Final review

Revisiting the Midterm

Fill in the implementation given the loop invariant:

```
{{ P: 0 < n <= str.length, chars.length, lens.length }}
int runLengthEncode(String str, int n, char[] chars, int[] lens) {
    int i = 0;
    int j = 0;
    int cur = '\0';
{{ Inv: P and str[0..i-1]=chars[0]*lens[0]+…+chars[j]*lens[j] and
chars[0]!=chars[1], …, chars[j-1]!=chars[j] and (i = 0 or cur = str[i-1]) }}
    while (i != n) {
        if (str.charAt(i) == cur) {
            lens[j] = lens[j] + 1;
        }
        else {
            j = j + 1;
            cur = str.charAt(i);
            chars[j] = cur;
            lens[j] = 1;
        }
        i = i + 1;
    }
}
```

Practice Final – Reasoning

```
/**
* Returns the indexes of the subarray with maximum sum.
* @param vals Array of values
* @returns {a, b} such that sum of vals[a..b] is the maximum of
* vals[i..j] over all choices of i and j
*/
public static int[] maxSubarraySum(int[] vals) {
    int[] maxSum = int new int[vals.length+1];
    int[] maxStart = int new int[vals.length+1];
    int index = 0;
    int maxIndex = ______________;

    // Inv: parts (1-3) below:
    // (1) for j=0..index, maxSum[j] =
    // maximum sum of vals[i..j-1] over all choices of i
    // (2) for j=0..index, maxStart[j] satisfies
    // maxSum[j] = sum of vals[maxStart[j]..j-1]
    // (3) maxSum[maxIndex] is the maximum of maxSum[0..index]
    while (_________________________) {
```

Practice Final – Reasoning cont.

```
int[] maxSum = int new int[vals.length+1];
int[] maxStart = int new int[vals.length+1];
int index = 0;
int maxIndex = _______________;

// Inv: parts (1-3) below:
// (1) for j=0..index, maxSum[j] =
// maximum sum of vals[i..j-1] over all choices of i
// (2) for j=0..index, maxStart[j] satisfies
// maxSum[j] = sum of vals[maxStart[j]..j-1]
// (3) maxSum[maxIndex] is the maximum of maxSum[0..index]
```

What should we set `maxIndex` to so that Inv is initially true?

0

Practice Final - Reasoning cont.

```
// Inv: parts (1-3) below:
// (1) for j=0..index, maxSum[j] =
// maximum sum of vals[i..j-1] over all choices of i
// (2) for j=0..index, maxStart[j] satisfies
// maxSum[j] = sum of vals[maxStart[j]..j-1]
// (3) maxSum[maxIndex] is the maximum of maxSum[0..index]
while (_________________________) {
```

How do we choose the loop condition so that we end with `maxSum[maxIndex]` being the maximum subarray sum of the entire array?

```
index+1 <= vals.length
```

Practice Final – Reasoning cont.

```
/**
* Returns the indexes of the subarray with maximum sum.
* @param vals Array of values
* @returns {a, b} such that sum of vals[a..b] is the maximum of
* vals[i..j] over all choices of i and j
*/
public static int[] maxSubarraySum(int[] vals) { ...
// Inv: parts (1-3) below:
// (1) for j=0..index, maxSum[j] =
// maximum sum of vals[i..j-1] over all choices of i
// (2) for j=0..index, maxStart[j] satisfies
// maxSum[j] = sum of vals[maxStart[j]..j-1]
// (3) maxSum[maxIndex] is the maximum of maxSum[0..index]
```

Write a return statement, to go **after the loop,** so that it satisfies the specification:

```
return new int[] { maxStart[maxIndex], maxIndex-1 };
```

Practice Final – Reasoning cont.

```
// Inv: parts (1-3) below:
// (1) for j=0..index, maxSum[j] =
// maximum sum of vals[i..j-1] over all choices of i
// (2) for j=0..index, maxStart[j] satisfies
// maxSum[j] = sum of vals[maxStart[j]..j-1]
// (3) maxSum[maxIndex] is the maximum of maxSum[0..index]
```

We will increase `index` by 1 on each iteration. What **additional** claims are made in Inv when `index` changes to `index+1`? Write them for each of the parts of the loop invariant:

```
(1) maxSum[index+1] = max(sum of vals[i..index])
(2) maxSum[index+1] = sum(vals[maxStart[index+1]..index])
(3) maxSum is the maximum of maxSum[0..index+1]
```

Practice Final – Reasoning cont.

