CCSI 245. Test 2 review.
Test 2: Friday, Apr 10, 2026

Since Test 1, we have covered a lot of things. They can be summarized under the following headings:

- Object-oriented design and class extension
  - UML diagrams
  - Relationships among classes: Acquaintance, aggregation, and composition
  - Design goals: Loose coupling (vs tight coupling), encapsulation, cohesion, responsibility-driven design, black box reuse (vs white box reuse).
  - Class extension: Abstract class (vs concrete class), abstract method, superclass (parent class) vs subclass (child class), inheritance, access modifiers (public, protected, private), `super`, overriding (compare with overloading).
  - Refactoring: Bad smells (redundant code, long method), extract method, pull-up method, pull-up instance variable.
- Computer memory, pointers, an dynamic allocation
  - Computer memory as organized into bytes (smallest addressable units), each containing a sequence of 8 bits; each C type has a size (number of bytes); `sizeof` operator.
  - Pointer values, types, and variables; `&` (address-of operator), `*` (dereference operator), `[]` (array dereference operator; note that `a[b]` is equivalent to `*(a+b)`), `->` (struct dereference operator; recall that `a->b` is equivalent to `(*a).b`).
  - Dynamic allocation of memory; C functions `malloc`, `calloc`, and `free`.
  - Bitwise operators: `&`, `|`, `^` (xor), `~` (negation), and `<<` (bitshift left).
- Nested classes and generics
  - The taxonomy of nested classes: Static vs non-static, member, inner, anonymous inner.
  - How to write generic methods.
  - How to use generic classes.
  - How to write generic classes.
- ADT and data structures
  - The concepts, operations, and uses of stack and queues.
  - Using arrays and linked lists to implement stacks and queues.
  - Hash tables as implementations of maps.

Kinds of problems to expect. This test is very much about assessing your ability to program. Here are the kinds of problems that will be on the test:

- A problem in which you'll be given a code example involving a collection of classes and interfaces, including classes that extend other classes; then, given a sequence of method invocations, you will be asked to determine which method implementations are called. This is to test how well you can reason through interface implementation, class extension, method overriding, and method overloading, as well as the difference between static and dynamic types. See below for an example.
- A problem in which you determine the types of various subexpressions in a C expression involving pointers, similar to the practice problem we did in class and the one you had for homework.
- A programming problem in which you write a Java class that uses a provided class, either by composition or subclassing.
- A programming problem in which you write a Java method that uses an anonymous inner class for its return value (likely either a `Comparable`, `ActionListener`, or `Iterator`).
- A programming problem in which you write a C function that uses `malloc`, `calloc`, and/or `free` to allocate and/or deallocate memory.

Practice problems.

1. Consider the following interface and classes.

```
interface I {
    void p();
    void q();
}
abstract class A implements I {
    public void p() { System.out.println("1"); }
    public abstract void r();
}
class B implements I {
    public void p() { System.out.println("2"); }
    public void q() { System.out.println("3"); }
}
class C extends A {
    public void q() { System.out.println("4"); }
    public void r() { System.out.println("5"); }
}
class D extends C {
    public void p() { System.out.println("6"); }
}
```

Consider also the following variable declarations, each variable initialized by an instantiation.

```
I i1 = new C();
I i2 = new B();
I i3 = new D();
A a = new C();
D d = new D();
```

For each of the following method invocations, determine which method implementation is called *or indicate that there will be a compiler error on that line.* Indicate by writing the number (1–6) that will be printed.

```
i1.p();
i1.q();
i1.r();
i2.p();
i2.q();
i3.p();
i3.q();
a.p();
a.q();
a.r();
d.p();
d.q();
d.r();
```

2. A *double-ended queue* or *deque* is like a stack or queue except that you can both add and remove at both the front and the back (but nowhere else). Suppose that a generic `Deque` class with the following public methods (and constructor) has already been written:

```
class Deque<E> {
  Deque();
  void addFront(<E> item);
  void addBack(<E> item);
  E getFront();
  E getBack();
  E removeFront();
  E removeBack();
```

```
  boolean isEmpty();
  boolean isFull()
}
```

Write a generic class that implements a stack—specifically, implementing the following `Stack` interface—by using the provided `Deque` class, either by composition or inheritance *and indicate which one (composition or inheritance) you are using.*

```
interface Stack<E> {
  void push(E item);
  E top();
  E pop();
  boolean isEmpty();
  boolean isFull();
}
```

3. Consider the following class that implements an `NSet`, like in the bit vector project, but with an internal boolean array, such that if the item x is contained in the conceptual set, then `internal[x]` is `true`.

```
public class BArrayNSet {
    private boolean[] internal;
    public BArrayNSet(int N) { internal = new boolean[N]; }
    public boolean conatins(int x) { return internal[x]; }
    public void add(int x) { internal[x] = true; }
    // more methods...
```

Write the following method that returns an iterator over the conceptual set; that is, successive calls to `next()` on the iterator returned should return the values contained in the set *not the booleans in the array.* For example, if $N = 15$ and the conceptual set is $\{0, 1, 3, 8, 12\}$, then the internal array is $[T, T, F, T, F, F, F, F, T, F, F, F, T, F, F]$; in that case, the iterator should return 0, 1, 3, 8, 12. Use an anonymous inner class inside the `iterator` method.

```
  // inside class BArrayNSet
  public Iterator<Integer> iterator() {
```

4. Write a C function `grow_array` that takes an array and makes a bigger array, copying the contents of the given array into the new one. Specifically, it takes an `int*` (interpreted as an `int` array) and two other `int`s `old_size` and `new_size`. Assume that the given array is of length `old_size`. The function you are to write should allocate a new array of length `new_size`, copy the contents of the old array to the new, and deallocate the old one. For example, if another part of the code has an array referred to by variable `a` and `n` is the length of that array, it can make the array appear to double in size by calling your function:

```
int *a;
int n;

/* ... */

a = grow_array(a, n * 2);
```

Here is the stub for the function you are to write:

```
int* grow_array(int *old_array, int old_size, int new_size)
  {
```