Day 6

COMP1006/1406

Summer 2016

M. Jason Hinek
Carleton University

today's agenda

- assignments
  - Assignment 3 is due on Monday
- a quick look back
  - abstract classes and interfaces
- casting objects
- abstract data types

last time...

- abstract classes
- interfaces
- quiz 3

casting objects...

```
class Parent{
  void foo(){...}
}
```

```
class Child extends Parent{
  void bar(){...}
}
```

```
Parent pp = new Parent();
Parent pc = new Child();
Child  cc = new Child();

pp.foo();
pc.foo();
cc.foo();

pp.bar();
pc.bar();
cc.bar();
```

casting objects...

```
class Parent{
  void foo(){...}
}
```

```
class Child extends Parent{
  void bar(){...}
}
```

```
Parent pp = new Parent();
Parent pc = new Child();
Child  cc = new Child();

pp.foo();  ✓
pc.foo();  ✓
cc.foo();  ✓

pp.bar();  ✗
pc.bar();  ✗
cc.bar();  ✓
```

casting objects...

```
class Parent{
  void foo(){...}
}
```

```
class Child extends Parent{
  void bar(){...}
}
```

```
Parent pp = new Parent();
Parent pc = new Child();
Child  cc = new Child();

pp.foo();  ✓
pc.foo();  ✓
cc.foo();  ✓

// pp.bar();  ✗✗✗
((Child)pc).bar();  ✓
cc.bar();  ✓
```

```
we can cast the Parent back to a Child
ONLY because it was created (new)
as a Child
```

We can cast **down** the hierarchy chain if the object really is what we want to cast it to.

casting objects...

casting **down** the hierarchy chain will often violate good OOP design/practice

```
class Parent{
  void foo(){...}
}
```

```
class Child extends Parent{
  void bar(){...}
}
```

```
Parent[] pp = new Parent[10];
for(int i=0; i<10; i+=1){
   if(Math.random() < 0.5){ pp[i] = new Parent(); }
   else{ pp[i] = new Child(); }
}

for(Parent p : pp){
   ((Child)p).bar();          // this will not work
}
```

casting objects...

casting **down** the hierarchy chain will often violate good OOP design/practice

```
class Parent{
  void foo(){...}
}
```

```
class Child extends Parent{
  void bar(){...}
}
```

```
Parent[] pp = new Parent[10];
for(int i=0; i<10; i+=1){
   if(Math.random() < 0.5){ pp[i] = new Parent(); }
   else{ pp[i] = new Child(); }
}

for(Parent p : pp){
   if(p instanceof Child)
      ((Child)p).bar();       // this will work
}
```

now let’s look at

date types/**abstract data types**/data structures

data types

primitive data types

- byte, short, int, long (integer types)
- float, double (approximate real number types)
- char, boolean
- variables store the actual data

reference data types

- classes (abstract and concrete)
- arrays
- interfaces
- enums
- variables store **references** to where the actual data is

data structures

What is a **data structure**?

a **data structure** is a **systematic approach** to **storing** and **accessing data** so that it can be **used efficiently** for a **specific purpose**

in Java, data structures are **classes** (or a primitive data type)

abstract data types

an **abstract data type** or **ADT** is **data** and **operations on that data** that are precisely specified **independent of any implementation**

the operations may or may not have efficiency guarantees

an **ADT** is a **mathematical construct**. We simulate them as **API**s or **interfaces**.

an **ADT** is NOT

- an implementation of storing data and operations on that data in any programming language
- a specific way to store data
- specific ways (algorithms) to act on the data

For example, the integers from a math class is an abstract data type. In Java, we use `int`s or `Integer`s.

abstract data types

a **data structure** is the **implementation** of an **abstract data type**

real world ⟶ ADT ⟶ data structure (class)

for us, **abstract data type** and **interface** (contract) will mean the same thing

- **ADT/interface**
  - tells us **what** can be done with the data
  - API, abstract class, java interface
  - should not reveal how the data is stored
- **data structure/implementation**
  - is **how** those things are done
  - concrete class
  - the actual code

COMP2402 will focus on ADTs and data types

abstract data types

good object-oriented design will **separate** the interface and the implementation

Why?