We want to compute `maxSum[index+1] = max(sum of vals[i..index])`

Range of values for `i: 0..index+1`

When `i >= index+1,` empty subarray (vals[index+1..index] is empty), so sum is 0

For any `i <= index,` subarray is non-empty because at least includes vals[index]

This means we can write:

sum of vals[i..index] = (sum of vals[i..index-1]) + vals[index]

If we take max(sum of vals[i..index]) over all i <= index, since vals[index] is the same for all i, this is max(sum of vals[i..index-1]) + vals[index]

Write a short Java expression to compute max(sum of vals[i..index-1]) + vals[index]

```
maxSum[index] + vals[index]
```

Practice Final – Reasoning cont.

The value of max(sum of vals[i..index]) is either the sum for an

i >= index+1 or the sum for an i <= index

Write a short Java expression that determines whether the maximum is achieved by some i <= index rather than by i >= index+1

```
maxSum[index] + vals[index] > 0
```

Practice Final – Reasoning cont.

Fill in code to ensure that parts (1-2) of the invariant are still satisfied when index is incremented in the case that max(sum of vals[i..index]) is achieved by some i <= index:

```
maxSum[index+1] = maxSum[index] + vals[index];
maxStart[index+1] = maxStart[index];
```

Practice Final – Reasoning cont.

Fill in code to ensure that parts (1-2) of the invariant are still satisfied when index is incremented in the case that max(sum of vals[i..index]) is achieved by some i >= index+1:

```
maxSum[index+1] = 0;
maxStart[index+1] = index+1;
```

Practice Final – Reasoning cont.

Fill in code to ensure that part (3) of the invariant is satisfied when index is incremented. (This should work for either of the two cases considered above, and it should not require any additional loops!)

```
if (maxSum[index+1] > maxSum[maxIndex])
   maxIndex = index+1;
```

The implementation of `maxSubarraySum` is now complete! Notice how we were able to decompose the problem using only what was already given to us despite the difficulty of the implementation

Practice Final – Iterators

Recall that the interface Iterator<T> includes the following methods:

```
/** Produces a stream of objects of type T. */
public iterator Iterator<T> {
    /** Determines whether there is another object to return. */
    public boolean hasNext();

    /** Returns the next object in the stream or throws
    * NoSuchElementException if there are no more remaining. */
    public T next() throws NoSuchElementException;
}
```

Here the term “stream” means that the objects are produced one-at-a-time in order with no way to retrieve the previous elements – once they are produced they are forgotten by the iterator.

Practice Final - ADTs

You are to design an ADT, `GenrePlayListIterator`, that implements `PlayListIterator` by looking through the user's library of songs. Your ADT has the fields and RI shown below.

Fill in the obvious parts of the RI that are missing in part (1):

```
// RI: parts (1-3) below:
// (1) library != null, albumGenres != null, genresSeen != null
// (2) genresSeen contains all genres returned previously by next()
// (3) start is the smallest value in [0..library.size()] that is
// strictly larger than the first index at which each genre in
// genresSeen appears in libary
private final List<Song> library;
private final Map<String, String> albumGenres;
private final Set<String> genresSeen;
private int start;
```

Practice Final – ADTs cont.

Fill in the blank to complete the abstraction function of your ADT:

AF(this) = stream of playlists, each containing all songs with a particular genre, with one list for each genre in library not included in genresSeen, return in the order they first appear in library.

