Data Structures and Algorithms

XMUT-COMP 103 - 2025 T1

Collections and Stack

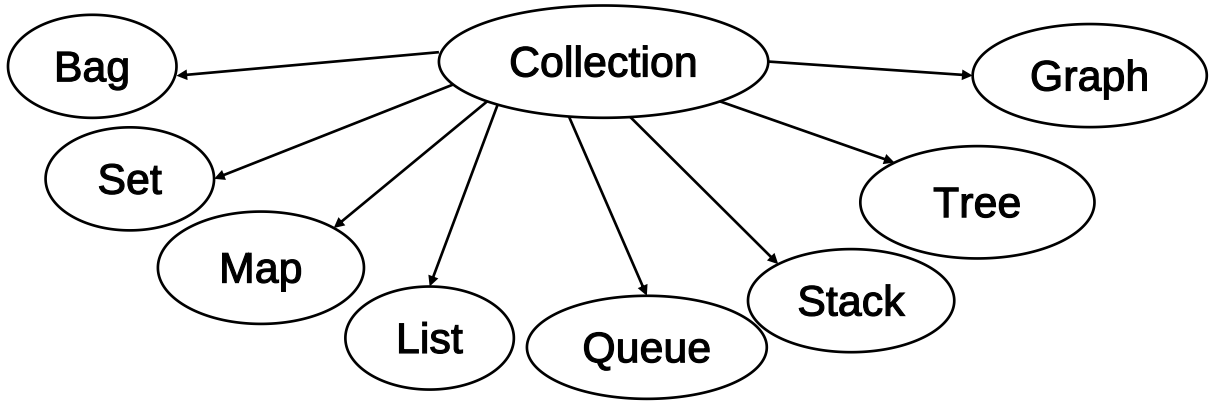
Felix Yan

School of Engineering and Computer Science

Victoria University of Wellington

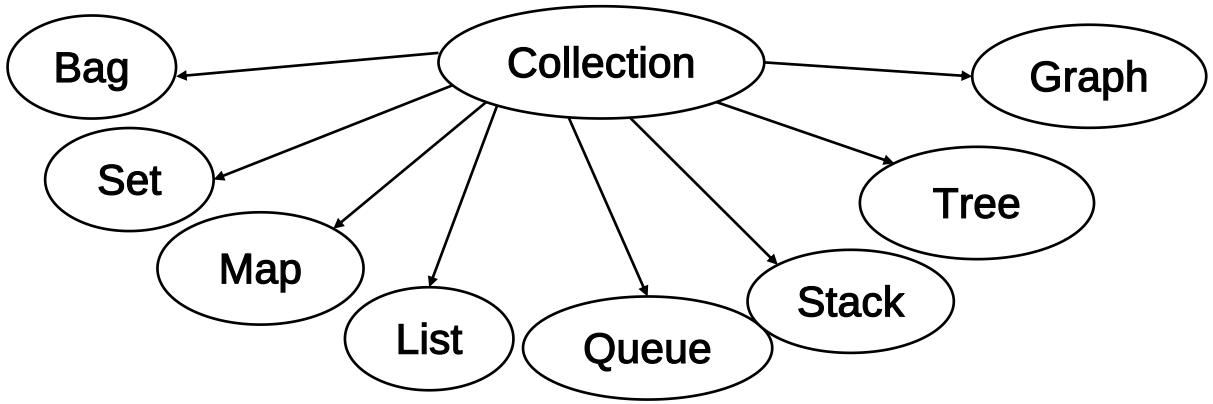
Collections: What's the difference

- Different structures
  - No structure – just a collection of values
  - Linear structure of values – the order matters
  - Set of key-value pairs
  - Hierarchical structures
  - Grid/table
  - ….
- Different constraints
  - duplicates allowed/not allowed
  - get, put, remove anywhere
  - get, put, remove only at the ends, or only at the top, or …
  - get, put, remove by position, or by value, or by key, or …
  - ….



Essential structures and constraints:

- Bags:
  - No structure
- Sets:
  - No structure, no duplicates
- Lists:
  - Linear structure of values. Access anywhere (by position)
- Stacks:
  - Linear structure of values. Access on at top (Last in => first out)
- Queues:
  - Linear structure of values. Add to the tail, remove from front (First in => first out)
- Maps:
  - Set of key-value pairs



Abstract Data Types

Set, Bag, Queue, List, Stack, Map, etc are

Abstract Data Types

- an ADT is a type of data, described at an abstract level:
  - Specifies the **operations** that can be done to an object of this type
  - Specifies how it will **behave**.

