CS61B sp24 week4

tips

1. Arrayset (due to the ask of Proj1)
    1. what is the inner logic of "for-in loop" in Java
        - the following 2 codes has the same effect
        - iterator is a specific object in Java
        - by using `Iterator<String> seer = s.iterator();` we get a new iterator object. You don't need to use the "new" keyword in this context because the creation of the Iterator is handled internally by the collection's implementation
        - `hasNext()` would return true if there are unseen items remaining, and false if all items have been processed

```
Set<String> s = new HashSet<>();
...

for (String city : s) {
    ...
}

// equals to

Iterator<String> seer = s.iterator();
while (seer.hasNext()) {
    String city = seer.next();
    // ...
}
```

2. Iterator
    1. In `public interface List<T> extends Iterable<T>`, List is a subtype of `Iterable`, so `Iterable` is the **hypernym** of List, List is the **hyponym** of `Iterable` (collection is an extension of iterable, list is an extension of collection)
    2. **What if someone calls `next` when `hasNext` returns false?**
        - it would throw a `NoSuchElementException` error
    3. **Will `hasNext` always be called before `next`?**
        - Nope. If user exactly how many iterators there are, there is no need to call hasNext before next
    4. Does the Iterator interface have next/hasNext() methods?
        - Yes, as well as `default void remove()` method.This method can only be called once per call to `121`.
    5. we need to tell java that the variables we defined is iterator like this : `private cass ArraysetIteraote implements Iterator<T>` (using implements keyword)
    6. to support the enhanced for loop, we need to make ArraySet implements the `Iterable` interface, which including `Iterator<T> iterator()`
    7. thought the set do not have order, but when we access/iterate it, it will used as the order we define/store the data
    8. Declaring that ArraySet implements Iterable, is better than creating subclasses again and again with each of them has a next and hasNext methods
3. object methods
    1. `toString()` method provide a string representation of an object (`println` would calls `x.toString()`)

2. whenever we add sth new to the origin string, we actually create a new string in Java, so `append()` method and initialise its type as a `StringBuilder` would be a better and faster method
3. the difference between `==` and `.equals()`
    - `==` means "referencing the same objects, which have the same address in the memory"
    - `.equals()` is every object has by default which inherit the same function of the `==`
    - but we could override it to make it compare the objects type and value
    - notice that when we override it , the parameter of the method must be `object` CHECK IT!
    - `instanceof` keyword is very powerful in Java, it could check if o's dynamic type is (one of the class like "dog/`guitarStrings`")
    -
4. address of current object
    - we use `this` to access our own instance variables or methods\
    - `this` is point to the object that currently run the method no matter where it placed

4. `extends` keyword
    1. eg. `public class RotatingSLList<Blorp> extends SLList<Blorp>` which means RotatingSLList is inherits from SLList
    2. but the RotatingSLList could have new methods that SLList does not have
    3. when to use "implements" and "extends"?
        - use "implements" if the hypernym is an interface and the hyponym is a class (first level subclass)
        - use "extends" in all other cases
    4. we cannot write nested class in the interface (it more like a implementation detail)
    5. `extend` inherit all the things in the parent class (and even grand parent class) like all instance and static variables (NOTICE: `private` means even the subclass cannot access the instance/method)
5. the `super` keyword
    1. super is the reference to the parent objects (go to the parent class and execute the corresponding functions)
    2. Constructors are **not** inherited. However, the rules of Java say that all constructors must start with a call to one of the super class's constructors like this

```
public VengefulSLList() {
   super();  // this line could be show implicitly
   deletedItems = new SLList<Item>();
public VengefulSLList(Item x) {
   super(x); // this line should be show explicitly
   deletedItems = new SLList<Item>();
   // In this situation, there are 2 constuctors one with argument, the other is without rhe
arugument
}
```

6. stack
    1. we could only do 2 things in the stack, push and pop
    2. we couldn't say stack is a list, which means it will inherit all the instance and methods that list have.
    3. but we could say a stack has a list, by using a `new LinkedList<>()` privately
7. Encapsulation

> **Module:** A set of methods that work together as a whole to perform some task or set of related tasks. A module is said to be **encapsulated** if its implementation is completely hidden, and it can be accessed only through a documented interface.