Practice Final – ADTs cont.

```
/**
* Creates an iterator that produces playlists for each album
* genre found in the given library.
* @param library List of songs in the user's library.
* @param albumGenres Maps albums to their genre
* @requires library & albumGenres are non-null
*/
public GenrePlayListIterator(
       List<Song> library, Map<String, String> albumGenres) {
    this.library = library;
    this.albumGenres = albumGenres;
    this.genresSeen = new HashSet<String>();
    this.start = 0;
}
```

Finish the Practice Final!

The practice final is intended to make you familiar with the format of the final exam and make you comfortable with similar style problems and terms you may encounter – be sure to study!

Stronger vs Weaker (one more time!)

- Requires more?

- Promises more? (stricter specifications on what the effects entail)

Stronger vs Weaker (one more time!)

- Requires more?

weaker

- Promises more? (stricter specifications on what the effects entail)

stronger

Stronger vs Weaker

```
@requires key is a key in this
@return the value associated with key
@throws NullPointerException if key is null
```

```
A.@requires key is a key in this and key != null
  @return the value associated with key
A.@return the value associated with key if key is a
      key in this, or null if key is not associated
with any value
B.@return the value associated with key
  @throws NullPointerException if key is null
  @throws NoSuchElementException if key is not a
  key this
```

Stronger vs Weaker

```
@requires key is a key in this
@return the value associated with key
@throws NullPointerException if key is null
```

A. `@requires key is a key in this and key != null`
   `@return the value associated with key` **WEAKER**

A. `@return the value associated with key if key is a key in this, or null if key is not associated with any value` **NEITHER**

B. `@return the value associated with key`
   `@throws NullPointerException if key is null`
   `@throws NoSuchElementException if key is not a key this`

STRONGER

Exceptions

- Unchecked exceptions are ignored by the compiler.
- If a method throws a checked exception or calls a method that throws a checked exception, then it must either:
  1. catch the exception
  2. declare it in `@throws`

Exceptions Examples

Should these be checked or unchecked?

- Attempt to write an invalid type into an array
  E.g., write `Double` into `Integer[]` cast to `Number[]`

- Attempt to open a file that does not exist

- Attempt to create a URL from invalidly formatted text
  E.g., “http:/foo” (only one “/”)

Exceptions Examples

Should these be checked or unchecked?

- Attempt to write an invalid type into an array
  E.g., write `Double` into `Integer[]` cast to `Number[]`

  unchecked

- Attempt to open a file that does not exist

  checked

- Attempt to create a URL from invalidly formatted text
  E.g., “http:/foo” (only one “/”)

  debatable – could see either one

Subtypes & Subclasses

- Subtypes are substitutable for supertypes
- If `Foo` is a subtype of `Bar`, `G<Foo>` is a **NOT** a subtype of `G<Bar>`
  - Aliasing resulting from this would let you add objects of type `Bar` to `G<Foo>`, which would be bad!
  - Example:

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
lo.add(new Object());
String s = ls.get(0);
```

- Subclassing is done to reuse code (extends)
  - A subclass can override methods in its superclass

Typing and Generics

- <?> is a wildcard for unknown
  - Upper bounded wildcard: type is wildcard or subclass
    - Eg: `List<? extends Shape>`
    - Illegal to write into (no calls to add!) because we can't guarantee type safety.
  - Lower bounded wildcard: type is wildcard or superclass
    - Eg: `List<? super Integer>`
    - May be safe to write into.

Subtypes & Subclasses

```
class Student extends Object { ... }
class CSEStudent extends Student { ... }
```

```
List<Student> ls;
List<? extends Student> les;
List<? super Student> lss;
List<CSEStudent> lcse;
List<? extends CSEStudent> lecse;
List<? super CSEStudent> lscse;
Student scholar;
CSEStudent hacker;
```

```
ls = lcse;
les = lscse;
lcse = lscse;
les.add(scholar);
lscse.add(scholar);
lss.add(hacker);
scholar = lscse.get(0);
hacker = lecse.get(0);
```

Subtypes & Subclasses

```
class Student extends Object { ... }
class CSEStudent extends Student { ... }
```