- Doesn't specify how it is implemented underneath – "black box"
  - though we will always need some concrete implementation of it.

Eg: Set ADT

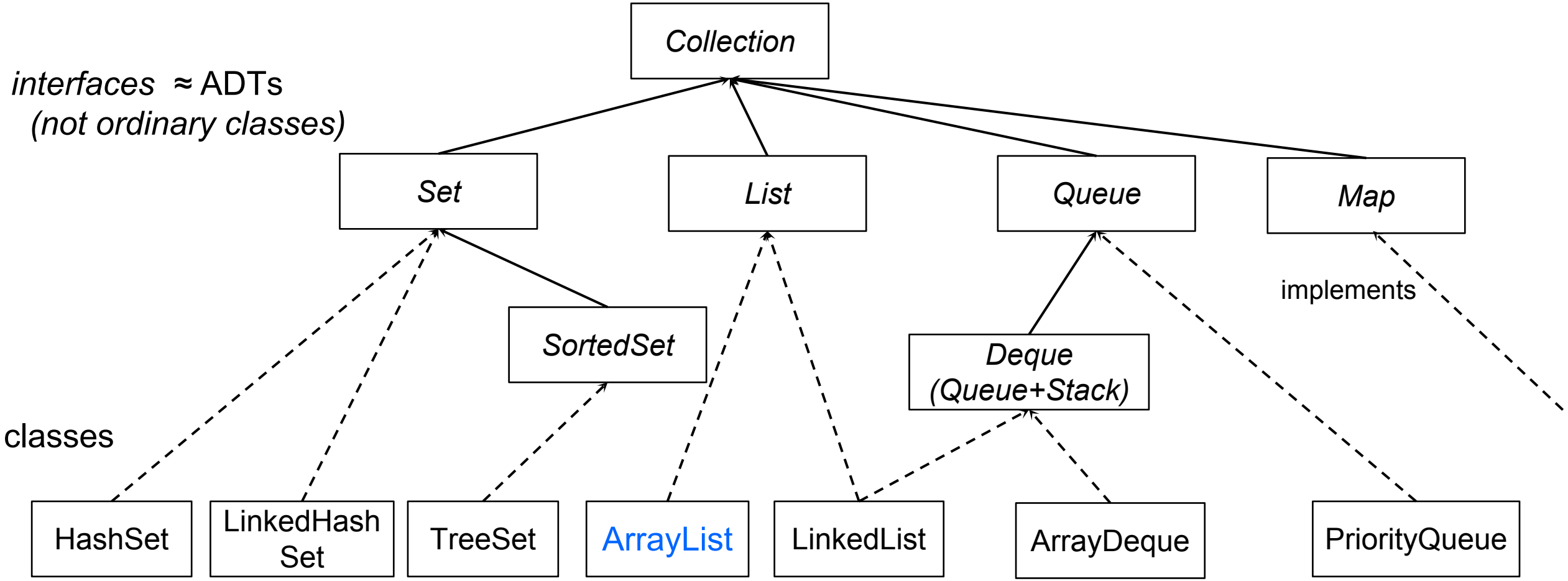
(simple version)

- Conceptual:
  - Collection of items with no structure and no duplicates.
- Operations:
  - **add(***value* **)**,
  - **remove(***value* **)**,
  - **contains(***value* **)** → *boolean*
- Behaviour:
  - A new set contains no values.
  - A set will contain a value *iff* the value has been added to the set and it has not been removed since adding it.
  - A set will not contain a value *iff* the value has never been added to the set, or it has been removed from the set and has not been added since it was removed.

Java Collections Library (in java.util)

Defines interfaces ⇒ Abstract Data Types

and classes:



Java Collections library

Interfaces:

- *Collection*
  = "Bag" (most general)
- *List*
  = ordered collection
- *Set*
  = unordered, no duplicates
- *Queue*
  ordered collection, limited access
  (add at one end, remove from other)
- *Map*
  = key-value pairs (or mapping)
- …

Specify the Types:

Classes

- List classes:
  ArrayList, LinkedList, [Stack]
- Set classes:
  HashSet, TreeSet, EnumSet,
  LinkedHashSet,…
- Queue classes:
  ArrayDeque, LinkedList, PriorityQueue
- Map classes:
  HashMap, TreeMap, EnumMap,
  LinkedHashMap, WeakHashMap, ...

