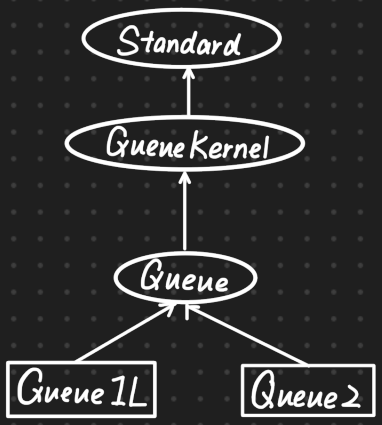
Final Review

2]. Queue

FIFO (first in first out)

generic type / parameterized type



Constructor :

① Queue<Integer> qi = new Queue1L<Integer>( );

② Queue<Integer> qi = new Queue1L<>( );

③ There is one constructor for each implementation class

④ The constructor has its own contract

Wrapper Type :

primitive type	wrapper type
boolean	Boolean
char	Character
int	Integer
double	Double

← immutable type

auto-boxing & auto-unboxing

Instance Method

①

enqueue

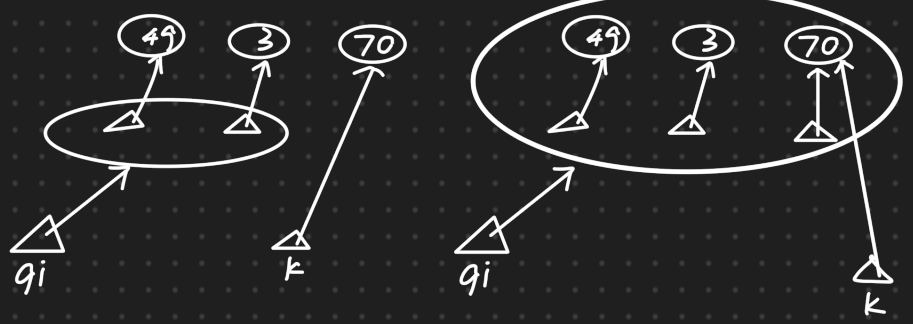
void enqueue(T x)

- Adds x at the back (right end) of **this**.
- Aliases: reference x
- Updates: **this**
- Ensures:

  this = #**this** * <x>

The list of references are advertised here because aliasing is important

	qi = <49, 3> ;
	k = 70 ;
qi.enqueue(k) ;	
	>>> qi = <49, 3, 70>



② dequeue

`T dequeue()`

- Removes and returns the entry at the front (left end) of **this**.
- Updates: **this**
- Requires:
  this `/= < >`
- Ensures:
  `#this = <dequeue> *` ***this***

	qi = <49, 3, 70> k = -584
k = qi.dequeue()	
	qi = <3, 70> k = 49

③ length

int `length()`

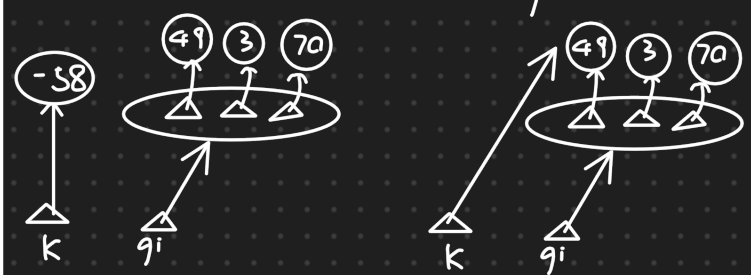
- Reports the length of **this**.
- Ensures:
  length = |***this***|

	qi = <49, 3, 70> k = -58
k = qi.front()	
	qi = <49, 3, 70> k = 49

④ front

`T front()`

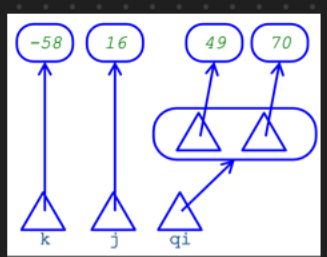
- Returns the entry at the the front (left end) of **this**.
- Aliases: reference returned by `front`
- Requires:
  this `/= < >`
- Ensures:
  `<front>` ***is prefix of this***

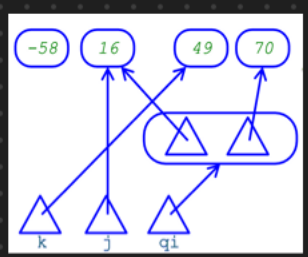


⑤ replaceFront

`T replaceFront(T x)`

- Replaces the front of **this** with `x`, and returns the old front.
- Aliases: reference `x`
- Updates: **this**
- Requires:
  this `/= < >`
- Ensures:
  `<replaceFront>` ***is prefix of #this and this*** `= <x> *` ***#this***`[1, |#this|)`





