CS 221 Review

Mason Vail
Every class - except the Object class - directly inherits from one parent class. Object is the only class with no parent.

If a class does not declare a parent using the ‘extends’ keyword in its class header, it implicitly extends Object.

A child class inherits all variables, constants, and non-constructor methods from its parent and, by extension, any ancestors of the parent. Object is at the root of every inheritance hierarchy.

Visibility of inherited elements does not restrict whether they are inherited, but it does affect whether the child can access them directly, by name.
Inheritance (2)

Anything declared ‘private’ will not be directly accessible in a child class, even though it is inherited. Anything protected or public will be accessible by name in a child class. (Though it should be avoided, elements with default visibility - no visibility modifier - are accessible if the child is in the same package as the ancestor class where it was declared.)

Constructors are not inherited. Constructors are specific to the class in which they are declared. A child class can (and should) initialize inherited variables by calling an appropriate parent constructor via the ‘super’ keyword as the first action of its own constructor(s).
Inheritance (3)

Code that is shared among multiple related classes can (and probably should) be placed in a common parent class. This avoids code duplication, reduces maintenance, and increases consistency in classes that would be expected to share functionality.

Inherited methods can be customized or replaced through ‘overriding’ - writing a new version of the method with the exact same signature as the inherited method. The original method can be accessed via the ‘super’ keyword.

Overriding inherited variables creates ‘shadow variables’ and should be avoided.
Polymorphism (1)

Object references can point to any object compatible with the reference. The type of an object reference can be a class name, an abstract class name, an interface name, or an enumerated type name. The object assigned to that reference can only be created from a concrete, instantiable class, however.

```csharp
ReferenceType ref = new AnyCompatibleClass();
```

As long as the class being instantiated is a descendent, directly or indirectly, of the reference type class or interface, the assignment is valid. If the class is not a descendent of the reference type class or interface, the assignment is invalid.
Polymorphism (2)

‘Late binding’:

At run-time, when the dot operator is encountered, the Java Runtime Environment (JRE) first checks to see if the method or variable that follows is known to the reference type.

- If it is not known to the reference type, the call is invalid and an Exception is thrown.
- If it is known to the reference type, the JRE follows the reference to the actual object at the other end and accesses the version of the method or variable found in that object.
Polymorphism (3)

The reference type restricts what methods or variables can be accessed with the reference, but the referenced object determines what happens when the method or variable is accessed.

Objects of many different, but compatible, classes could be stored in the same ancestor reference, but the results of calling a shared method could be very different if each child overrides the inherited method differently, or implements an interface method differently.
Exceptions (1)

Exceptions are classes/objects that can be ‘thrown’ and ‘caught’ in response to (usually negative) events that should cause the program to break out of its standard flow of control until handled.

An unhandled Exception will cause a program to exit with a stack trace message printed to the Standard Error Stream (stderr) - usually mapped to the console.

A handled Exception will return the program to its standard flow of control following the code where the Exception was handled.
Exceptions (2)

Three choices for dealing with Exceptions:

1. Don’t catch / handle it. Some Exceptions \textit{should} result in the program exiting with a stack trace.
2. Handle it where it occurs. Wrap the dangerous code, and any code that should only execute if no Exception occurs, in a ‘try’ block, followed by one or more ‘catch’ blocks - a.k.a. ‘handlers’. The type of each handler reference is polymorphic, like any other reference, so the first compatible handler will catch an Exception.
3. Propagate the Exception through the call stack until it is caught in another method’s handler.
Exceptions (3)

You can create and throw Exceptions in your code. A custom Exception class needs to extend Exception (a ‘checked’ Exception that must be acknowledged) or RuntimeException (an ‘unchecked’ Exception that can be ignored in code, but is expected to crash the program if it occurs).

An Exception object is created via its constructor, like other objects, but is thrown with the keyword ‘throw’:

```
throw new MyException("Spiders! Get them off!");
```
‘Efficiency’ of an algorithm can refer to the amount of computation it requires, the amount of memory it requires, or both. Most often, we are concerned with the computational effort - roughly equivalent to the number of statements executed for a problem of size $n$.