Implement the interfaces

- Each implementation has advantages and disadvantage.

Java Interfaces and ADT's

- A Java ***Interface*** corresponds to an Abstract Data Type
  - Specifies what methods can be called on objects of this type (specifies name, parameters and types, and type of return value)
  - Behaviour of methods is only given in comments (but cannot be enforced)

✕ No constructors - can't make an instance: ~~**new** Set()~~ ~~**new** List()~~

✕ No fields - doesn't say how to store the data

Type variable – stands for whatever it is a set of..

```
public interface Set <E> {
    public boolean add(E item);        /*…description…*/
    public boolean remove(E item);     /*…description…*/
    public boolean contains(E item);   /*…description…*/
    …
        // (plus lots more methods in the Java Set interface)
```

Using Java Collection library

- Your program can
  - Declare a variable, parameter, or field of the interface type

```
private  List <String>  images;                    // defined using the ADT – interface
Set <Student>  students;
```

  - Create a collection object

```
images= new ArrayList <String> ();        // constructed using a class
students= new HashSet <Student> ();
```

  - Call methods on that variable, parameter, or field

```
images.add(UIFileChooser.open("Choose an image file");
images.set(i, images.remove(j));
Collections.fill(images, “sunset.jpg");
if (students.contains(st)){ ….
```

Why?

- Why use the Interface type to declare the field/variable then use a class to make the object?

- Can't make an object of the interface type:

```
List <Double> myNumbers = new List <Double>();
```

- Why not just use the class?

```
ArrayList <Double> myNumbers = new ArrayList <Double>();
```

- More flexible design to use the Interface type:

```
List <Double> myNumbers = new ArrayList <Double>();
```

Could change the class later; rest of program works on any kind of List.

Stack is an exception

- There is no Interface for Stack, just a class:

```
Stack <Action>  undoStack = new Stack <Action> ();        // constructed using a class
```

- Stacks have four stack operations:
  - empty() -> boolean is the stack empty or not
  - push(item) push an item onto the top of the stack
  - pop() -> item removes and returns the item at the top of the stack
    (error if the stack is empty)
  - peek() -> item returns the item at the top of the stack (without removing it)
    (error if the stack is empty)

- Better, use the Deque (double-ended queue) interface and
  the ArrayDeque class (has same push, pop, and peek, plus offer and poll)

Comments on code style for 103

- I will drop “this.” except when needed.
  - instead of this.loadFromFile(fname)
    just loadFromFile(fname)
  - instead of this.shapes.addShape(shape)
    just shapes.addShape(shape)

  - When is "this" needed? if a local variable or parameter has the same name as a field.

Lists vs. Stack

- In COMP102 arrays and ArrayLists were often used to store data in list. In a list the elements can be accessed using the position of the element

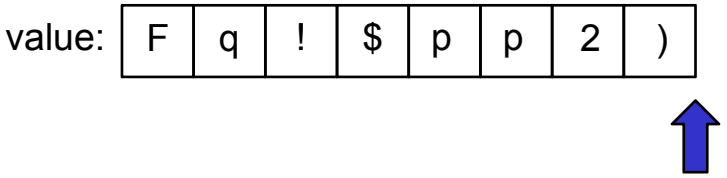
value:	F	q	!	$	p	p	2	)
pos:	0	1	2	3	4	5	6	7

Lists vs. Stack

- In COMP102 arrays and ArrayLists were often used to store data in list. In a list the elements can be accessed using the position of the element

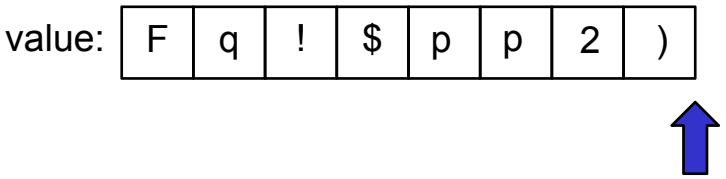
value:	F	q	!	$	p	p	2	)
pos:	0	1	2	3	4	5	6	7

- Using a stack: Elements can only be accessed at the end in the line of elements



Lists vs. Stack

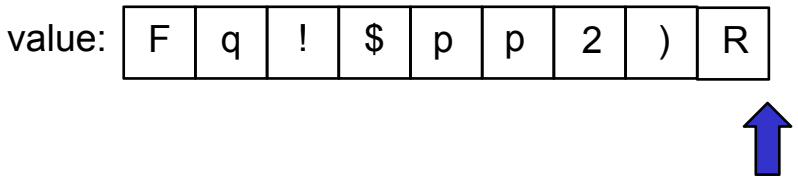
- Using a stack elements can only be accessed at the end in the line of elements



push(‘R’)