① Aliase

	qi = <49, 3, 70> k = -58 j = 16
k = qi.replaceFront(j)	
	qi = <16, 3, 70> k = 49 j = 16

② No Aliase (Swap j)

	qi = <49, 3, 70> j = 16
j = qi.replaceFront(j)	
	qi = <16, 3, 70> j = 49

⑥

append

`void append(Queue<T> q)`

- Concatenates ("appends") `q` to the end of `this`.
- Updates: `this`
- Clears: `q`
- Ensures:
  `this = #this * #q`

	q1 = <4, 3, 2> q2 = <1, 0>
q1.append(q2)	
	q1 = <4, 3, 2, 1, 0> q2 = <>

⑦

sort

`void sort(Comparator<T> order)`

- Sorts `this` according to the ordering provided by the `compare` method from `order`.
- Updates: `this`
- Requires:
  [the relation computed by order.compare is a total preorder] → any two values are comparable, there are no "cycles"
- Ensures:
  `this` = *[#this ordered by the relation computed by order.compare]*

→ No a>b>c>a

Comparators

- The Java interface `Comparator<T>` is:

```
public interface Comparator<T> {
  /**
   * Returns a negative integer, zero, or
   * a positive integer as the first
   * argument is less than, equal to, or
   * greater than the second.
   */
  int compare(T o1, T o2);
}
```

Creating a Comparator :

```
private static class IntergerLT
        implements Comparator<Integer> {
    @Override
    public int compare(Integer o1, Integer o2) {
        if (o1 < o2) {
            return -1;
        } else if (o1 < o2) {
            return 1;
        } else {
            return 0;
        }
    }
}
```

A class that implements Comparator is usually a nested class (declared for local use inside another class), if so should be declared as private static

Easy Comparator.

```
private static class IntergerLT implements Comparator<Integer> {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o1.compareTo(o2);
    }
}
```

A generic type must be reference type, and each wrapper type T implements the interface Comparable<T>, each has a compareTo method.

⑧

rotate

void rotate(**int** distance)

- Rotates **this**.
- Updates: **this**
- Ensures:

```
if #this = <> then
  this = #this
else
  this =
    #this[distance mod |#this|, |#this|) *
    #this[0, distance mod |#this|)
```

	qi = < 8, 6, 92, 3 >
qi. rotate(-1)	
	qi = < 3, 8, 6, 92 >

8. Set - Theory

① Mathematical sets : a mathmetical model

② A finite set can be think of as a collection of 0 or more elements of any other mathematical type T

- T is element type
- this math type is called finite set of T

③ No duplicate elements

No order → {1, 2, 3} = {3, 1, 2}

④ Mathematical Notation: ∈ ① 'G' is in {'G', 'o'} 1 is not in {2, 3}

∪ ② {1, 2} union {2, 3} = {1, 2, 3}

∩ ③ {1, 2} intersection {2, 3} = {2}

④ difference s\t or s-t : {1, 2, 3, 4} \ {3, 2} = {1, 4}

⊆ ⑤ s is subset of t

⊂ ⑥ s is proper subset of t (doesn't allow s=t)

⑦ entries (< 2, 2, 1, 2 >) = {1, 2}

29. Set

1. Mathematical Model : Set is modeled by finite set of T

2. Constructors : each of them has its own contract in kernel interface SetKernel

```
Set<Integer> s1 = new Set1L<>();
```

3. Methods : (Instance Method)

add() remove() removeAny() | contains() size() | add() remove() isSubset()

add

void add(T x)
- Adds x to **this**.
- Aliases: reference x
- Updates: **this**
- Requires: *x is not in this*
- Ensures: *this = #this union {x}*

remove

T remove(T x)
- Removes x from **this**, and returns it.
- Updates: **this**
- Requires: *x is in this* — whether x is an aliase or not, it doesn't matter
- Ensures: *this = #this \ {x} and remove = x*

removeAny

T removeAny()
- Removes and returns an arbitrary element from **this**.
- Updates: **this**
- Requires: *|this| > 0*
- Ensures: *removeAny is in #this and this = #this \ {removeAny}*

contains

boolean contains(T x)
- Reports whether x is in **this**.
- Ensures: *contains = (x is in this)*

it doesn't matter whether x is an aliase or not

add

void add(Set<T> s)
- Adds to **this** all elements of s that are not already in **this**, also removing just those elements from s.
- Updates: **this**, s
- Ensures: *this = #this union #s and s = #this intersection #s*

	s1 = {1, 2, 3, 4} s2 = {3, 4, 5, 6}
s1.add(s2)	
	s1 = {1, 2, 3, 4, 5, 6} s2 = {3, 4}

isSubset

boolean isSubset(Set<T> s)
- Reports whether **this** is a subset of s.
- Ensures: *isSubset = this is subset of s*

remove

Set<T> remove(Set<T> s)
- Removes from **this** all elements of s that are also in **this**, leaving s unchanged, and returns the elements actually removed.
- Updates: **this**
- Ensures: *this = #this \ s and remove = #this intersection s*

