Comp 333 Midterm

1. Name one reason why someone might want to use virtual dispatch.

   - Allows for 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.

   - 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.
     - 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.

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(final I1 value) {
    value.doThing();
  }
  public static void main(final String[] args) {
    final I1 t1 = new C1();
    final I1 t2 = new C2();
    makeCall(t1);
    makeCall(t2);
  }
}
```

What is the output of the main method?

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.println(res2);  // line 8; should print 5
  }
}
```

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

```
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;
  }
}
```

```
public class Main {
 // dynamic typing; (aka unit typing)
 // Base class / interface: Operation
 //    - Subclass: AddOperation
 //    - Subclass: SubtractOperation
 // Operation has doOp method
 //    - Takes two integers
 //    - Returns an integer
 public static void main(String[] args) {
   Operation op1 = new AddOperation();
   Operation op2 = new SubtractOperation();
   int res1 = op1.doOp(5, 3);
   int res2 = op2.doOp(5, 3);
   System.out.println(res1); // prints 8
   System.out.println(res2); // prints 2
 }
}
```

5. 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: 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. (Continued on to next page)

```
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. (Continued on to next page)

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

```
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;
  }
}
```

```
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;
  }
}
```

6. The code below does not compile. Why?

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

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

ANSWER:

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

```
// Reason #1 of not compiling: extends used instead of implements;
// implements is needed for interfaces
public class MyClass implements MyInterface {
  public void foo() {}
  public void bar() {}
  public static void main(String[] args) {
    MyInterface a = new MyClass();
    a.bar(); // Reason #2: bar is not defined on MyInterface
  }
}
```

7. Java supports sub-typing. Write a Java code snippet that compiles and uses sub-typing.
    - 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 types Object.
8. Name one reason why someone might prefer static typing over dynamic typing.
    - Helps eliminate programming errors at compile time.
    - Helps structuring code (types can be a guide for program development).
    - Compilers and runtime can usually optimize code better with type information.
9. Name one reason why someone might prefer dynamic typing over static typing.
    - 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.
10. Name one reason why someone might prefer strong typing over weak typing.
    - Strongly-type languages are more predictable, especially in the presence of bugs and errors.
    - Strongly-type languages tend to emphasize code correctness.
11. Name one reason why someone might prefer weak typing over strong typing.
    - No need for runtime array bounds check, so performance benefit.
    - Weakly-typed languages tend to be more expressive.
        - C lets you do more than Java in terms of low-level manipulation, but its easy to hit undefined behavior, meaning programs lose all meaning.
        - This can happen even when the code appears to be working correctly.

static: happens at compile time

dynamic: happens at runtime

type inference: statically typed, but less explicit

JavaScript: dynamically typed, strongly typed

Types are associated values:

- 5: int
- “foo”: String

Java: statically typed, strongly typed

Types are associated values:

- 5: int
- “foo”: String

Types are ALSO associated with variables

- int x; String y; Object z; ImmutableList list;

C: statically typed, weakly typed

- Weak typing: the type might not be correct; the type can lie

Assembly: untyped