Lists vs. Stack

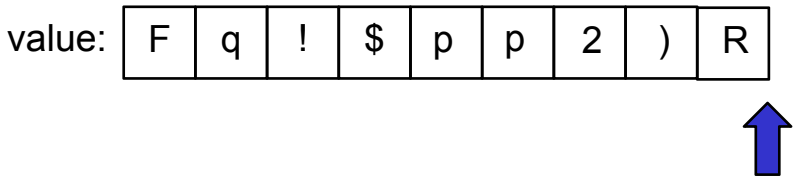
- Using a stack elements can only be accessed at the end in the line of elements



push(‘R’)

Lists vs. Stack

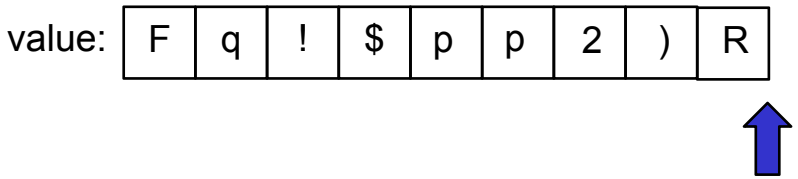
- Using a stack elements can only be accessed at the end in the line of elements



peek()

Lists vs. Stack

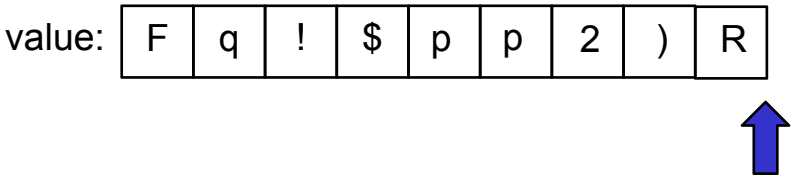
- Using a stack elements can only be accessed at the end in the line of elements



peek() => returns ‘R’ and nothing is changed on the stack.

Lists vs. Stack

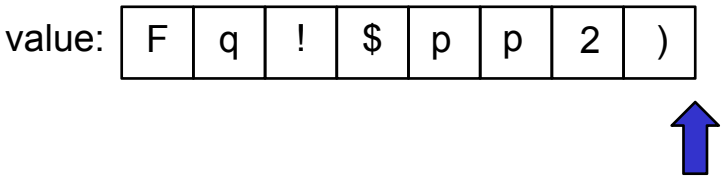
- Using a stack elements can only be accessed at the end in the line of elements



pop()

Lists vs. Stack

- Using a stack elements can only be accessed at the end in the line of elements



pop() => returns ‘R’, which is removed from the stack.

Lists vs. Stack

- This is an extremely useful behaviour
  - Any time you want events happening in the reverse order, e.g.
    - reverse a word
    - undo functionality
  - Check matching events are correct, e.g.
    - check if braces in a statement is used correctly

Lists vs. Stack

- Reverse a word

value:

Lists vs. Stack

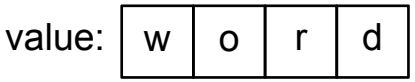
- Reverse a word

value: w

push(‘w’)

Lists vs. Stack

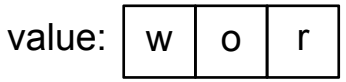
- Reverse a word



push(‘w’)
push(‘o’)
push(‘r’)
push(‘d’)

Lists vs. Stack

- Reverse a word



push(‘w’)
push(‘o’)
push(‘r’)
push(‘d’)

str = pop()
d

Lists vs. Stack

- Reverse a word

value:

push(‘w’)
push(‘o’)
push(‘r’)
push(‘d’)

str = pop() + pop() + pop() + pop();
d r o w

Lists vs. Stack

- Check if (([ ]([])) is a correct use of brackets

value:

- For each bracket in order
  - If opening bracket push(opening bracket) unto stack
  - If closing bracket pop stack
    - If returned bracket is opening version of closing bracket ok for now, continue
    - else error! //including no brackets, i.e. an empty stack
- At end of string
  - if the stack is empty then the brackets are correct
  - else error!

Lists vs. Stack

- Check if (([ ]([])) is a correct use of brackets

value: 

- For each bracket in order
  - If opening bracket push(opening bracket) unto stack
  - If closing bracket pop stack
    - If returned bracket is opening version of closing bracket continue
    - else error! //including no brackets, i.e. an empty stack
- At end of string
  - if the stack is empty then the brackets are correct
  - else error!

