COMP 333
Spring 2026
Midterm Practice Exam #1 (Solutions)

This is representative of the kinds of topics and kind of questions you may be asked on the midterm. This practice exam, along with assignment 1 and the in-class handouts on Java and types, are intended to be comprehensive of everything on the exam. That is, I will not ask anything that's not somehow covered by those sources.

You are permitted to bring two 8.5 x 11 sheets of paper into the exam with you, containing either printed or handwritten notes. Both sides of both sheets can be used.

Virtual Dispatch - Conceptual Understanding

1.) Name one reason why someone might want to use virtual dispatch (i.e., Java's way of implementing ad-hoc polymorphism).

Non-exhaustive list of possibilities:

- Allows for the same method call to have different behaviors at runtime, which is more flexible
- Allows for abstraction over code behavior. Calling code only needs to know the signature of a method, and the runtime behavior can be dynamically changed by changing the underlying object the method is called on.
- Improves code modularity. Each distinct behavior can be isolated from each other behavior, and behaviors generally don't need to know about each other.

2.) Name one reason why someone might **not** want to use virtual dispatch (i.e., Java's way of implementing ad-hoc polymorphism).

Non-exhaustive list of possibilities:

- Code can become more bloated. Usually, multiple classes have to be introduced, and these all have their own boilerplate associated with them.
- Code's behavior can become less explicit, and potentially more difficult to reason about. For example, an if/else clearly indicates a condition to test and code to execute depending on the condition. However, a method call may *implicitly* do something similar.
- It generally has worse performance than an explicit if/else. (Beyond what I'd expect: behind the scenes, indirect jumps are often needed, and these can easily jump out of cached instructions. Moreover, it's difficult to predict where an indirect jump will go.)

Virtual Dispatch in Java

3.) Consider the following Java code:

```
public interface I1 {
  public void doThing();
}
public class C1 implements I1 {
  public void doThing() { System.out.println("c1"); }
}
public class C2 implements I1 {
  public void doThing() { System.out.println("c2"); }
}
public class Main {
  public static void makeCall(I1 value) {
    value.doThing();
  }
  public static void main(String[] args) {
    I1 t1 = new C1();
    I1 t2 = new C2();
    makeCall(t1);
    makeCall(t2);
  }
}
```

What is the output of the `main` method above?

```
c1
c2
```

4.) Consider the following code snippet:

```
public class Main {
  public static void main(String[] args) {
    Operation op1 = new AddOperation();       // line 3
    Operation op2 = new SubtractOperation(); // line 4
    int res1 = op1.doOp(5, 3);                // line 5
    int res2 = op2.doOp(5, 3);                // line 6
    System.out.println(res1); // line 7; should print 8
    System.out.pritnln(res2); // line 8; should print 2
  }
}
```

Define any interfaces and/or classes necessary to make this snippet print `8`, followed by `2`.

```
// From lines 3-4, we know that Operation must be a superclass
// of AddOperation and SubtractOperation, based on the types of
// op1 and op2.  From line 5, we know that Operation must have a
// doOp method, that it must return an int, and that it must
// take two ints.  From line 3, 5, and 7, we can infer that
// AddOperation's doOp must be adding its arguments, and
// similarly from lines 4, 6, and 8, we can infer
// SubtractOperation's doOp must be subtracting its arguments.

public interface Operation {
  public int doOp(int first, int second);
}

public class AddOperation implements Operation {
  public int doOp(int first, int second) {
    return first + second;
  }
}

public class SubtractOperation implements Operation {
  public int doOp(int first, int second) {
    return first - second;
  }
}
```

5.) Consider the following incomplete Java code and output:

```
public class Incomplete {
  public static void printResult(final Runner r, final int i) {
    boolean result = r.someMethod(i);
    System.out.println(result);
  }
  public static void main(final String[] args) {
    final IsEven even = new IsEven();
    printResult(even, 3); // prints false
    printResult(even, 4); // prints true
    final IsLessThan ltFive = new IsLessThan(5);
    printResult(ltFive, 4); // prints true
    printResult(ltFive, 6); // prints false
    final IsLessThan ltZero = new IsLessThan(0);
    printResult(ltZero, -1); // prints true
    printResult(ltZero, 1); // prints false
  }
}
```