Overloading

① method with same name but different parameter profile (number, types, order of formal parameters)

② May not be overloaded on basis of its return type

③ Java disambiguate based on number → type → order at the point of call

4. Iterating Over a Set

① With removeAny

```
Set<T> temp = s.newInstance();
temp.transferFrom(s);
while (temp.size > 0) {
    T x = temp.removeAny();
    // do something
    s.add(x);
}
```

returns a new object with same object type as the receiver

① No need for copy, and transferFrom is more efficient
② set s to empty

② With Iterator (for-each loop)

iterator

`Iterator<T> iterator()`

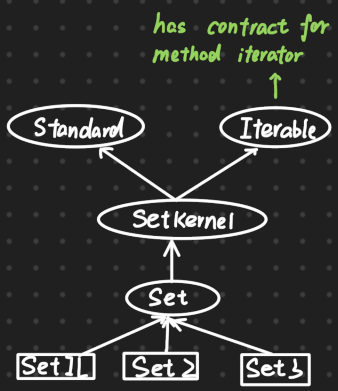
- Returns an iterator over a set of elements of type `T`.
- Ensures: finite set of T

`entries(~this.seen * ~this.unseen) = this` and

`|~this.seen * ~this.unseen| = |this|`

two mathematical variables:

① ~this.seen String of T
② ~this.unseen String of T




```
for (T x : s) {
    // do somethig with x, but do not change
    // the value of x or s
}
```

x is alised to a different element of s

if T is Integer / String,

change x will not change the set s

Properties: ① It introduces alises, so the loop body should not call any method on s

② If what you want to do to each element is to change it (when T is mutable type), the approach doesn't work cause the loop body should not change x

③ more efficient than removeAny (make no copies)

30. String - Reassembly

31. Sequence

Allow to manipulate strings of entries of any type through direct access by position similar to array

clear()	add()	entry()	flip()
newInstance()	remove()	replaceEntry()	insert()
transferFrom()	length()	append()	extract()

remove

`T remove(int pos)`

- Removes and returns the entry at position `pos` of `this`.
- Updates: `this`
- Requires:
  `0 <= pos and pos < |this|`
- Ensures:
  `this = #this[0, pos) * #this[pos+1, |#this|) and <remove> = #this[pos, pos+1)`

add

`void add(int pos, T x)`

- Adds `x` at position `pos` of `this`.
- Aliases: reference `x`
- Updates: `this`
- Requires:
  `0 <= pos and pos <= |this|`
- Ensures:
  `this = #this[0, pos) * <x> * #this[pos, |#this|)`

entry

`T entry(int pos)`

- Reports the entry at position `pos` of `this`.
- Aliases: reference returned by `entry`
- Requires:
  `0 <= pos and pos < |this|`
- Ensures:
  `<entry> = this[pos, pos+1)`

② swap

si = <49, 70>
z = -8

z = si.reE(1, z)

si = <49, -8>
z = 70

replaceEntry

`T replaceEntry(int pos, T x)`

- Replaces the entry at position `pos` of `this` with `x`, and returns the old entry at that position.
- Aliases: reference `x`
- Updates: `this`
- Requires:
  `0 <= pos and pos < |this|`
- Ensures:
  `this = #this[0, pos) * <x> * #this[pos+1, |#this|) and <replaceEntry> = #this[pos, pos+1)`

①

si = <49, 70>
z = -8
w = -584

w = si.reE(1, z)

si = <49, -8>
z = -8
w = 70

append

`void append(Sequence<T> s)`

- Concatenates ("appends") `s` to the end of `this`.
- Updates: `this`
- Clears: `s`
- Ensures:
  `this = #this * #s`

flip

`void flip()`

- Reverses ("flips") `this`.
- Updates: `this`
- Ensures:
  `this = rev(#this)`

insert

`void insert(int pos, Sequence<T> s)`

- Inserts `s` at position `pos` of `this`, and clears `s`.
- Updates: `this`
- Clears: `s`
- Requires:
  `0 <= pos and pos <= |this|`
- Ensures:
  `this = #this[0, pos) * #s * #this[pos, |#this|)`

extract

`void extract(int pos1, int pos2, Sequence<T> s)`

- Removes the substring of `this` starting at position `pos1` and ending at position `pos2-1`, and puts it in `s`.
- Updates: `this`
- Replaces: `s`
- Requires:
  `0 <= pos1 and pos1 <= pos2 and pos2 <= |this|`
- Ensures:
  `this = #this[0, pos1) * #this[pos2, |#this|) and s = #this[pos1, pos2)`

si1 = <8, 6, 92, 0>
si2 = <1, -7, 562>

si1.extract(1, 3, si2)

si1 = <8, 0>
si2 = <6, 92>

32. Stack

LIFO (last in first out)

