JAVA vs PYTHON

statically typed — dynamically typed

types of vars are known at compile time — Checking is deferred to runtime,

useful for finding bugs
- static checking
- dynamic checking
- no checking

arrays are fixed length sequences

~~[illegible]~~
~~[illegible]~~

Changing values means refer to another memory space

Immutability is a good design because doesn't allow thing to change, avoiding unexpected values

final makes a reference immutable.

The compiler will check statically ~~[illegible]~~ if ~~[illegible]~~ the final var changes

DOCUMENTING

TESTING

Is a part of validation:

formal reasoning
informal "
testing

The goal is to make the program fail
- Be systematic
- DO it early
- Automate it

TEST-FIRST APPROACH

STATE MACHINE

Strings are immutable

StringBuilder object can mutate through operation

The difference between those ~~[illegible]~~ ~~[illegible]~~ is good performance for mutable. However immutable allows to keep a state

Iterator

```
for (String s : l)
  ↓ converts to..
Iterator iter = l.iterator()
while (l.hasNext())
    String s = iter.next()
```

Has methods hasNext() and next()

A state machine is a set of states that the system can be in

transitions between states are called events

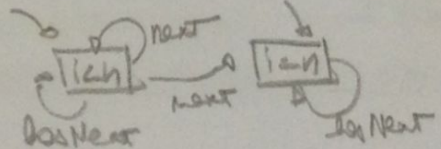
For a mutable object, it's state is representing by the instance variable and events are the operations that can be performed to it

Example of an ~~[illegible]~~ Iterator, hasNext is an observer, it doesn't change the state, Next() is a mutator, what we called an event

"STATE MACHINE DIAGRAMS" — "More abstract, combine more states"

. . .

-Iterator



ALIASING

Multiple references to the same object (aliasing) mean that multiple places are relying on it to remain consistent.

There's a cost in bug safety in doing so, but we have to for performance and convenience

Examp MIDI PIANO

InputStream and OutputStream objects are state machines that read and write streams of characters

A keyboard / synth has two states for each key
- key UP — key DOWN

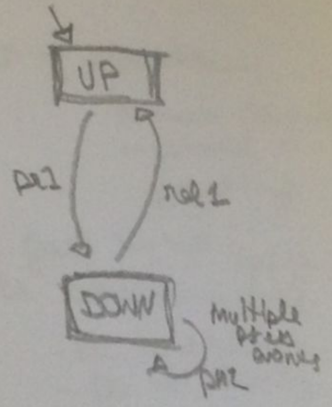
events for each; pr: key pressed — rel = key released

MIDI PIANO Example

Input stream and OutputStream are Java objects, behaving like state machines that read/write streams of characters

A keyboard/switch has two states for each key: key UP key DOWN

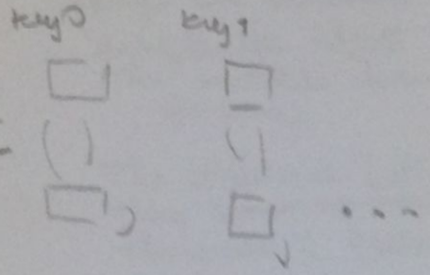
events are: pr, key pressed   rel; key released



the set of traces in a state machine represent the sequence of events

<>
<pr>
<pr, rel>
<pr, pr, rel>
...

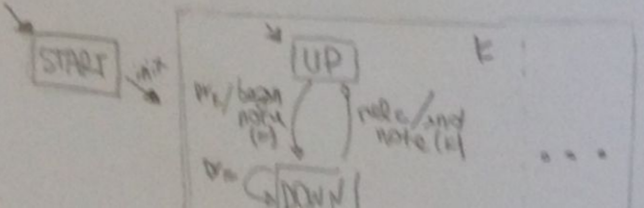
Each key's machine can take steps independently.
Generating shared events (transit. with same label) must be synchronized, this is not the case.
We assign to each key a different label!