The growth function of an algorithm exactly describes the number of statements, but is more specific than is typically necessary. As $n$ becomes large, the dominant factor of the growth function will (wait for it) dominate and lesser factors will become less significant. Regardless of coefficients, all growth functions with the same dominant factor will be characterized by the dominant factor and can be categorized together. We call this the ‘order’ of the algorithm and write it in Big-O notation.
Analysis of Algorithms (2)

Some common orders, from most efficient to least efficient:

- $O(1)$ - constant - the number of statements is independent of $n$
- $O(\log n)$ - logarithmic - to add one more statement, $n$ must double in size
- $O(n)$ - linear - the number of statements directly scales with $n$ - typical for a single loop or non-branching recursion
- $O(n \log n)$ - log-linear - linear times logarithmic - slightly worse than linear, but nowhere near as bad a quadratic
- $O(n^2)$ - quadratic - typical for a nested loop or loop containing a $O(n)$ method call
- $O(2^n)$ - exponential - each increase in $n$ doubles the number of statements
Analysis of Algorithms (3)

Multiply the order of each level of nesting (or method calls) to get the total order.

A sequence of non-nested loops is added, not multiplied. 100 consecutive $O(n)$ loops is simply $O(100n) \rightarrow O(n)$.

When looking at a loop, ask yourself if the number of iterations is affected as $n$ changes, making it likely the loop is $O(n)$. It may be necessary to think in terms of the average number of loop iterations, when conditions are variable.

Remember, conditional statements inside loops are not nested loops.

When in doubt, assume a ‘worst case scenario’ where every condition check results in the most work.
Abstract Data Types (ADTs) are the ‘mental models’ of an object that organizes and manages data.

ADTs are defined in Java Interfaces - collections of methods with clear documentation of what is expected from a class implementing the interface.

The user of a particular implementation of an ADT should not need to know anything about the inner workings of the class to successfully use the object. Knowledge of the mental model and method descriptions should be sufficient.

There may be many valid implementations of an ADT. For example, the IndexedList ADT can be implemented as an ArrayList, SingleLinkedList, or DoubleLinkedList.
Well-written code should use polymorphic references to the ADT interface, rather than hard-code a particular implementation throughout the code.

This way, swapping an alternative implementation is as simple as changing the few lines of code where a constructor call instantiates the specific class being used.

Using ADT references also helps prevent inadvertently relying on non-standard methods that might have been provided by a particular implementation of the ADT.

Once discussion includes how a data type is implemented (e.g. array-based vs linked nodes) you are not talking about an ADT, any more. IndexedList is an ADT. ArrayList is not.
Stacks

Also known as ‘Last in, first out’ or ‘LIFO’.

Mental Model for Stack:
- Vertical linear data structure.
- Elements can only be accessed from the top.
- Methods: void push(E), E pop(), E peek(), int size(), boolean isEmpty()

Good for situations where you are checking for symmetries (like matching parentheses), postfix calculations, and remembering a history of events (like the call stack).

Can be implemented efficiently with arrays or linked nodes, as long as the ‘top’ of the stack is chosen wisely.
Queues

Also known as ‘First in, first out’ or ‘FIFO’.

Mental Model for Queue:
Linear data structure.
Elements are only added at the rear and removed from the front, so they are processed in the order in which they were added. (Like standing in line.)
Methods: void add(E)/offer(E)/enqueue(E), E remove()/poll()/dequeue(), E element()/peek()/first(), int size(), boolean isEmpty()

Used when elements should be processed in the order they are received (e.g. a print queue).
Lists

A more ‘general-purpose’ linear data structure.

3 basic categories:

1. Ordered - the list, itself, decides where an added element is placed, to maintain the order of the list
2. Unordered - no inherent order, so elements can be added to the front or rear of the list, or can be added after a target element already in the list
3. Indexed - no inherent order, but elements can be accessed by their index positions in the list

Unordered and Indexed list concepts/methods do not conflict, so these two sets of functionality are often combined
Iterators (1)

Iterator is another abstract type (and a well-known design pattern) for accessing elements of other ADTs in a standardized, reliable way.

Iterators promise to deliver all elements of a collection, exactly once, and to ‘fail fast’ if any change occurs to the collection that the Iterator, itself, did not perform.

Iterator methods:

- boolean hasNext() - returns true if there is at least one more element to return
- E next() - returns the next element and moves the Iterator forward
- void remove() - removes the last returned element
Iterators (2)

Mental model of Iterator:

The Iterator waits between elements of the collection. hasNext() looks ahead to see if there is another element after the Iterator. next() advances the Iterator past the next element and returns its value. remove() removes the last element the Iterator returned from a call to next().
Iterators (3)

A basic Iterator is only expected to walk through a collection once, and the only modification it can make along the way is via the optional remove().