1. Methods

clear()
newInstance()
transferFrom()
push()
pop()
length()
top()
replaceTop()
flip()

	si = <3, 7o> k = 49
si.push(k)	
	si = <49, 3, 7o> k = 49

push

`void push(T x)`

- Adds x at the top (left end) of **this**.
- Aliases: reference x
- Updates: **this**
- Ensures:
  this = <x> * ***#this***

pop

`T pop()`

- Removes and returns the entry at the top (left end) of **this**.
- Updates: **this**
- Requires:
  this /= < >
- Ensures:
  #this = <pop> * ***this***

	si = <49, 7o> k = -58 j = 16
k = si.reT(j);	
	si = <16, 7o> k = 49 j = 16

replaceTop

`T replaceTop(T x)`

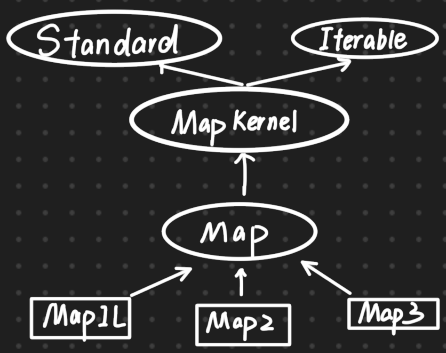
- Replaces the top of **this** with x, and returns the old top.
- Aliases: reference x
- Updates: **this**
- Requires:
  this /= < >
- Ensures:
  <replaceTop> ***is prefix of #this and this*** = <x> * ***#this***[1, |***#this***|)

	si = <49, 7o> j = 16
j = si.reT(j)	
	si = <16, 7o> j = 49

34 Map

K V partial function from K to V

type *Map* **is modeled by**
PARTIAL_FUNCTION



add

void add(K key, V value)

- Adds the pair (key, value) to **this**.
- Aliases: references key, value
- Updates: **this**
- Requires:
  key ***is not in*** *DOMAIN(**this**)*
- Ensures:
  this *=* ***#this union*** *{(key, value)}*

Map.Pair<String, Integer> p =
m.remove(k)

Map.Pair Methods

- This (immutable) type has only a constructor (taking a K and a V) and a ***getter*** method for each pair component
  - K key()
    - Returns the first component of **this**
    - Aliases: reference returned by key
  - V value()
    - Returns the second component of **this**
    - Aliases: reference returned by value

remove

Map.Pair<K,V> remove(K key)

- Removes from **this** the pair whose first component is key and returns it.
- Updates: **this**
- Requires:
  key ***is in*** *DOMAIN(**this**)*
- Ensures:
  remove.key = key ***and***
  remove ***is in*** *#**this*** ***and***
  this *= #**this** \ {remove}*

removeAny

Map.Pair<K,V> removeAny()

- Removes and returns an arbitrary pair from **this**.
- Updates: **this**
- Requires:
  *|**this**| > 0*
- Ensures:
  removeAny ***is in*** *#**this*** ***and***
  this *= #**this** \ {removeAny}*

value

`V value(K key)`

- Reports the value associated with `key` in **`this`**.
- Aliases: reference returned by `value`
- Requires:
  key ***is in*** *DOMAIN(****this****)*
- Ensures:
  (key, value) ***is in this***

hasKey

`boolean` `hasKey(K key)`

- Reports whether there is a pair in **`this`** whose first component is `key`.
- Ensures:
  hasKey =
  (key ***is in*** *DOMAIN(****this****))*

replaceValue

`V replaceValue(K key, V value)`

- Replaces the value associated with `key` in **`this`** by `value`, and returns the old value.
- Aliases: reference `value`
- Updates: **`this`**
- Requires:
  key ***is in*** *DOMAIN(****this****)*
- Ensures:
  this *= (#****this*** *\ {(key, replaceValue)})*
  union *{(key, value)}* ***and***
  (key, replaceValue) ***is in*** *#****this***

key

`K key(V value)`

- Reports some key associated with `value` in **`this`**.
- Aliases: reference returned by `key`
- Requires:
  value ***is in*** *RANGE(****this****)*
- Ensures:
  (key, value) ***is in this***

hasValue

`boolean` `hasValue(V value)`

- Reports whether there is a pair in **`this`** whose second component is `value`.
- Ensures:
  hasValue =
  (value ***is in*** *RANGE(****this****))*

combineWith

`void` `combineWith(Map<K,V> m)`

- Combines `m` with **`this`**.
- Updates: **`this`**
- Clears: `m`
- Requires:
  *DOMAIN(****this****)* ***intersection***
  DOMAIN(m) = {}
- Ensures:
  this *= #****this*** ***union*** *#m*

combineWith

`void` `combineWith(Map<K,V> m)`

- Combines `m` with **`this`**.
- Updates: **`this`**
- Clears: `m`
- Requires:
  *DOMAIN(****this****)* ***intersection***
  DOMAIN(m) = {}
- Ensures:
  this *= #****this*** ***union*** *#m*