```
List<Student> ls;
List<? extends Student> les;
List<? super Student> lss;
List<CSEStudent> lcse;
List<? extends CSEStudent> lecse;
List<? super CSEStudent> lscse;
Student scholar;
CSEStudent hacker;
```

```
ls = lcse;          X
les = lscse;
lcse = lscse;
les.add(scholar);
lscse.add(scholar);
lss.add(hacker);
scholar = lscse.get(0);
hacker = lecse.get(0);
```

Subtypes & Subclasses

```
class Student extends Object { ... }
class CSEStudent extends Student { ... }
```

```
List<Student> ls;
List<? extends Student> les;
List<? super Student> lss;
List<CSEStudent> lcse;
List<? extends CSEStudent> lecse;
List<? super CSEStudent> lscse;
Student scholar;
CSEStudent hacker;
```

```
ls = lcse;                 X
les = lscse;               X
lcse = lscse;
les.add(scholar);
lscse.add(scholar);
lss.add(hacker);
scholar = lscse.get(0);
hacker = lecse.get(0);
```

Subtypes & Subclasses

```
class Student extends Object { ... }
class CSEStudent extends Student { ... }
```

```
List<Student> ls;
List<? extends Student> les;
List<? super Student> lss;
List<CSEStudent> lcse;
List<? extends CSEStudent> lecse;
List<? super CSEStudent> lscse;
Student scholar;
CSEStudent hacker;
```

```
ls = lcse;            X
les = lscse;          X
lcse = lscse;         X
les.add(scholar);
lscse.add(scholar);
lss.add(hacker);
scholar = lscse.get(0);
hacker = lecse.get(0);
```

Subtypes & Subclasses

```
class Student extends Object { ... }
class CSEStudent extends Student { ... }
```

```
List<Student> ls;
List<? extends Student> les;
List<? super Student> lss;
List<CSEStudent> lcse;
List<? extends CSEStudent> lecse;
List<? super CSEStudent> lscse;
Student scholar;
CSEStudent hacker;
```

ls = lcse; **X**

les = lscse; **X**

lcse = lscse; **X**

les.add(scholar); **X**

lscse.add(scholar);

lss.add(hacker);

scholar = lscse.get(0);

hacker = lecse.get(0);

Subtypes & Subclasses

```
class Student extends Object { ... }
class CSEStudent extends Student { ... }
```

```
List<Student> ls;
List<? extends Student> les;
List<? super Student> lss;
List<CSEStudent> lcse;
List<? extends CSEStudent> lecse;
List<? super CSEStudent> lscse;
Student scholar;
CSEStudent hacker;
```

```
ls = lcse;                X
les = lscse;              X
lcse = lscse;             X
les.add(scholar);         X
lscse.add(scholar);       X
lss.add(hacker);
scholar = lscse.get(0);
hacker = lecse.get(0);
```

Subtypes & Subclasses

```
class Student extends Object { ... }
class CSEStudent extends Student { ... }
```

```
List<Student> ls;
List<? extends Student> les;
List<? super Student> lss;
List<CSEStudent> lcse;
List<? extends CSEStudent> lecse;
List<? super CSEStudent> lscse;
Student scholar;
CSEStudent hacker;
```

ls = lcse; X

les = lscse; X

lcse = lscse; X

les.add(scholar); X

lscse.add(scholar); X

lss.add(hacker);

scholar = lscse.get(0);

hacker = lecse.get(0);

Subtypes & Subclasses

```
class Student extends Object { ... }
class CSEStudent extends Student { ... }
```

```
List<Student> ls;
List<? extends Student> les;
List<? super Student> lss;
List<CSEStudent> lcse;
List<? extends CSEStudent> lecse;
List<? super CSEStudent> lscse;
Student scholar;
CSEStudent hacker;
```

ls = lcse; X

les = lscse; X

lcse = lscse; X

les.add(scholar); X

lscse.add(scholar); X

lss.add(hacker); 

scholar = lscse.get(0);X

hacker = lecse.get(0);

Subtypes & Subclasses