<>
<pr1>
<pr1, rel1>
<pr1, rel1, pr1>
<pr2, pr2, pr2>

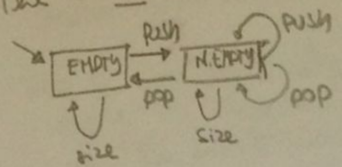
this model handles cases that can never happen But it's fine as long as it doesn't fail

The software needs explicit initialization (init event)
After that we connect the pieces



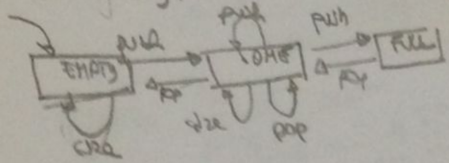
I take an input event and produce an output event

The STACK state machine



As you notice, pop is not legal in empty stack, we're following the spec-first, test-next, implement-last

stack is flexible but for high-performance we will use ArrayList which size can change the whole stack is copied when it gets too big.
If we know there's a maximum size we can still use an array which looks like ↓



REGULAR EXPRESSIONS & GRAMMAR

For today's example we are going to use ~~our~~ markup <i/m> languages

HTML Markdown latex

<i> </i> _italic {\em emphasis}

Let's first draw state machine representing the behavior of a renderer for these markup lang.!

Our want is divided in lexical analysis (lexing) and parsing (processing). We will implement them using separation of concerns, they will interact through an interface

HTML lang
<i>

<i><i>
parsed

- Lexical analysis

  Takes a stream of low level symbols and aggregates them, forming into lexems, outputs are called tokens.
  It can remove whitespaces or combine symbols into classes useful to the parser.
  So tokens can also be objects.
  In Java, an enum class is a useful way to define tokens.

TOKENIZATION

<i>\n\0italic</i>
(<i> italic </i>)

int square(int x)
↓
int Id("square")
(int Id("x"))

For our markup languages we want tokens for ~~the lex~~ _, <i>, {\em} ~~for italics~~ meaning italic

We will consider all other text as a special token called ~~the~~ "text"

- Lexer

A lexer does lexing. Like an iterator, contains events Next() and HasNext()/END token.
It is a state machine that processes character input stream and generates a token sequence

HTML

This is an <i>italic</i> word

text("This is an ")
<i> Text("italic") </i>

GRAMMAR

grammar defines a set of sentences

Ex URL := Protocol :// Address
Address ::= Domain . TLD
Protocol ::= http | ftp
Domain ::= mit | apple | pbs
TLD ::= com | edu | org

terminals are

://, ., http, ftp, mit, apple, pbs..

non-terminals

TLD = {com, edu, org}
Domain {mit, apple, pbs}

REGULAR GRAMMARS

markdown ::= ([^_]* | _[^_]*_)*

regular expression

A grammar that can't be reduced to a single nonterminal production is called context free.
Most prog. lang. are

ADP

After defined types, a major step forward is given by Abstract types. Which are user defined data types within the ~~language~~ code.

This is allowed by the programming language

Notable for the develop are Dahl, Hoare, Parnas and so on.

The key idea in ~~abstraction~~ is that a type is characterized by the operations you can ~~do~~ to it perform

Users can create a type and focus on operations, without worrying about compilers work (address where values are stored for examp).

~~Types~~ types are classified as mutable and immutable

A mutable type contains ~~[illegible]~~ operations which when executed ~~change the result of the same object~~ cause the result of other operations on ~~the~~ the same object to give different results

Date.SetMonth() → GetMonth() values

~~[illegible]~~ But strings are immutable because it's operations creates new string objects rather than changing existing ones

Operations on abstract types are classified as follows → Creators. creates a new object from another object, when needed

Observers
Mutators
Producers

takes objects change ~~of the abstract~~ objects ~~abstract~~, and returns objects of diff type

create new objects from old of the same type

Creator $t^* \to T$
producer $T^+, t^* \to T$
mutator $T^+, t^* \to$ void
observer $T^+, t^* \to t$

- T is abstract type
- t other type
-

int in java is immutable, so has no mutators for ex

List is mutable, and is also an interface (used by ArrayList and LinkedList).