Iterator

```
NN raise = new NN2(10000);
for ( Map.Pair<String, NN> p : m) {
    NN salary = p.value();
    salary.add(raise);
}
```

Violate the rule of iterator

The Safe Way

- Here's how you *should* give every employee a $10,000 raise:

```
NaturalNumber raise = new NaturalNumber2(10000);
Map<String, NaturalNumber> temp = m.newInstance();
temp.transferFrom(m);
while (temp.size() > 0) {
  Map.Pair<String, NaturalNumber> p =
    temp.removeAny();
  p.value().add(raise);
  m.add(p.key(), p.value());
}
```

35. GUI - Graphical User Interfaces

User Interaction Problem

end-user can change the "state" of any active user interface widget

user interaction includes { keyboard
any other input device

Terminology : ① event : the act of user manipulating a widget

② subject : the widget the user has manipulated

③ observer / listener for the subject : the object in program that need to do something in response to the event for a particular subject

The Observer Pattern

one of many object-oriented design pattern that address OOP issue

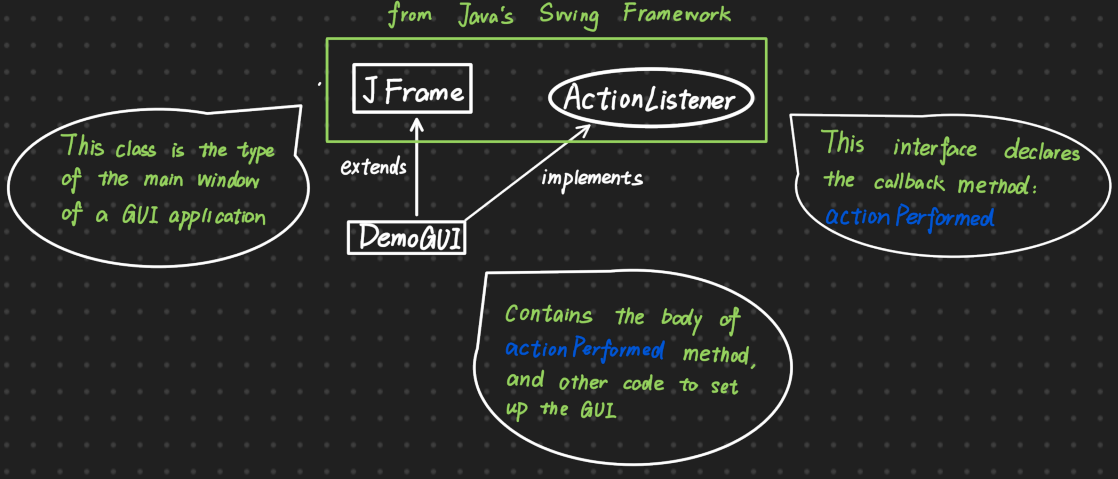
① each subject expect each observer to register (usually done by calling a method of the subject) itself with that subject if it's interested in the subject's event

② Each subject keeps track of its own set (can be stored in a set variable) of interested observers

③ Whenever an event occurs, the subject invokes a specific callback method for each registered observer, passing an event argument that describes the event

the method is described in an interface that any potential observer must implement

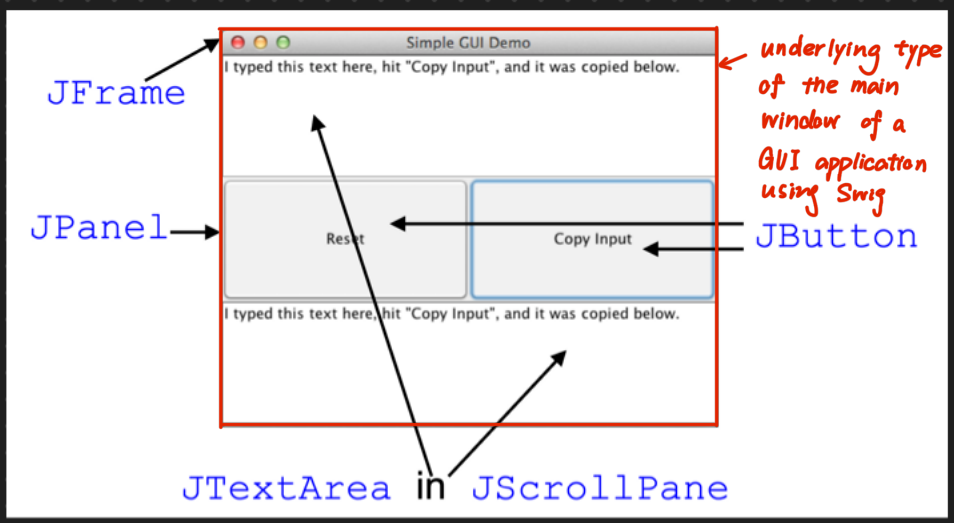
Simple GUI Demo



```
interface ActionListener {
  void actionPerformed(ActionEvent e);
}

interface ActionEvent {
  Object getSource();
  ...
}
```