```
class Student extends Object { ... }
class CSEStudent extends Student { ... }
```

```
List<Student> ls;
List<? extends Student> les;
List<? super Student> lss;
List<CSEStudent> lcse;
List<? extends CSEStudent> lecse;
List<? super CSEStudent> lscse;
Student scholar;
CSEStudent hacker;
```

```
ls = lcse;              X
les = lscse;            X
lcse = lscse;           X
les.add(scholar);       X
lscse.add(scholar);     X
lss.add(hacker);
scholar = lscse.get(0); X
hacker = lecse.get(0);
```

Subclasses & Overriding

```
class Foo extends Object {
     Shoe m(Shoe x, Shoe y){ ... }
}

class Bar extends Foo {...}
```

Method Declarations in Bar

- The result is method overriding
- The result is method overloading
- The result is a type-error
- None of the above

Object
↓
Foo
↓
Bar

Footwear
↓
Shoe
↓
HighHeeledShoe

- FootWear m(Shoe x, Shoe y) { ... } **type-error**
- Shoe m(Shoe q, Shoe z) { ... } **overriding**
- HighHeeledShoe m(Shoe x, Shoe y) { ... } **overriding**
- Shoe m(FootWear x, HighHeeledShoe y) { ... } **overloading**
- Shoe m(FootWear x, FootWear y) { ... } **overloading**
- Shoe m(Shoe x, Shoe y) { ... } **overriding**
- Shoe m(HighHeeledShoe x, HighHeeledShoe y) { ... } **overloading**
- Shoe m(Shoe y) { ... } **overloading**
- Shoe z(Shoe x, Shoe y) { ... } **none (new method declaration)**

Event-Driven Programs

- Sits in an event loop, waiting for events to process
  - often does so until forcibly terminated

- Two common types of event-driven programs:
  1. GUIs
  2. Web servers

- Where is the event loop in Java AWT/Swing?
  - it is created behind the scenes when you call `JFrame.setVisible(true)`

Design Patterns

- Creational patterns: get around Java constructor inflexibility
  - Sharing: singleton, interning
  - Telescoping constructor fix: builder
  - Returning a subtype: factories
- Structural patterns: translate between interfaces
  - Adapter: same functionality, different interface
  - Decorator: different functionality, same interface
  - Proxy: same functionality, same interface, restrict access
  - All of these are types of wrappers

Design Patterns

- Interpreter pattern:
  - Collects code for similar objects, spreads apart code for operations (classes for objects with operations as methods in each class)
  - Easy to add objects, hard to add methods
  - Instance of Composite pattern
- Procedural patterns:
  - Collects code for similar operations, spreads apart code for objects (classes for operations, method for each operand type)
  - Easy to add methods, hard to add objects
  - Ex: Visitor pattern

Design Patterns

Adapter, Builder, Composite, Decorator, Factory, Flyweight, Iterator, Intern, Interpreter, Model-View-Controller (MVC), Observer, Procedural, Prototype, Proxy, Singleton, Visitor, Wrapper

- What pattern would you use to…
  - add a scroll bar to an existing window object in Swing
  - We have an existing object that controls a communications channel. We would like to provide the same interface to clients but transmit and receive encrypted data over the existing channel.
  - When the user clicks the “find path” button in the Campus Maps application (hw9), the path appears on the screen.

Design Patterns

Adapter, Builder, Composite, Decorator, Factory, Flyweight, Iterator, Intern, Interpreter, Model-View-Controller (MVC), Observer, Procedural, Prototype, Proxy, Singleton, Visitor, Wrapper

- What pattern would you use to…
  - add a scroll bar to an existing window object in Swing
    - **Decorator**
  - We have an existing object that controls a communications channel. We would like to provide the same interface to clients but transmit and receive encrypted data over the existing channel.
    - **Proxy**
  - When the user clicks the “find path” button in the Campus Maps application (hw9), the path appears on the screen.
    - **MVC**
    - **Observer**