Define any interfaces and/or classes necessary to make the output in the comments work. You should not have to modify any code here. Multiple answers are possible.

```
public interface Runner {
  public boolean someMethod(int i);
}
public class IsEven implements Runner {
  public boolean someMethod(int i) {
    return i % 2 == 0;
  }
}
public class IsLessThan implements Runner {
  public final int value;
  public IsLessThan(final int value) {
    this.value = value;
  }
  public boolean someMethod(int i) {
    return i < value;
  }
}
```

6.) Consider the following Java code, which simulates a lock which can be either locked or unlocked. The lock is an immutable data structure, so locking or unlocking returns a new lock in an appropriate state.

```
public class Lock {
    private final boolean locked;

    public Lock(final boolean locked) {
        this.locked = locked;
    }

    public Lock unlock() {
        if (locked) {
            System.out.println("lock unlocked");
            return new Lock(false);
        } else {
            System.out.println("lock already unlocked");
            return this;
        }
    }

    public Lock lock() {
        if (!locked) {
            System.out.println("lock locked");
            return new Lock(true);
        } else {
            System.out.println("lock already locked");
            return this;
        }
    }

    public boolean isLocked() {
        return locked;
    }
}
```

Refactor this code to use virtual dispatch, instead of using `if`/`else`. As a hint, you should have a base class/interface for `Lock`, and subclasses for locked and unlocked locks. `Lock` itself doesn't need a constructor, and you do not need to worry about maintaining compatibility with existing code that uses `Lock`.

---(Continued on to next page)---

```
public interface Lock {
    public Lock unlock();
    public Lock lock();
    public boolean isLocked();
}

public class UnlockedLock implements Lock {
    public Lock unlock() {
        System.out.println("lock already unlocked");
        return this;
    }

    public Lock lock() {
        System.out.println("lock locked");
        return new LockedLock();
    }

    public boolean isLocked() {
        return false;
    }
}

public class LockedLock implements Lock {
    public Lock unlock() {
        System.out.println("lock unlocked");
        return new UnlockedLock();
    }

    public Lock lock() {
        System.out.println("lock already locked");
        return this;
    }

    public boolean isLocked() {
        return true;
    }
}
```

Types

7.) The code below does not compile. Why?

```
public interface MyInterface {
  public void foo();
}

public class MyClass implements MyInterface {
  public void foo() {}
  public void bar() {}

  public static void main(String[] args) {
    MyInterface a = new MyClass();
    a.bar();
  }
}
```

The `bar` method is only available on `MyClass`, not `MyInterface`. `a` is of type `MyInterface`, so the `bar` method is not available, even though the runtime type of `a` will be `MyClass` (which does have `bar`).

8.) Java supports subtyping polymorphism. Write a Java code snippet that compiles and uses subtyping polymorphism.

```
Object obj = "foo";
```

(`"foo"` is of type `String`, and `String` is a subtype of `Object`, so values of type `String` can be assigned to variables of type `Object`.)

9.) Name one reason why someone might prefer static typing over dynamic typing.

Possible answers:

- Can help eliminate programming errors at compile time
- Can help in structuring code (types can be a guide for program development)
- Compilers and runtimes can usually optimize code better with type information

10.) Name one reason why someone might prefer dynamic typing over static typing.

Possible answers:
- No need to annotate programs with types, which are a significant amount of code
- More programs are possible, and can fundamentally represent things which cannot be represented in statically-typed code.

11.) Name one reason why someone might prefer strong typing over weak typing.

Possible answers:
- Strongly-typed languages are more predictable, especially in the presence of bugs and errors
- Strongly-typed languages tend to emphasize code correctness

12.) Name one reason why someone might prefer weak typing over strong typing.

Possible answers:
- There is usually a performance benefit. For example, there is no need for runtime array bounds checking.
- Weakly-typed languages tend to be more expressive. (Beyond what I'd expect: C lets you do more than Java in terms of low-level manipulation, but it's easy to hit undefined behavior, meaning programs lose all meaning. This can happen even when the code appears to be working correctly.)