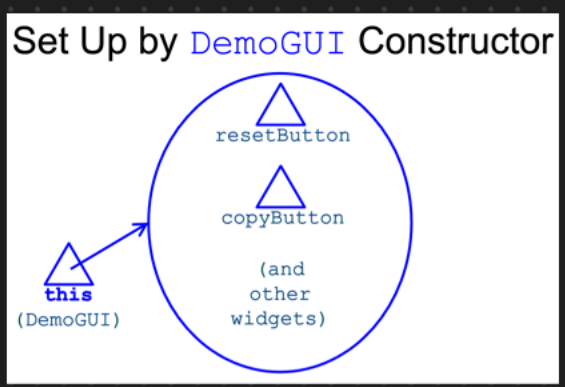
every class extends Object

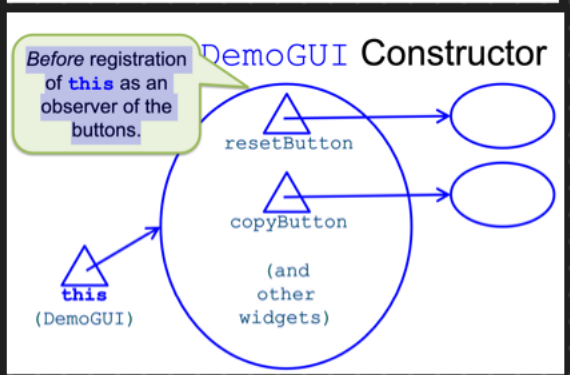


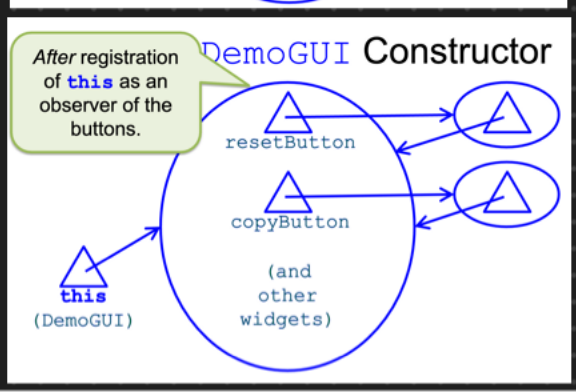
JFrame content pane
- JPanel
  - JButton
- JScroll Pane
  - JText Area

Instance Variables

- Variables can be declared:
  - in method bodies: ***local variables***
  - in method headers: ***formal parameters***
  - in classes: ***fields*** or ***instance variables***
- Examples of instance variables:
  - resetButton, copyButton, inputText, outputText, input, output
- Instance variables are essentially *global* variables that are shared by and can be accessed from all instance methods in the class