Lists vs. Stack

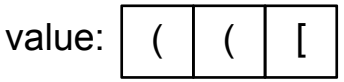
- Check if (([ ][])) is a correct use of brackets

value: [( ]

- For each bracket in order
  - If opening bracket push(opening bracket) unto stack
  - If closing bracket pop stack
    - If returned bracket is opening version of closing bracket continue
    - else error! //including no brackets, i.e. an empty stack
- At end of string
  - if the stack is empty then the brackets are correct
  - else error!

Lists vs. Stack

- Check if (([ ]([])) is a correct use of brackets

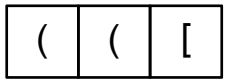


- For each bracket in order
  - If opening bracket push(opening bracket) unto stack
  - If closing bracket pop stack
    - If returned bracket is opening version of closing bracket continue
    - else error! //including no brackets, i.e. an empty stack
- At end of string
  - if the stack is empty then the brackets are correct
  - else error!

Lists vs. Stack

- Check if (([ ]([])) is a correct use of brackets

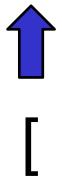
value:



- For each bracket in order
  - If opening bracket push(opening bracket) unto stack
  - If closing bracket pop stack
    - If returned bracket is opening version of closing bracket continue
    - else error! //including no brackets, i.e. an empty stack
- At end of string
  - if the stack is empty then the brackets are correct
  - else error!

Lists vs. Stack

- Check if (([ ][])) is a correct use of brackets



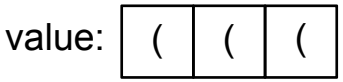
[

value: | (| (|

- For each bracket in order
  - If opening bracket push(opening bracket) unto stack
  - If closing bracket pop stack
    - If returned bracket is opening version of closing bracket continue
    - else error! //including no brackets, i.e. an empty stack
- At end of string
  - if the stack is empty then the brackets are correct
  - else error!

Lists vs. Stack

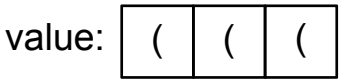
- Check if (([ ](])) is a correct use of brackets



- For each bracket in order
  - If opening bracket push(opening bracket) unto stack
  - If closing bracket pop stack
    - If returned bracket is opening version of closing bracket continue
    - else error! //including no brackets, i.e. an empty stack
- At end of string
  - if the stack is empty then the brackets are correct
  - else error!

Lists vs. Stack

- Check if (([ ](])) is a correct use of brackets



- For each bracket in order
  - If opening bracket push(opening bracket) unto stack
  - If closing bracket pop stack
    - If returned bracket is opening version of closing bracket continue
    - else error! //including no brackets, i.e. an empty stack
- At end of string
  - if the stack is empty then the brackets are correct
  - else error!

Undo (for Sokoban assignment)

How does it work?

- Have to keep a record of all the actions as they are done
  - Have to keep enough information to be able to undo them later

- Undo button steps backwards through the record of actions, undoing the next one.

What kind of collection do we need for the record of actions?

What kind of object do we need for each action?

Brick Builder

(run the program)

- What are the actions to remember?
- What info do we need to remember for each action?
- To incorporate undo:
  - create a new class to store all the information for each undo record
    - fields
    - constructors
    - getters
  - make a stack of undo record
  - at each action in the program, add a new record to the stack
  - add an undo button and method which pops the top record from the stack and undoes the action

Java: if *and* if … else LDC 4.2

- Two forms of the **if** statement:

```
if ( ⟨condition⟩ ) {
    ⟨actions to perform if condition is true⟩
}
```