Double-linked Lists support a more powerful and flexible ListIterator. In addition to basic Iterator functionality, a ListIterator can move backward, as well as forward, and can add and replace elements along the way.

ListIterator’s remove() and set() methods affect the last returned element, whether that element was returned by next() or previous().

ListIterator’s add() method inserts a new element in front of the current position. A call to next() would not return the newly added element, but a call to previous() would.
Testing (1)

Testing reduces the risk that an implementation of an ADT (or any other code, for that matter) will not operate as expected under all circumstances.

Test-driven development - writing tests first, so all production code is tested from the very beginning - greatly reduces debugging time, in addition to providing assurance that the code is correct. Though somewhat counterintuitive, TDD reduces overall development time.

‘Black-box’ testing works with ADT methods, only, and confirms that an implementation provides the described ADT functionality and fits the model’s expectations without any consideration of how that implementation works internally.
Exhaustive testing is infeasible, so careful planning of tests is needed to minimize risk.

‘Boundary cases’ are high-risk cases that should be exhaustively tested, because they are where problems are most likely to occur. For example, with lists, boundary cases include actions such as adding an element to an empty list, removing the only element from a list, and adding or removing the first or last element.

‘Equivalence cases’ represent a range of similar cases. For example, with lists, modifications to an element at any position within a multi-element list (not at the front or rear) are likely equivalent to the same modifications to an element at another position within the list. Carefully choosing equivalence cases can reduce the total number of tests, while still providing confidence that the code is correct.
Implementing Linear Data Structures: Arrays (1)

Arrays are already linear and can store any kind of reference, which makes them a good starting point for linear collections. Direct access to indexed array positions makes arrays particularly fast for collection operations that look up elements by indexed position.

Regardless of the data structure, if the current array runs out of capacity, a larger array will have to be created, but this is an infrequent cost that is averaged out over all add operations.

The major issue with array-based data structures is the need to shift elements prior to adding a new element or after removing an element. Shifting is a O(n) process.
Implementing Linear Data Structures: Arrays (2)

Stack: If the end of the array is used as the top, push() and pop() operations are O(1), because no shifting is required.

Queue: Regardless of which end is assigned to be the front and rear, either the add or remove operation will be O(n), due to the need to shift elements, unless a ‘circular array’ implementation is used, in which case all operations are O(1).

List: Direct access by index is the strength of an array-based list. Add and remove operations at the rear of the list are also O(1). However, the need to shift causes all other add and remove operations to be O(n).
Implementing Linear Data Structures: Linked Nodes (1)

A linked chain of node objects, each of which holds a reference to a list element, avoids the need to shift elements and provides a flexible foundation for all linear data structures. Indexed operations, however, require $O(n)$ traversal of the chain.

The most basic linear node has one reference to the ‘next’ node in the chain. Structures built from these nodes are ‘single-linked’. Data structures must keep a reference to the first, or ‘head’ node, but it is common to also keep a ‘tail’ reference and the size of the collection to make some operations more efficient.

Adding a second reference to the ‘previous’ node in the chain allows navigating in reverse. Structures using these nodes are ‘double-linked’. In addition to increased flexibility, some operations are made more efficient by this addition.
Implementing Linear Data Structures: Linked Nodes (2)

Stack: Adding and removing from the head end of any linked chain is a $O(1)$ operation, making it the appropriate ‘top’ for a Stack. Either end of a double-linked chain would serve as the ‘top’, but all functionality is efficiently satisfied by a single-linked structure.

Queue: Adding at either end of a linked chain is $O(1)$, but removing from the tail of a single-linked structure is $O(n)$. Therefore, a Queue should use the tail end as its rear and the head end as its front. Either end works for a double-linked structure, but all functionality is efficiently satisfied by a single-linked structure.

Though a double-linked structure certainly works for Stacks and Queues, they have no need for the flexibility (or increased complexity and memory footprint) of a double-linked structure.
Lists:
Efficient O(1) insertion and removal of elements is the strength of linked structures. However, for single-linked lists, the node *ahead* of the insertion point or node to be deleted must be known. If it is not, a O(n) traversal to locate it is required. Because double-linked nodes can look back as well as forward, this is not a problem.

Linked structures also have no inherent indexing, requiring O(n) traversal for all indexed methods. For this reason, use of Iterators / ListIterators is much more important for navigating linked lists than for array-based lists to avoid repeated traversals.
Recursion (1)

Recursion is the process of simplifying a problem in a stepwise fashion until a simple base case is reached, then reassembling the final solution.