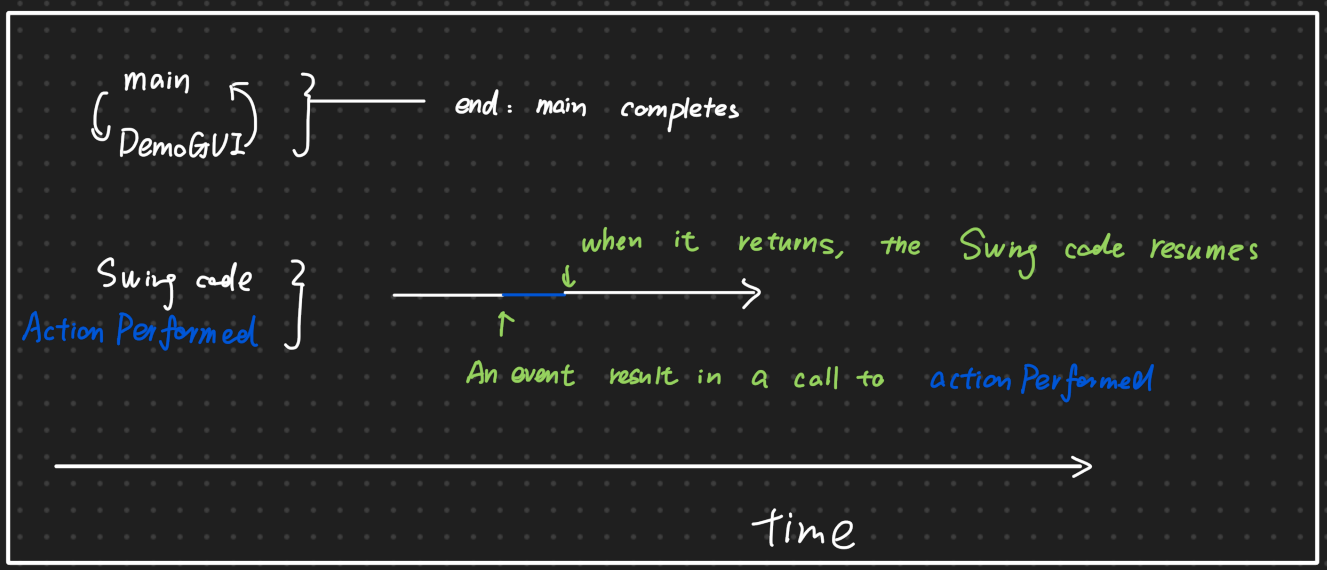




when DemoGUI is executed
- DemoGUI.main starts execution
  - Constructor for DemoGUI is called by main
  - Constructor for DemoGUI returns to main
- DemoGUI.main finishes execution

Threads

- A standard Java program executes in a thread
- A GUI program with Swing use at least 2 threads
  - Initial Thread executes main (until it complete)
  - An event dispatch thread executes everything else, including actionPerformed



Layout Manager

A ***layout manager*** allows you to arrange widgets without providing specific location coordinates

- `GridLayout` (simplest?; used in `DemoGUI`)
- `FlowLayout` (default for `JPanel`)
- `BorderLayout` (default for `JFrame`)
- ...

Java GUI Packages

A ***layout manager*** allows you to arrange widgets without providing specific location coordinates

- `GridLayout` (simplest?; used in `DemoGUI`)
- `FlowLayout` (default for `JPanel`)
- `BorderLayout` (default for `JFrame`)
- ...

Java Swing Widgets

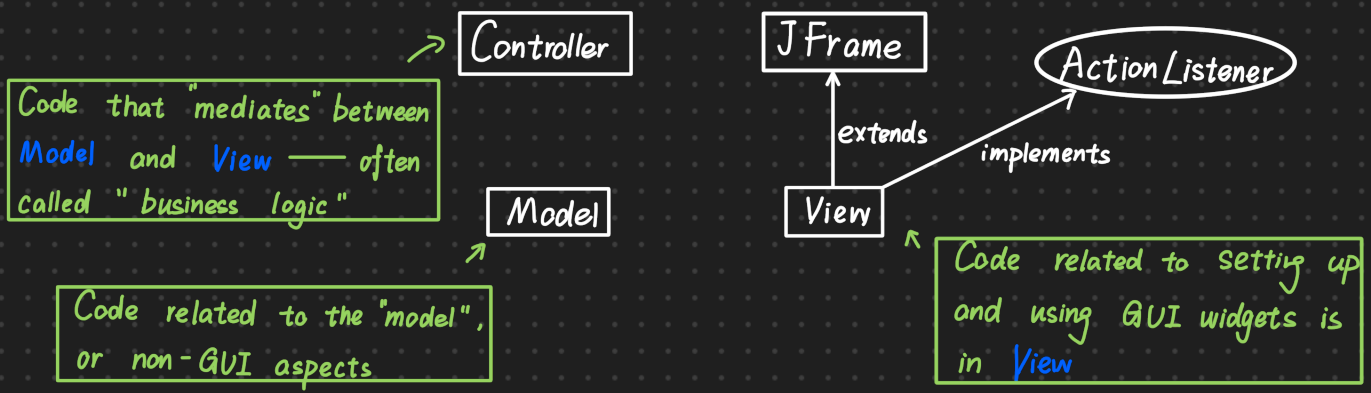
A ***layout manager*** allows you to arrange widgets without providing specific location coordinates

