the result when task is done
  - `.invoke()`: executes task without spawning a new task (in-place)
  - subclasses need to define `compute()`

  Note `fork()`, , `join()`, `join()` don't work (well) in Java, solved by using[11]

  ```
  t1.fork(), r2 = t2.compute()
  return r2 + t1.join()
  ```

- Problems of overhead:
  - Bad speedup due to too much overhead[12] (scheduling etc.)
  - If the work of each task is small, overheads dominates
    * can be solved by making each task work more, here: increase cutoff

[10] Have a look at the code in the slides, pp36-38

[11] "+" is in this case the arithmetic addition but can also be something else of a combining nature

[12] Overheads are time costs that have nothing to do with the computation

14 Transactional Memory (TM)

REM1: If I remember correctly, this was already discussed, maybe remove?

- Problems using locks:
  - Ensuring ordering (and correctness) is **really hard**
  - Locks are not composable
  - Locks are pessimistic
  - Locking mechanism is hard-wired to the program

  END REM1

- Aims at removing the burden of having to deal with locks from the programmer and place it on the system instead
- With TM, the programmer explicitly defines atomic code sections and is only concerned with the *what* and not the *how* (declarative approach)
- TM benefits:
  - Easier and less error-prone
  - Higher semantics
  - Composable
  - Optimistic by design
- changes made by a transaction are made visible atomically
- transactions run in isolation - while a transaction is running, effects from other transactions are not observed (as if transaction takes a snapshot of the global state when it begins and operates on that snapshot)
- Note: while locks enforce atomicity via mutual exclusion, transaction does not require that

TM is inspired by transactions in databases where transactions are vital

- ACID: **A**tomicity, **C**onsistency, **I**solation, **D**urability

Implementation of TM:

- Keep track of operations performed by each transaction
- Concurrency control, system ensures atomicity and isolation properties

Transactions can be aborted if a conflict has been detected by the concurrency control (CC) mechanism

- aborts are possible e.g. if theres a deadlock
- on abort, a transaction can be retried automatically or the user is notified

Where TM is/can be implemented:

- Hardware TM : can be fast but cannot handle big transactions
- Software TM (STM) : in the language, greater flexibility, performance might be challenging
- Hybrid TM; TM is still work in progress with many different approaches and is still under active development

Design choice:

- strong vs weak isolation:
  - Q. What happens when shared state accessed by a transaction, is also accessed outside of a transaction? Are the transactional guarantees still maintained?
  - A.
    - Strong isolation: Yes,
      * easier for porting existing code
      * difficult to implement, overhea
    - Weak isolation: No
- Nesting:
  - Q. What are the semantics of nested transactions? (Note: nested transactions are important for composability)
  - A.
    - flat nesting (inner aborts → outer aborts)
    - inner commits → changes visibly only if outer commits, closed nesting (inner abort does not result in an abort for the outer transaction
    - inner transaction commits → changes visible to outer transaction but not to other transaction
    - only when outer transaction commits, changes of inner transactions become visible), other approaches (e.g. open nesting)

The more variables are part of a transaction (and thus protected) the easier it gets to port existing code but the more difficult to implement ,too (need to check every memory operation)

Reference-based STMs: mutable state is put into special variables

- these variables can only be modified inside a transaction, everything else is immutable (or not shared; see functional programming)

Mechanism of retry:

- implementations need to track what reads/writes a transaction performed to detect conflicts, typically called read-/write-set of a transaction
- when retry is called, transaction aborts and will be retried when any of the variables that were read, change

Issues with transactions:

- it is not clear what the best semantics for transactions are

- getting good performance can be challenging
- I/O operations: can we perform I/O operations in a transaction?