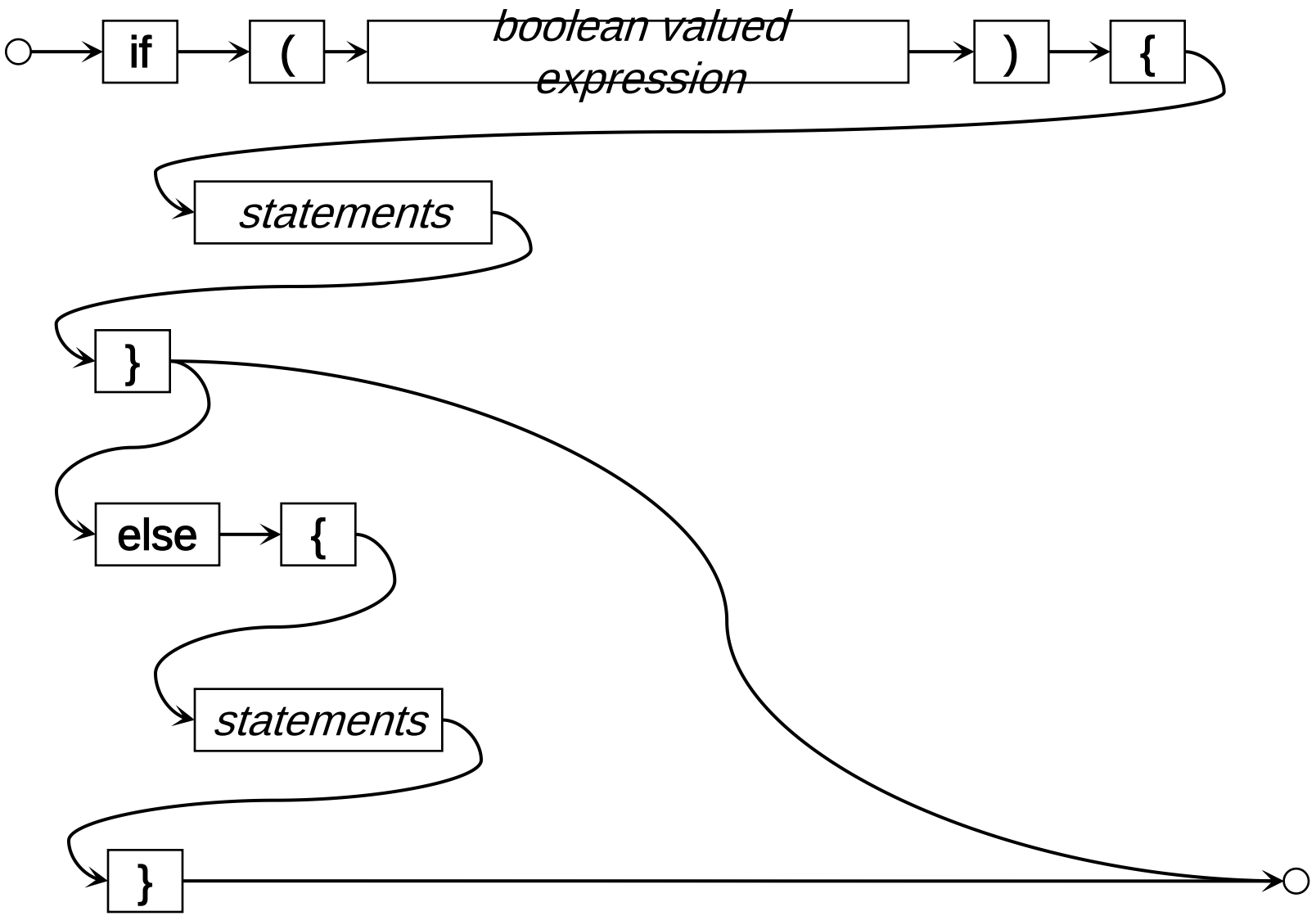
⇒ just skip the actions when the condition is not true !

and

```
if ( ⟨condition⟩ ) {
    ⟨actions to perform if condition is true⟩
}
else {
    ⟨actions to perform if condition is false⟩
}
```

Note: the { … } represent a "Block" – a sequence of actions that are wrapped up together into a single statement.

if … *vs* if … else …



Method with a condition

```
/** Ask for amount and currency; print note if –ve,  print value.*/
public void  convertMoney( ) { ;
    double amount = UI.askDouble("Enter amount $NZ");
    if ( amount < 0 ) {
        UI.println("Note: you have entered a debt!");
    }

    String currency = UI.askString ("Enter currency (US or Aus)");

    if ( currency.equals("US")  ) {
        UI.printf("$NZ%.2f  = $US%.2f\n", amount,  (amount * 0.668));
    }
    else  {
       UI.printf("$NZ%.2f  = $AUS%.2f\n", amount,  (amount * 0.893));
    }
}
```

Multiway choice: if … else if … else if …

- Can put another **if** statement in the **else** part:

```
if ( ⟨condition1⟩ ) {
   ⟨actions to perform if condition1 is true⟩
      :
}
else if ( ⟨condition2⟩ ) {
   ⟨actions to perform if condition 2 is true (but not condition 1)⟩
      :
}
else if ( ⟨condition3⟩ ) {
   ⟨actions to perform if condition 3 is true (but not conditions 1, 2)⟩
      :
}
else {
   ⟨actions to perform if other conditions are false⟩
      :
}
```