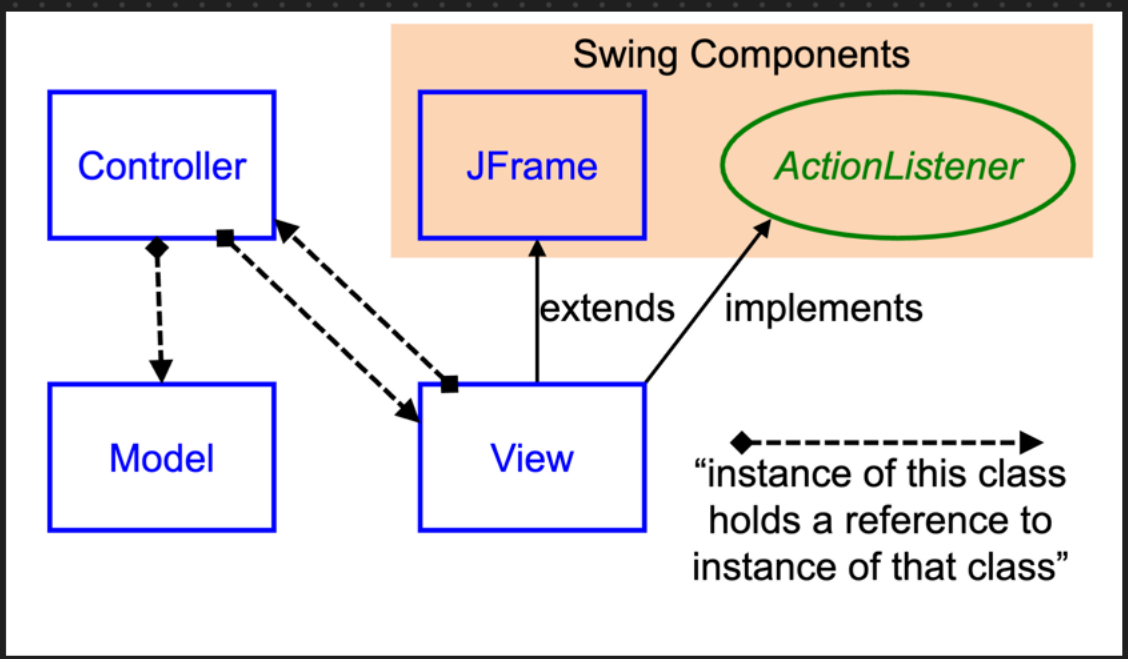
- `GridLayout` (simplest?; used in `DemoGUI`)
- `FlowLayout` (default for `JPanel`)
- `BorderLayout` (default for `JFrame`)
- ...

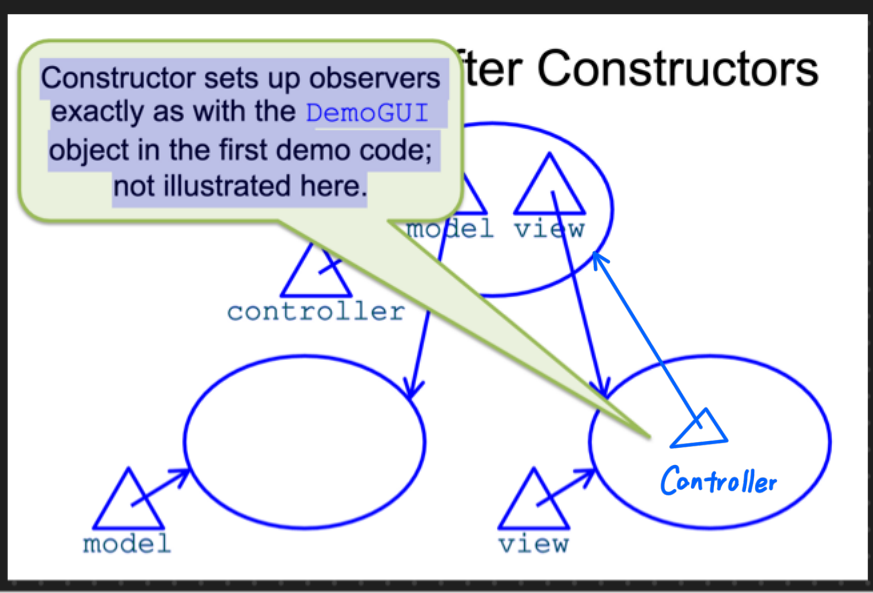
36. MVC - Model View Controller

MVC Design Pattern

- Dominant approach to organizing software with GUIs is model - view - controller design pattern

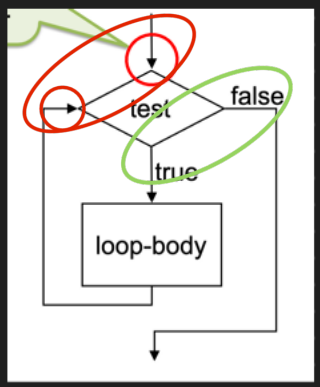






37. Loop Invariants

1. Loop Invariant describes what while loop do ⟺ Contracts describe what method call does
   - Invariant: a property that is true every time the code reaches a certain point, —— the loop condition test
   - Precondition: true before call
   - Postcondition: true after call



The loop invariant is true both ① before loop begin ② after every execution

Whatever is true is also true after the loop terminates

Loop invariant is true at two red points → "test" update nothing → Loop invariant is true at two green points

2.

```
/**
 * @updates this, q
 * @maintains
 * this * q = #this * #q
 * @decreases
 * |q|
 */
while (q.length() > 0) {
  T x = q.dequeue();
  this.enqueue(x);
}
```

Any variable not listed is restores - mode variable

loop invariant

progress metric

3. Best practice: ① test should not update any variable

4. Can help avoid: ① off-by-one error
   ② Wrong / Missing code in the loop body
   ③ Declaration of variables outside the loop that are only used inside the loop