I/O in transactions: in general, I/O operations cannot be rolled-back and thus generally cannot be aborted

- that is why I/O operations are not allowed in transactions
- one of the big issues with using TM
- (some) STMs allow registering I/O operations to be performed when the transaction is committed

15 Designing Parallel Algorithms

- There are no rules whatsoever, yet - as (very) often - it is a matter of experience
- The following points can/should be considered:
  - Where do the basic units of computation (tasks) come from?
    - This is sometimes called "partitioning" or "decomposition"
    - Depending on the problem partitioning in terms of input and/or output can make sense or functional decomposition might yield better results
  - How do the tasks interact?
    - We have to consider the dependencies between tasks (dependency, interaction graphs)
    - Dependencies will be expressed in implementations as communication, synchronization and sharing (depending upon the machine model)
  - Are the natural tasks of a suitable granularity?
    - Depending upon the machine, too many small tasks may incur high overheads in their interaction. Should they be collected together into super-tasks?
  - How should we assign tasks to processors?
    - In the presence of more tasks than processors, this is related to scaling down
    - The"owner compute" rule is natural for some algorithms which have been devised with a data-oriented partitioning
    - We need to ensure that tasks which interact can do so as (quickly) as possible.
- D&C is a very important technique and particularly helpful in PP since the recursive step can instead be parallelized

- Number of threads to be used:
  - “Runtime.getRuntime().availableProcessors()” might be the right amount but your program may not get access to all cores
  - too few threads are bad because core(s) is/are idle
  - too many threads can be bad because of the overhead[13]
- Sorting : If the array is sorted the following condition must hold (equal only if $A_i = A_j$):
  - $A_i \leq A_j$ for $i < j$
  - features of a sorting algorithm:
    * stable (duplicate data is allowed and the algorithm does not change duplicate's original ordering relative to each other), in-place ($\mathcal{O}(1)$ auxiliary space), non-comparison
    * horrible $\Omega(n^2)$: bogo, stooge
    * simple $\mathcal{O}(n^2)$: insertion[14] , selection[15] , bubble, shell
    * fancier $\mathcal{O}(n \log n)$: heap, merge, quick sort (on average!)
    * specialized $\mathcal{O}(n)$: bubble, radix
- Linked Lists and Big Data:
  - Mergesort can very nicely work directly on linked lists
  - Heapsort and Quicksort do not;
  - InsertionSort and SelectionSort can too but slower
  - Mergesort also the sort of choice for external sorting
- Differences:
  - Quicksort and Heapsort jump all over the array
  - Mergesort scans linearly through arrays
  - In-memory sorting of blocks can be combined with larger sorts
  - Mergesort can leverage multiple disks
- PRAM model:
  - processors working in parallel, each is trying to access memory values

[13] This depends on the actual overhead the language introduces (in Java rather big)

[14] At step k, put the k^{th} input element in the correct position among the first elements

[15] At step k, find the smallest element among the unsorted elements and put it at position k

- when designing algorithms, the type of memory access required needs to be considered
- scheme for naming different types: [concurrent | exclusive]READ[concurrent | exclusive]WRITE[16]
- typically CR are not a problem since the memory isnt changed whereas EW requires code to ensure writing is exclusive
- PRAM is helpful to envision how it works and the needed data access pattern but isnt necessarily the way processors are arranged in practice

16 Java GUIs - MVC - Parallelism

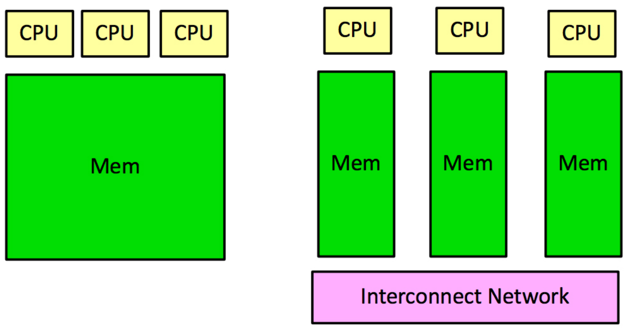
Dont get me wrong, but Im having a hard time writing up this lecture