Example with multiway choice

```
public void  convertMoney( ) {
    double amount = UI.askDouble("Enter amount");
    if (amount < 0 ) {
        UI.println("Note: you have entered a debt!");
    }
    String currency = UI.askString("Enter currency (US or Aus)");
    if (currency.equals("US")  ) {
        UI.printf("$NZ%.2f  = $US%.2f\n", amount , amount * 0.668);
    }
    else if ( currency.equals("Aus") ) {
       UI.printf("$NZ%.2f  = $AUS%.2f\n", amount , amount * 0.893);
    }
    else  {
       UI.printf("I cannot convert to %s currency\n", currency);
    }
}
```

Switch Statements

- Java has a second structure for conditionals: **switch** ... **case** … **->** **{**…**}** … **default**
  - Handles multi-way choice more neatly
  - BUT restricted to very simple choices with integers, characters, and Strings

```
switch  ( day )   {
    case 1, 2, 3, 4, 5  -> {
        UI.printf("Pay = $ %.2f \n", RATE * hours);
    }
    case 6 -> {
        UI.printf("Pay = $ %.2f  ( time-and-a-half) \n", RATE * hours * 1.5);
    }
    case 7 -> {
        UI.printf("Pay = $ %.2f  ( double-time) \n", RATE * hours * 2);
    }
    default -> {
        UI.println(" Day must be between 1 and 7 ");
    }
}
```

The choice tests whether the switch value matches the literal values

Switch Statements

- **switch** does the comparison – no need for == or .equals(…)

If day is a String

```
switch  ( day )   {
    case  "Mon",  "Tue", "Wed", "Thu", "Fri"  -> {
        UI.printf("Pay = $ %.2f \n", RATE * hours);
    }
    case "Sat" -> {
        UI.printf("Pay = $ %.2f  ( time-and-a-half) \n", RATE * hours * 1.5);
    }
    case "Sun" -> {
        UI.printf("Pay = $ %.2f  ( double-time) \n", RATE * hours * 2);
    }
    default -> {
        UI.println(" Day must be between 1 and 7 ");
    }
}
```

Switch Statements

- **switch** does the comparison – no need for == or .equals(…)

```
if  ( day.equals("Mon") || day.equals("Tue") || day.equals("Wed") || day.equals("Thu") ||
      day.equals( "Fri") {
    UI.printf("Pay = $ %.2f \n", RATE * hours);
}
else if  (day.equals("Sat") ) {
    UI.printf("Pay = $ %.2f  ( time-and-a-half) \n", RATE * hours * 1.5);
}
else if  (day.equals("Sun") )  {
    UI.printf("Pay = $ %.2f  ( double-time) \n", RATE * hours * 2);
}
else  {
    UI.println(" Day must be between 1 and 7 ");
}
```