~~Strings are immutable~~

Designing ADP

Is better to have ~~a~~ simple operations that can be combined

Each should have a well defined purpose and should have coherent behaviour among ~~all~~ general cases

A type could be generic, like a graph. Or domain specific, like a street map. But it should not mix domain specific and generic features

The use of an abstract type must be independent ~~from~~ of it's representation, so that changes in representation, does not affect code outside

Specifications helps to know what you can change when modifying an implementation

An ADP should preserve ~~its~~ its invariant (a property of a program that is always true). Immutability for ex

A common threat to immutability is representation exposure

~~PAGE 4-5~~ 34

REPRESENTATION EXPOSURE

The code outside the class can modify the abstract type's representation directly. Invariants and representation independence (prev. page) are threatened.

We can use public and private statements, final guarantees that the field won't be reassigned.

In the case we need to expose a field that we want to be immutable, we can use defensive copying to avoid leaving references to the rep

TIP TIME

To avoid these problems, carefully inspect the argument types and return types of your ADT operations

JAVA

Java collection contains immutable wrappers to spare you all these operations. It takes a mutable obj and wraps with something that looks the same, but whose modifying operations are disabled.

ESTABLISH INVARIANTS

An invariant is a property that is true for the entire program. So we need to:

- establish invariant at init stage
- ensure that transitions preserve it

then the invariant is true for all ADT instances

creators producers INIT

observers mutators PRESERVE

"DEEP ADT THEORY"

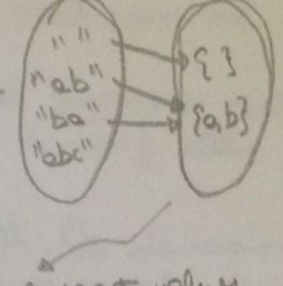
Thinking of Abstracts, helps when considering two spaces of values

rep values consists of values of the actual implementation, commonly implemented as a small network of objects

abstract values, are the values that the type is designed to support, they are the way we want to view the elements of an abstract type

"Any rep value could be an array but the abstract value could be the list instead, we need an init, we don't need to know ~~how to know it~~ how is produced."

Given an abstract type, the implementation objective is to implement rep values. In a way to create abstract values, given by the @return specification



That's fine!

All abstract values must be mapped

If the type of the rep is mantained, it doesn't make sense to map all values

However all abstracts must be mapped.

We can impose properties, of the array has no duplicates, removal is not needed

An abstract function AF: R→A maps reps to the abstract they represent. This is, as we saw, onto, not necessarily one to one and partial.

A rep invariant maps reps to bools RI: R→bool, it tells us whether a rep is well formed (its the subset of rep values on which AF: R→A is defined)

IM. LIST IMPLEMENT

Immutable is not only safe, but sharing produces performance benefits (less memory, no need for copying)

Let's define a data type for immutable list, ImList<E>. It has four operations

empty: void → ImList
returns an empty list — empty() = [ ]

cons: E × ImList → ImList
returns a new list from an older list — cons(0, empty()) = [0]

x ≡ cons(0, cons(1, cons(2, ~~cons~~ empty)))) = [0,1,2]

first: ImList → E
returns the first element — first(x) = 0

rest: ImList → ImList
returns all elements except for the first — rest(x) = [1, 2]

The same operations are used in Scheme and Lisp for list implementation. They are widely used in functional programming.

To implement this we will use an interface, ImList and two classes that implements it, Empty & Cons

empty operation result

cons operation result

The right way to implement Empty is as a static that takes no arguments and produces an empty instance. But Java doesn't allow method in interfaces so we create a class, we'll sacrifice rep independence

RECURSIVE D. TYPE

ImList and Empty with Cons form a recursive data type, that because Cons recursively requires an implementation of ImList (for rest) inside its rep.

DATATYPE DEFINITION

ImList = Empty + Cons(first:E, rest:ImList)

cons(0, cons(1, cons(2, Empty)))

RECURSIVE REALLY VISIBLE

Using tree example

Tree = Empty + Node(e:E, left:Tree, right:tree)

datatype on left

variants each is defined by a constructor

Recursive datatypes are a convenient way to describe operations, using a function for each case.