The General Recursive Algorithm:

- If the current problem can be solved (the ‘base case’):
  - Solve it.
- Else:
  - Recursively apply the algorithm to one or more simplified versions of the problem.
  - Combine the solutions to the simplified problems to produce the solution to the original problem.
Recursion (2)

Every recursive solution has a corresponding iterative solution.

Recursive solutions are frequently less efficient than iterative solutions, even when the order is the same, due to the overhead of multiple method invocations on the call stack.

However, for some problems, the recursive solution is easier to understand and more elegant to express. Clarity may be worth the performance sacrifice.
Searching Lists

‘Searching’ is the process of locating and retrieving an element in a collection.

Recognizing a match does not require knowing *everything* about the target object (or why are you searching?) but being able to recognize ‘equivalence’ - e.g. looking up contact information by the contact’s name.

The equals() method, assuming it has been overridden to actually compare object properties other than their locations, can be used for this purpose, but a more trustworthy mechanism is the compareTo() method from the Comparable interface, or the compare() method from a Comparator.
Comparable

The *Comparable* interface’s `compareTo(other)` method returns an integer giving the relative order of the target object and the other object. A negative return value indicates the target object comes first. A positive value indicates the other object comes first. Zero indicates they are equivalent.

```
int result = target.compareTo(other);
```

Classes that implement *Comparable* define the ‘natural order’ of objects of their type, via `compareTo()`. Because only one `compareTo()` can be defined in the class, it should be the most common, or natural, ordering.

Though it cannot be enforced, it is expected that classes implementing *Comparable* will override `equals()` to match the equivalence conditions of `compareTo()`. 
Classes that implement the `Comparator` interface allow comparison of objects that did not implement `Comparable` or comparison by different criteria than those defined in the objects’ `compareTo()`.

`Comparator` requires one method, `compare(obj1, obj2)`, that returns integer values in the same way as `compareTo()`. A negative return value indicates `obj1` comes first. A positive value indicates `obj2` comes first. Zero indicates they are equivalent.

```java
Comparator c = new MyComparator();
int result = c.compare(obj1, obj2);
```
Linear Search

If a list is not in a sorted order, the only certain way to find a target object (or determine that it is not present) is to look at each element in the list until the target is identified or all elements have been examined.

On average, half the elements in the list will be examined to locate a target in the collection, making the order of Linear Search - wait for it - linear. (The ‘linear’ in ‘linear search’ refers to the order of the algorithm, not the organization of the data structure.)

The advantage of linear search is that it is sure to work regardless of the ordering of elements. The downside is that it cannot take advantage of elements that are in a sorted order.
Binary Search

If elements are in a sorted order, we can leverage this organization to greatly improve our search speed by eliminating half of the remaining elements with every comparison. (The ‘binary’ in ‘binary search’ refers to the repeated halving of the problem size according to the target being less than or greater than the element being examined.)

Reducing the problem size by half with each comparison yields a $O(\log n)$ runtime.

The advantage of binary search is its efficiency. The disadvantage is that it only works if the data is already sorted.
Sorting Lists (1)

There are many algorithms for sorting linear data structures, but general-purpose algorithms that make use of *Comparable* or *Comparator* functionality fall into one of two categories:

- **Sequential Sorts - a.k.a. quadratic sorts**
  - characterized by comparing all elements to all other elements in a nested loop
  - $O(n^2)$
  - Sequential Sort, Insertion Sort, Bubble Sort

- **Logarithmic Sorts**
  - characterized by recursively dividing elements into smaller sets to minimize comparisons
  - $O(n \log n)$ - actually ‘log-linear’, not ‘logarithmic’
  - Quick Sort, Merge Sort
Sorting Lists (2)

Despite their poor scalability, the Sequential Sorts have the advantages of simplicity and low overhead, as long as the problem size is very small. In particular, Insertion Sort, which takes advantage of the growing sorted portion of the list as the algorithm progresses, is often used for small sorting tasks.

Logarithmic Sorts more than make up for their overhead and increased complexity as problem sizes increase.

Merge Sort is theoretically the most efficient of the Logarithmic Sorts, but its overhead causes it to lag behind Quick Sort on average in actual practice. Quick Sort has a worst case efficiency of $O(n^2)$, but on average is the fastest of the general-purpose sorting algorithms. Choice of the pivot element is critical to efficiency.