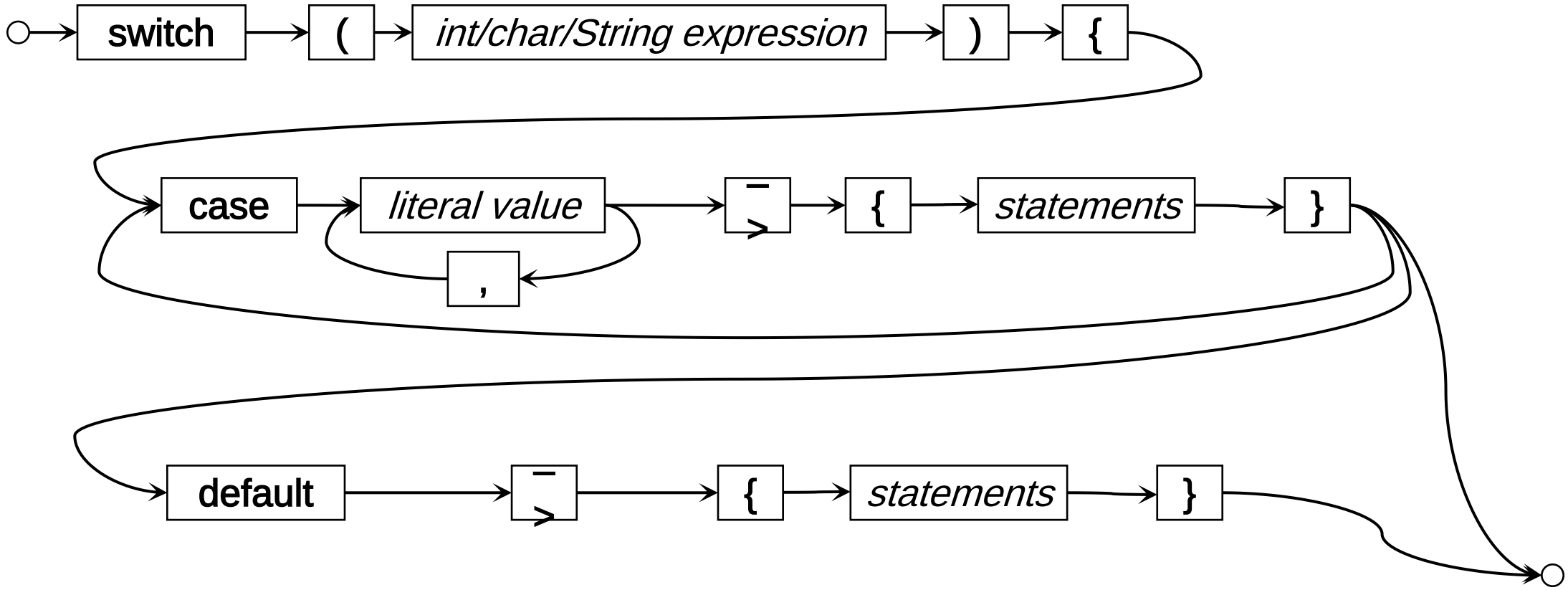
Switch

```
switch ( 〈 expression 〉 ) {
    case 〈 literal value 〉 , 〈 literal value 〉 ,…  -> { 〈statements〉 }
    case 〈 literal value 〉 , 〈 literal value 〉 ,…  -> { 〈statements〉 }
      :
    default -> { 〈statements〉 }
}
```

- The 〈 *expression* 〉 must be of type int, String, char (or a few others)
- It cannot be a double or other kinds of value (yet)
- Can't have any comparisons or logical operators.

- Be careful: always use the **->** (**:** is allowed but it does something different)

switch … case … default …



More about the Collections library

- Java provides more methods for the collections than just the “essential” operations on the collections.
  -> Read the documentation

- Makes them more useful and easier to use.

- Quick run through some of the documentation

- Note: Stack class predated the Collections library, and doesn’t have an ADT interface.

More operations on Collections

- Collection:
  - isEmpty(), size(), clear(),
  - **add(…), remove(…), contains**(...)
  - addAll(…), removeAll(…), containsAll(…), retainAll(…), removeIf(…)
  - toArray()
  - for(E item : collection){….}
- Set: collection that won't allow duplicates
  - exactly the operations for Collection
  - HashSet, TreeSet
- List:
  - additional operations based on the order:
  - add(index, …), get(index), remove(index), indexOf(…), set(index, …), sublist(…)
  - ArrayList, LinkedList

Collections Class and Arrays Class

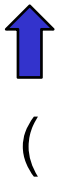
- Two Classes of useful methods to operate on
  - Collection objects
    - some methods work on any kind of collection
    - some only work on lists
    - some on work on collections of certain kinds of objects
  - Array objects

- eg, sort(…), reverse(…), max(…), fill(… …)

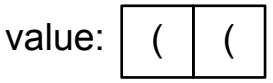
- Most methods are static, (like the Math class)
  - Collections.reverse(mylist);

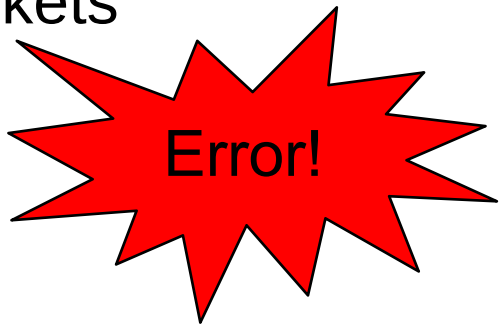
Lists vs. Stack

- Check if (([ ](])) is a correct use of brackets



(





- For each bracket in order
  - If opening bracket push(opening bracket) unto stack
  - If closing bracket pop stack
    - If returned bracket is opening version of closing bracket continue
    - else error! //including no brackets, i.e. an empty stack
- At end of string
  - if the stack is empty then the brackets are correct
  - else error!