We can provide the following recursive definition for size()

size(Empty) = 0
size(Cons(0, Cons(1, Empty)))
= 1 + size(Cons(1, Empty))
= 1 + (1 + size(Empty))
= 2!

The method is declared in ImList interface and initialized differently in Empty and Cons

TUNING THE REP

The size method goes through the whole list recursively going to the next element. It takes O(n) everytime.

We can solve it by caching the size and return it everytime.

Size is mutable, that's a ~~[illegible]~~ beneficient mutation, the state change doesn't change the abstract value

EMPTY vs NULL

Using an object instead of a null reference to signal base case and end case is an example of a design pattern called sentinel object. It acts like a datatype so that we can call methods on it ~~[illegible]~~ even when empty

if (lst!null) n=lst.size()

n = lst.size()

TYPE CHECKING

Like ~~[illegible]~~ every object oriented language, Java has different type checkings.

Compile time, ~~[illegible]~~ before the program runs

Run time, when the program is executing

~~At compile~~ compile time, every variable has a declared type. This is used by the compiler to [illegible] declared [illegible]

int 5 , double 4.5

double 9.5

The actual type is [illegible] at run-time.

string obj

~~[illegible]~~ ~~represent the~~

A parser for a textual language produces a tree shaped datatype called abstract syntax tree

~~[illegible]~~

The AST datatype definition can be shaped from grammar . . .

SAT

Given a boolean expression, find out for which boolean values it is true

The easier solution is to enumerate every ~~assignment~~ assignment and check against ~~the~~ every of them.

Given k value, the ~~[illegible]~~ of operation is 2^k, can we do better? hardly

Showed [illegible] in 1980, after SAT was defined NP Complete, showed that SAT problems are ~~[illegible]~~ mostly ~~[illegible]~~ very time easy. That worst case analysis brought us to a partially true conclusion, ~~[illegible]~~ however ~~most problem we could meet~~ there is a phase transition between hard problems and easy ones

However this won't work

Socrates ⇒ Human ∧ Mortal ⇒ Mortal ∧ ¬(Socrates ⇒ Mortal)

CONCURRENCY

- Multiple computers on a network
- Multiple application running on one computer
- Multiple processors in a computer

Even in programming with users, GUIs, apps

SHARED MEMORY model

Two processes sharing the same physical memory

A thread is a locus of control in a process (represents the current point in computation)

MESSAGE PASSING model

Two computers communicating by network connections

A process, is an instance of a running program, by default processes use ms. passing.

When a process works on same memory at the same time, there is a race condition and instructions are interleaved

correctness of the program depends on what the timing of events in concurrency

THE CORRECTNESS OF A PROGRAM SHOULD NOT DEPEND ON TIMING

To avoid race condition in shared memory, concurrent modules need to synchronize each other

- Locks are a common sync mechanism, locking means "I'm changing it, don't touch it now". The other module must wait in that case.

If A and B acquire the lock, there's a deadlock

Message passing

Modules interact through messages, requests are handled one at a time using a queue

Race conditions are not eliminated

DEBUGGING

It's very hard to discover those concurrency bugs, even if you spot a bug it's hard to know where it is.

Heisenbug (opposed to bohrbugs)

An approach ~~is to~~ build a lightweight event log and stop the program when bugs appear to examine the log

CLIENT/SERVER

It is a design pattern for message passing.

There different processes divided in server and client processes.

Clients sent request, the server responds and then client disconnects.

Example of clients could be web browsers for web servers, or Gmail for mail servers.

Usually the run in different machine, but it doesn't have to be.

SOCKETS

A network interface is identified by an IP address.
An interface has 65536 ports (0-65535).

A server binds to a port, client must know which port it is.
There are standard ports.