- (important) concepts:
  - MVC (model (application domain, state and behavior)
  - view (display layout and interaction views)
  - controller (user input, device interaction))
  - layout managers
  - event-driven design (listener, worker[17] , callback, fire/handle)
  - GUI (painting)
- Swing threads:
  - initial[18]
  - event dispatch[19]
  - worker thread[20]
- MVC:
  - Model: complete, self-contained representation of object managed by the application, provides a number of services to manipulate the data, computation and persistence issues
  - View: tracks what is needed for a particular perspective of the data, presentation issues
  - Controller: gets input from the user, and uses appropriate information from the view to modify the model, interaction issues

[16] Abbreviated as E/C and R/W; ERCW is never considered
[17] In Swing, this implements Runnable
[18] Main thread
[19] Drawing/painting the GUI
[20] Background thread, can be used for (heavy) computation to keep GUI responsive

17 Concurrent Message Passing

- Goal: avoid (mutable) data sharing, instead use concurrent message passing (actor programming model) since many of the PP problems (so far) are due to shared state
- isolated mutable state:
  - state is mutable, but not shared
  - each thread/task has its private state
  - tasks cooperate with message passing



- shared memory architecture (left side in image)
  - message passing and sharing state is used
  - message passing can be slower than sharing data yet is easier to implement and to reason about
- Distributed Memory Architecture (right side in image)
  - sharing state is challenging and often inefficient, using almost exclusively message passing
  - additional concerns such as failures
- Message passing works in both shared and distributed memory architectures making it more universal
  - Example: shared state counting (i.e. atomic counter) with increase() and get():
    #1: one counter thread, the other threads ask for its value
    #2: every thread has its own (local) counter (Java: ThreadLocal), when sum is requested all threads return the value of their local counter
  - Example: bank account:

        - sequential programing: single balance
        - PP shared state: single balance & protection
        - PP distributed state: each thread has a local balance (budget), threads share balance coarsely

- distributed bank account (cont.): each task can operate independently, only communicate when needed
- Synchronous vs. Asynchronous messages:
    - sync: send blocks until message is received (Java: SynchronousQueue)
    - async: send does not block ("fire-and-forget"), placed into a buffer for receiver to get (Java: BlockingQueue, async as long as there is enough space (to prevent memory overflow))
- concurrent message passing programming models:
    - actors: state-full tasks communicating via messages (e.g. erlang)
    - channels[21]: can be seen as a level of indirection over actors, Communicating Sequential Process (CSP) (e.g. go)
- go (by Google): language support for: lightweight tasks (aka goroutines), typed channels for task communications which are synchronous (unbuffered) by default
- actor programming model:
    - a program is a set of actors that exchange (async) messages
    - actor embodies:
        - state
        - communication
        - processing
    - An Actor may:
        - process messages
        - send messages
        - change local state
        - create new actors
- event-driven programming model: a program is written as a set of handlers (typical application: GUI)
- Erlang:
    - functional language

[21] not an official term

- developed for fault-tolerant applications, if no state is shared, recovering form errors becomes much easier
- concurrent, following the actor model
- open-source

- Actor examples:
  - Distributor: forward received messages to a set of names in a round-robin fashion
    * State: an array of actors with the array index of the next actor to forward a message
    * Receive: messages $\rightarrow$ forward message and increase index (mod), control commands (e.g. add/remove actors)
  - Serializer: unordered input (e.g. due to different computation speed) $\rightarrow$ ordered output;
    * State: sorted list of received items, last item sent
    * Receive: if we receive an item that is larger than the last item plus one, add it to the sorted list; if we receive an item that is equal to the past item plus one: send the received item plus all consecutive items form the last and reset the last item
- concurrent message passing in Java:
  - For simple applications, queues can be used which might be difficult especially for large tasks
  - Instead use Akka framework (written in Scala, interface for Java): follows the actor model (async messages), rich set of features[22]
- Akka actors example:
  - ping-pong: client sends n PINGs to server which responds with Pong upon receiving back to sender, master stops execution when receiving DONE
  - Version 2 with restart on DONE: add a message type SETUP to the client passing the server actor reference and the count, if the client receives SETUP before DONE it can either wait for DONE and the restart or discard the message
- Collective operations:
  - Broadcast: send a message to all actors (related: multicast, sending a message to some actors), parallel broadcast using a tree where every parent forwards the message to its children until it reaches the leafs (top-down)