8. compile-time type checking

    1. compiler is only check based on the static type, if there is any chance that the method cannot run successfully, no matter what the data type the argument truly is, it will stop at that line immediately
    2. this rule also work for value assignment, and the compiler only check the return value
    3. if we absolutely sure that the variable types is a specific type that program could run successfully, we could use type casting to avoid complier stop running it
    4. cast would not change what type the variable is! it just let the complier do not stop detecting the possible other types
    5. but sometime it will crash if the variable that should not be casted
9. High Order Function
10. From my perspective, interface actually do the pointers functions in C programming that contain pointers to functions in one class
11. we must use dot to get the methods inside the object
12. eg. this Java and Python code are equivalent

```java
public interface IntUnaryFunction {
        int apply(int x); //interface with the pointer to the func
}

public class TenX implements IntUnaryFunction {
        public int apply(int x) {
                return 10 * x; // the function that truly make difference
        }
}

public class HoFDemo {
        public static int do_twice(IntUnaryFunction f, int x) {
                // using f after calling interface name
                return f.apply(f.apply(x)); // use dot
        }

        public static void main(String[] args) {
                System.out.println(do_twice(new TenX(), 2));
        }
}
```

```Python
def tenX(x):
   return 10*x

def do_twice(f, x):
   return f(f(x))

print(do_twice(tenX, 2))
```

10. Subtype Polymorphism vs Explicit Higher Order Functions
    1. In object-oriented programming, polymorphism relates to how an object can be regarded as an instance of its own class, an instance of its superclass, an instance of its superclass's superclass, and so on.
    2. In a word, it means an object has many roles in classes, it could play as an instance in subclass. its own class and superclass
    3. there is a little trick that instead of using if-else block, we could just using `return this.size - other.size` to compare the size and return which one is bigger (with condition as `returnValue < 0`)
    4. there is a typical example of difference between these two approach from cs61b textbook

```
# Explicit HoF Approach
def print_larger(x, y, compare, stringify):
    if compare(x, y):
        return stringify(x)
    return stringify(y)

# Subtype Polymorphism Approach
def print_larger(x, y):
    if x.largerThan(y):
        return x.str()
    return y.str()
```

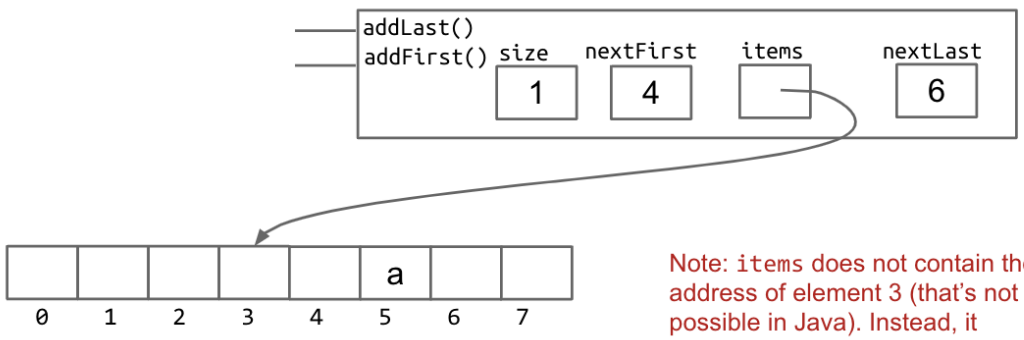
11. the Comparable **Interface** in Java
    1. it receive generic `<T>` by default, which means we could pass out data class (like dog) into it
    2. *Natural Order* means if we used `compareTo` method to all variables a specific class, then we would finally get an order based on their size/length.width etc.
    3. if we do not want to order them in their natural order, we could use high order function to rewrite it by passing a different compare function to `def print_larger(x, y, compare, stringify)`
    4. but in Java actually we could not pass the function, we could only pass **functions that inside a class**
    5. `Comparator<T>` is like a object compare machine that compare 2 same classes objects using its own `compare()` method. this point is useful in Project1
    6. the `compareTo` method is not available for primitive data types like int. It is available for certain classes that implement the Comparable interface. NOTICE: The String class in Java implements the `Comparable<String>` interface. This means that String objects have a compareTo method.
    7. instead of asking the outside world to create the object, it would be much better to give them a public static method to create a new object, thus we just need to call the class and the method and assign it like this `Comparator<Dog> nc = Dog.getNameComparator();`
    8. `Comparator` itself is not belong to the dog class !!! it is just a compare machine (The Comparator interface is part of the `java.util` package and is used to define a custom ordering for objects)

from project1

1. array deque do not need sentinel and node for most of time, since array themselves already provide an efficient way to store and access elements instead of by using index and pointers
2. The array implementation does not need to store additional pointers for each element (such as the predecessor and successor pointers in a linked list node), so the space utilisation is higher.
3. Boundary conditions in array implementations, such as whether the array is full or empty, can be easily determined using the `nextFirst and nextLast` pointers and the length of the array without the need for

sentinel nodes, eg. (from cs61b sp19-sp22 proj1 slides)

- addLast("a")



Note: `items` does not contain the address of element 3 (that's not possible in Java). Instead, it contains the address of the entire array!

You don't have to start your array at 4 or 5. I just picked these arbitrarily.