The listening port is just used to ~~establish~~ ~~new connection~~ accept incoming connections.
Once the connection is established, the server creates a new socket with a fresh port number

BUFFERS

Chunks of data of varying size are sent over the network.
The received "chunks" a reassembled into a buffer where it hold until the is read

BLOCKING

A blocked method is waiting for another event to occur.
Socket streams blocks
- Incoming socket's buffer empty, read() blocks
  Destination socket's buffer is full, write() blocks

WIRE PROTOCOLS

A protocol is a set of messages that can be exchanged between two communication

A wire protocol is a set of messages represented as byte sequences

Protocols are defined by RFC specifications

DEADLOCK

the system is deadlocked if there is a cycle in the network graph.
they can include many sockets.

The solution is to design a system with no possibility of cycles

Another approach is timeouts, if the process is blocked for too long, stop blocking and throw an exception

THREAD SAFETY

Recall race conditions? There are basically four ways to access a shared memory area safely

- Don't share. Also called confinement.
- Make the shared data immutable
- Encapsulate in a threadsafe datatype from java library that does coordination for you
- Use synchronization to build your own threadsafe datatype

Threads in Java, see tutorial

Thread confinement

Local variables are always confined in a stack, each thread has its own. Even with multiple invocations, each of them has its own copy in stack. Be careful, it might be a reference. If the referenced object is mutable, we need to confine it as well

Avoid globals

Unlike local vars, globals (called static in Java) are not thread confined. Try to avoid using them, else make sure it's used once at the time.

Threadsafe

A datatype is threadsafe if it behaves correctly when used from multiple threads

Immutability

We saw that a datatype could ~~be~~ have beneficent mutations when there are invisible mutations to its rep.

Even if the abstract is immutable, the ~~[illegible]~~ internal rep mutation might cause trouble.

In this case we must use locks and treat it as a mutable datatype.

A datatype is ~~[illegible]~~ immutable when:

- No mutator methods
- Fields are private or final
- No mutation of mutable objects
- No rep exposure.

Collections API provide wrapper methods that make collections immutable.
On those, method are made atomic, the method is not leaved until it finishes computing.

To get performance

- Create only few threads (take resources)
- Don't move work between threads unnecessarily

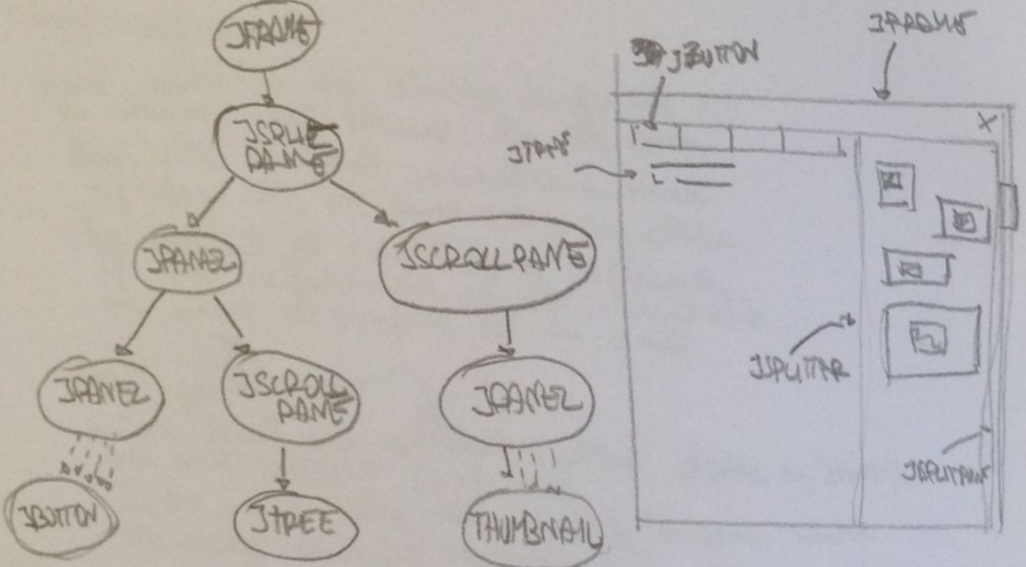
GUI

There are three main design pattern

- model view controller
- view tree
- listener

VIEW ~~TREE~~

The interface is composed of view objects each of them occupying a certain section of the screen, ~~[illegible]~~ the rectangular area is a bounding box. The view take different names depending on the UI toolkit.