[22] important methods to be overridden: preStart(), onReceive()

- reduction: perform a computation from values of multiple nodes (e.g. balance of all bank accounts), using a tree where a parent receives the message from its children, performs operation and sends it to parent (bottom-up)

18 Data Parallel Programming

18.1 Data Parallel Programming

- Task vs Data parallelism:
  - Task: work is split into parts, by parallelizing the algorithm, very generic but cumbersome
  - Data: simultaneously applied operation on an aggregate of individual items (e.g. array), declarative (= what not how), splitting up the data for parallelism, less generic
- Main Operations:
  - map:
    * input: array (x), operation ($f(\cdot)$)
    * output: aggregate with applied operation ($f(x)$)
    * parallel execution: split array into chunks and assign chunks to processors (scheduling)
    * generally more chunks leads to better load balancing (parallel slackness)
    * order of execution must not influence the result (since order depends on scheduling), given by pure functions (no side effects, same result for same argument)
  - reduce (reduction) :
    * input: aggregate (x), binary associative operator ($\bigoplus$) with an identity I, output: $x_1 \bigoplus x_2 \bigoplus \cdots \bigoplus x_n$
    * result stays the same for sequential vs binary tree if operator is associative ($(a+b)+c = a+(b+c)$)
    * f operation is commutative ($a+b = b+a$), different scheduling is possible; e.g. sum, max
  - prefix scan: if it is an addition, it is a prefix sum
    * input: aggregate (x), binary associative operator ($\bigoplus$) with an identity I
    * output: ordered aggregate $(x_1, x_1 \bigoplus x_2, \ldots, x_1 \bigoplus x_2 \bigoplus \cdots \bigoplus x_n)$
  - prefix scan algorithm parallel version:
    * addition example:

- · 1st step is a reduction where two numbers are summed together and then pass their sum up the tree until it reaches the root i.e. bottom-up summing up all the values, two at a time
  - · 2nd step is a down sweep where every node gets the sum of all the preceding leaf values passed whereas preceding is defined as pre-order
  - · Have a look at slides 18 - 21 if in doubt

- application of pre-scan:
  - ∗ line-of-sight, visible points (e.g. mountain tops) from a given observation point: point I is visible if no other point between I and the observer has a greater vertical distance $\theta_i = \arctan \frac{altitude_i - altitude_0}{i}$
  - ∗ compute angle for every point, do a max-pre-scan on angle array (e.g. 0,10,20,10,30,20 → 0,0,10,20,20,30), if $\theta_i > maxprevangle_i$ then $visible_i = true$ else $visible_i = false$
  - ∗ parallelizable parts:
    - · for loop to compute angles
    - · for loop to compute visibility can be written as parfors (parallel for loops)
- parfor:
  - ∗ iterations can be performed in parallel, work partitioning → partition iteration space
  - ∗ potential source of bugs if thought of as a sequential loop (data races; think factorial)

18.2 Data Parallel Programming in Java 8

- Functional programming crash course:
  - functions are first-class values (composition), pure functions (immutability)
  - such function are called lambdas or anonymous functions
- Functions as values: functions can be passed to other functions as arguments (such functions accepting such arguments are called high-order functions), e.g. $map(f, list) : f$, $filter(fn, list) : f$
- Lambdas make programming more convenient
- Data parallel programming in Java 8 is done using streams, providing means to manipulate data in a declarative way, allowing for transparent parallel processing
- Menu example:
  - input: stream

  - output: stream, map/filter/etc. are applied
  - collect in the end, doesn't create a stream
  - overall translates a stream into a collection

- Parallel streams: created by applying .parallel() on a stream, splits it up into chunks for different threads; implemented using ForkJoin