some fundamental ADTs

the **list** ADT

some fundamental ADTs

the **list** ADT

- ordered collection of data $x_0, x_1, \ldots, x_{n-1}$

some fundamental ADTs

the **list** ADT

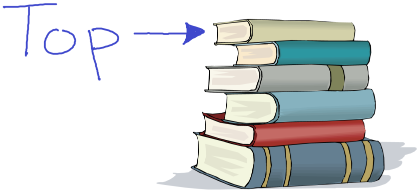
- ordered collection of data $x_0, x_1, \ldots, x_{n-1}$
- look at what is in the list
- size of the list
- add to the list
- remove from the list
- ask if the list is empty
- create a new list

some fundamental ADTs

the **stack** ADT

some fundamental ADTs

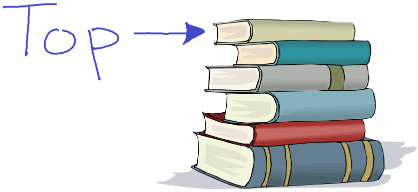
the **stack** ADT



- ordered collection of data (`top` is the last item added)

some fundamental ADTs

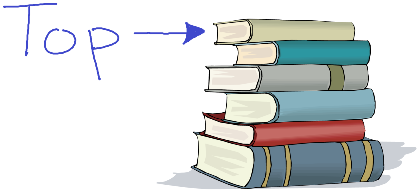
the **stack** ADT



- ordered collection of data (`top` is the last item added)
- push (add an element to the top)
- pop (remove top element)
- peek (look at the top)
- isEmpty (is the stack empty)

some fundamental ADTs

the **stack** ADT



- ordered collection of data (top is the last item added)
- push (add an element to the top)
- pop (remove top element) **Last In First Out (LIFO)**
- peek (look at the top)
- isEmpty (is the stack empty)

some fundamental ADTs

the **queue** ADT

some fundamental ADTs

the **queue** ADT

- ordered collection of data (`front` and `back`)

some fundamental ADTs

the **queue** ADT

- ordered collection of data (`front` and `back`)
- enqueue (add an element to the back)
- dequeue (remove element from the front)
- peek (look at the front)
- isEmpty (is the queue empty)

some fundamental ADTs

the **queue** ADT

- ordered collection of data (`front` and `back`)
- enqueue (add an element to the back)
- dequeue (remove element from the front) **First In First Out (FIFO)**
- peek (look at the front)
- isEmpty (is the queue empty)

some fundamental ADTs

stack and **queue**

- both are restricted lists
- stack removes items according to LIFO principle
- queue removes items according to FIFO principle

some fundamental ADTs

stack and **queue**

- both are restricted lists
- stack removes items according to LIFO principle
- queue removes items according to FIFO principle

Why do we need (or want) restricted lists?

some fundamental ADTs

stack and **queue**

- both are restricted lists
- stack removes items according to LIFO principle
- queue removes items according to FIFO principle

Why do we need (or want) restricted lists?

- forces the LIFO or FIFO principal
- might be more efficient

some fundamental ADTs

stack and **queue**

- both are restricted lists
- stack removes items according to LIFO principle
- queue removes items according to FIFO principle

Why do we need (or want) restricted lists?

- forces the LIFO or FIFO principal
  - prevents intended misuse (cheating)
  - prevents unintended misuse (mistakes)
- might be more efficient

some fundamental ADTs

stack and **queue**

- both are restricted lists
- stack removes items according to LIFO principle
- queue removes items according to FIFO principle

Why do we need (or want) restricted lists?

- forces the LIFO or FIFO principal
  - prevents intended misuse (cheating)
  - prevents unintended misuse (mistakes)
- might be more efficient
  - array vs linked lists (we'll look at this soon)

some fundamental ADTs

priority queue

- a queue in which removal is based on a priority (regardless of when added)
- highest/lowest priority item removed first
- emergency room in a hospital uses a priority queue

deque

- a double-ended queue
- allows arbitrary adding/removing from front and back

set

- an unordered collection of data
- add, remove, isMember, size, etc

some fundamental ADTs

the **map (or dictionary)** ADT

some fundamental ADTs

the **map (or dictionary)** ADT

- a collection of `(key,value)` pairs
  - keys are unique

some fundamental ADTs

the **map (or dictionary)** ADT

- a collection of `(key,value)` pairs
  - keys are unique
- mutable keys are dangerous
- some maps have order and some do not

abstract data types

why use ADTs?

abstract data types

why use ADTs?

good ADTs

- provide a model of real things (abstraction)
- capture the essence of the real things
  - the data
- capture the fundamental behaviour/operation of the real things
  - the operations on the data

abstract data types

why use ADTs?

good ADTs

- provide a model of real things (abstraction)
- capture the essence of the real things
  - the data
- capture the fundamental behaviour/operation of the real things
  - the operations on the data
- will lead to the interface/contract for the actual data type

abstract data types

why use ADTs?

good ADTs

- provide a model of real things (abstraction)
- capture the essence of the real things
  - the data
- capture the fundamental behaviour/operation of the real things
  - the operations on the data
- will lead to the interface/contract for the actual data type

when creating data types from ADTs

- interface/contract is mostly defined
  - abstract class or Java interface
- encapsulation comes for free
  - changing data representation does not affect users of data type
  - changing implementations does not affect users of the data type

abstract data types

all the basic container ADTs are implemented in Java
(some have multiple implementations)