Primitive views ~~don't~~ ~~used to group~~ contain other views

Composite views are use to group of modifying other views

Views are responsible for displaying themselves. The view tree directs the display process using a redraw algorithm

The view tree controls ~~The~~ bounding box using an automatic algorithm ~~that~~ ~~[illegible]~~ controls

INPUT

```
while (true) {
    read mouse click
    → based on where occur
    the view tree distributes
    the event through the
    hierarchy* /
    IFs
    ...
```

Mouse might be source and event is its ~~position~~ moves or buttons.

When it occurs, the source is distributed to all subscribe listeners ready to call their methods.

Event handling is an instance of ~~the~~ listener pattern

- a source ~~an event~~ generates a stream of events that could e change in state on the source
- One of more listeners register interest to the stream providing a function function for new events.

"Separation of concerns, output handled by views input handled by listeners"

FRONT-END AND BACKEND

We've seen the front-end, however an application is controlling data and logic in the background. That is called ~~[illegible]~~ backend

~~[illegible]~~ Model-view-controller pattern has this separation as primary goal ~~[illegible]~~. It puts the back end code into the model and front end into the view and controller.

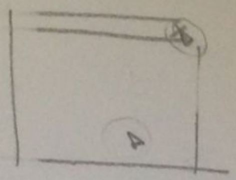
Input is handled by the controller, output by the view.

Models are mutable (often), and notifies its ~~clients~~ clients when there are changes so that views can update their display and controllers can respond (Listener Pattern)

So a view queries the model for data and draws it on the screen

Controller handles input, like mouse or keyboard events.

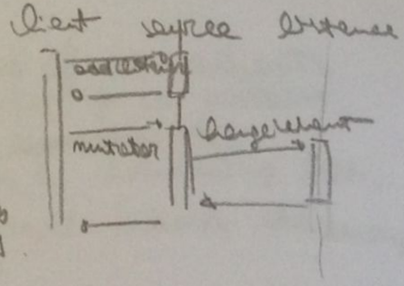
RISKS IN EVENTS



Control flow in an event-based program is not simple, ~~as~~ input events happen nondeterministically. This hidden flow leads to pitfalls

sequence diagram

Time flows downward, vertical lines represent objects while arrows represent calls, rectangles show when a method is active



Pitfall #1

The listener calls observer methods, ~~bcos~~ The listener reacts with the change on method by calling observers. Observer methods may occur while a mutator is active. If the mutator hasn't returned yet it's possible that the state is not yet ready. The observer might return garbage. This a concurrency problem!

Pitfall #2

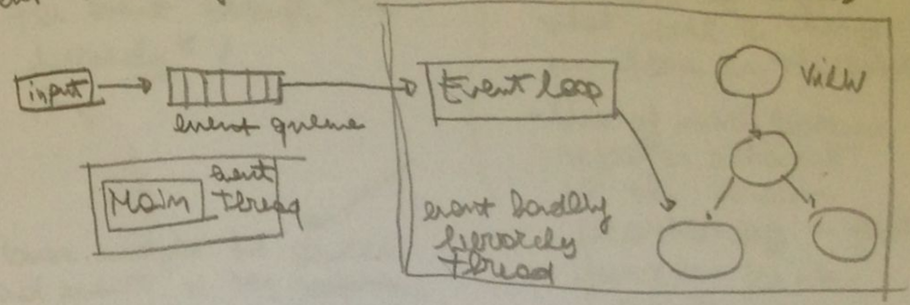
When a listener responds to an update message by calling the mutator on the model. Mutator [illegible] may occur while mutator is still in progress

Pitfall #3

Listener remove itself with removeListener. This is a problem if there is a collection of listeners and ~~you~~ are iterating over it. It's safer to use a copy.

BACKGROUND PROCESSING

If for example a button, more than event handling has the task to load a new view, the [illegible] interface would freeze until ~~the~~ it is done because repainting and event handling is done by the same thread.



Swing is multithreaded then, there's often a mutable datatype, the model. You can't modify the model without blocking the event hand. thread

[illegible]