- list ⟶ `ArrayList<T>`, `LinkedList<T>`
- priority queue ⟶ `PriorityQueue<T>`
- set ⟶ `HashSet<T>`, `TreeSet<T>`
- dictionary ⟶ `Dictionary<K,V>`

abstract data types

let's implement a **string list** ADT

$$x_0, x_1, \ldots, x_{n-1}$$

data

- an ordered collection of strings

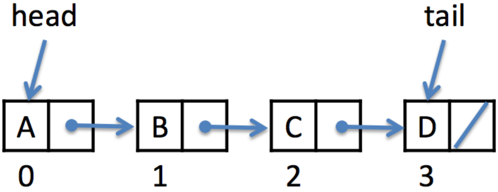
operations

- size() - returns the number of strings in the list n
- get(i) - returns string at given position x_i
- set(i, s) - replaces element at given position with given data $x_i = s$
- add(i, s) - adds a string at a given position in the list
  - make a hole by **shifting** elements and insert in the **hole**
- remove(i) - removes and returns the element in the given position
  - remove the item and then close the resulting **gap**

linked lists

a **linked list** is a sequence of **nodes**

a **node** stores **data** and a **reference** to the **next node** in the sequence (or indicates that that there is no next node)



the **head** of the linked list is the first node and the **tail** is the last

note: the diagram represents a linked list with four nodes.

some review slides

(in progress)

abstract...

- **abstract methods**
  - a method declared without a definition
  - `public abstract int foo(String[] in);`
  - forces the class to be abstract as well
  - cannot be `final`
- **abstract classes**
  - cannot be instantiated
  - may or may not contain abstract methods
  - are valid reference types and can be subclassed
  - cannot be `final`
- **concrete classes**
  - all methods (declared or inherited) must be defined
  - can be instantiated (all objects other than arrays are instantiations of concrete classes)
  - is a valid data reference type

final...

- **final attributes**
  - value cannot be changed once it is defined
  - must be defined in constructor or initialization block
  - primitive data types, strings and immutable data types are **constants**
- **final methods**
  - cannot be overridden
  - cannot be `abstract`
- **final classes**
  - cannot be extended
  - cannot be `abstract`

access modifiers...

modifier	class	package	subclass	world
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	✗
none (default)	✓	✓	✗	✗
`private`	✓	✗	✗	✗

- everything is accessible from within the class
- a class in the same package has access to everything except private members
- a subclass has access to public and protected members
- everyone else only has access to public members

arrays...

an **array** is a container that store a collection of items of the same type

```
int[] intArray;                 // variable declaration

intArray = new intArray[12];    // allocation of memory in heap for array

intArray[0] = 13;               //
...                             // population of the array with data
intArray[11] = 163;             //
```

When you declare an array variable you can also initialize it using `{...}`.
This only works when you declare the variable.

```
/* array declaration, allocation and initialization */
int[] intArray = {1,3,5,7,9};

/* all of these are equivalent */
String[] words = {"cat", "dog", "eel"};
String[] words = new String[]{"cat", "dog", "eel"};
String[] words = new String[3]{"cat", "dog", "eel"};
```

arrays

creation and initialization

an **array** is a container that store a collection of items of the same type

```
int[] intArray;                 // variable declaration

intArray = new intArray[12];    // allocation of memory in heap for array

intArray[0] = 13;               //
...                             // population of the array with data
intArray[11] = 163;             //
```

When you declare an array variable you can also initialize it using `{...}`. This only works when you declare the variable.

```
/* array declaration, allocation and initialization */
int[] intArray = {1,3,5,7,9};

/* all of these are equivalent */
String[] words = {"cat", "dog", "eel"};
String[] words = new String[]{"cat", "dog", "eel"};
String[] words = new String[3]{"cat", "dog", "eel"};
```

partially filled array

sometimes it is useful to have an array that is not completely filled

```
/* allocate memory to store 10 ints */
Integer[] numbers = new Integer[10];

/* partially fill the array */
numbers[0] = 1;
numbers[1] = 2;
numbers[2] = 4;
numbers[3] = 8;
```

when we use a partially filled array there are two things we need to keep track of when using the array

- the **capacity** of the array. This tells us how many things the array can possibly store. The `length` attribute of an array tells us this.
- the **size** of the array. This is the number of things (data) stored in the array. We will need to have a variable (attribute) to manually keep track of this.

partially filled array

sometimes it is useful to have an array that is not completely filled

```
/* allocate memory to store 10 ints */
Integer[] numbers = new Integer[10];
int       size    = 0;    // number of ints in the array

/* partially fill the array */
numbers[0] = 1;
numbers[1] = 2;
numbers[2] = 4;
numbers[3] = 8;
size = 4;    // we need to update this as we add/remove from the array
```

there are several ways that we can keep our data in the array