Midterm

- The reflection classes are in two packages:
  - `java.lang`
    - `Object`
    - `Class`
  - `java.lang.reflect`
    - `Method`
    - `Field`
    - `Constructor`
    - etc
- `java.lang.Object`
  - Is the root superclass of every object in a program
  - Each base-level object keeps a reference to its class object
    - Accessed with the method `public final Class getClass()`
    - Eg. `Object myObj = new ..; Class classObj = myObj.getClass();`

Classes:

- `java.lang.Class`
  - Is the class of metalevel class *objects*
  - Has many useful reflective methods to:
    - Create new instances
    - Find methods, constructors, and fields of a class
    - Traverse the inheritance hierarchy
- Finding class objects
  - For an already instantiated base-level object, use `getClass()`
  - If you know the class name at compile time, use the class literal `.class`
    - `Class classObject = Color.class;`
  - If the class name is represented as a String (usually at runtime) use the method:
    - `public static Class forName(String className);`
    - If not already loaded, dynamically loads the class from bytecode in the .class file
    - If the class is in a named package, use the fully qualified name
      - To work, the classpath must be set properly
    - `String name = "java.io.File"; Class classObj = Class.forName(name);`
- Java uses class objects (instances of Class) to represent the types of all entities:
  - Ordinary objects
  - Primitives (int, float, char, etc)
    - Although primitives are not objects, Java uses class objects to represent their type
    - Use a class literal to specify the class object
    - `int.class, double.class`
    - `void.class` represents the void return type
    - To check if primitive, use `isPrimitive()` on the class object (`if (classObject.isPrimitive()) ...`)
  - Arrays
    - Java arrays are objects whose classes are created at runtime by the JVM
    - A new class for each element type and dimension
    - Use a class literal to specify the class object
      - `int[].class.Object[].class`
    - To check if an array, use `isArray()`
    - To find the base type of an array, use `public Class getComponentType()`
  - Interfaces
    - Each declared interface is represented with a class object
    - Can be specified with a class literal (`Collection.class`)
    - Can be queried for supported methods and constants
    - To check if an interface, use `isInterface()`

Methods:

- Methods for a class or interface are represented with metaobjects of the type `java.lang.reflect.Method`
- Methods can be found at runtime by querying the class object
  - To find a **public** method (either declared or inherited), use `Method getMethod(String name, Class[] paramTypes);`
  - Eg: `Method m = classObject.getMethod("setColor", new Class[] { Color.class });`
  - If no parameters, use `null` or zero-length array for the 2nd argument
- Use `getDeclaredMethod()` to find a method explicitly declared by the class (not inherited)
  - Returns methods of all visibilities (public, protected, package, private)
- To find *all* public methods of a class (either declared or inherited) use: `Method[] getMethods()`
  - To get all declared methods of any visibility: `Method[] getDeclaredMethods()`
- A method object can be queried with:
  - `String getName()`
  - `Class getDeclaringClass()`
  - `Class[] getExceptionTypes()`
  - `Class[] getParameterTypes()`
  - `Class getReturnType()`
  - `int getModifiers()`
    - The returned int can be decoded with methods in Modifier class
- To call a method dynamically, use: `Object invoke(Object obj, Object[] args)`

```
Object myObj = new ...;
Class classObject = myObj.getClass();
Color c = new ...;
Method m = classObject.getMethod("setColor", new Class[] {Color.class});
m.invoke(myObj, new Object[] {c});
```

  - If there are no arguments, use null or zero length array for the second parameter
  - If a static method, use null for the 1st parameter
- Primitives are passed as parameters by putting them into a "wrapper object"

```
int i = 10;
Method m = classObject.getMethod("get", new Class[] {int.class});
m.invoke(myObj, new Object[] {new Integer(i)});
```

- If a method normally returns a primitive, `invoke()` will return the primitive in a wrapper object
  - Since typed as Object, you must cast it to the correct type
  - Then unwrap it using an xxx Value() method
  - Note: Java 5.0 introduced automatic boxing and unboxing

```
int code;
Method m = classObject.getMethod("hashCode", null);
code = ((Integer)m.invoke(myObj, null)).intValue();
```

- To find the superclass object of a class object, use `Class getSuperClass()`
  - `Class superclassObject = classObject.getSuperClass();`
  - Returns null if `classObject` represents a primitive type, void, an interface, or Object class
  - Returns class object for Object if an array
- Use `Class[] getInterfaces()` on a class object to find all interfaces that the class directly implements
  - If used on a class object that represents an interface, then returns the direct superinterfaces

Fields:

- Fields for a class or interface are represented with metaobjects of the type `java.lang.reflect.Field`
  - Fields can be found at runtime by querying the class object
  - To find a public field (either declared or inherited), use:
    `Field getField(String name)`
    `Field f = classObject.getField("id");`
  - Use `getDeclaredField(String name)` to find a field explicitly declared by the class or interface (not inherited)
    - Returns fields of all visibilities
  - To find *all* public fields of a class (either declared or inherited) use
    `Field[] getFields()`
    `Field fArray[] = classObject.getFields();`
  - To find *all* declared fields of any visibility, use `Field[] getDeclaredFields()`
- A Field object can be queried with:
  - `String getName()`
  - `Class getDeclaringClass()`
  - `Class getType()`
  - `int getModifiers()`
    - The returned int can be decoded with methods in Modifier class
- You can find the value of a field reflectively using `Object get(Object obj)`

```
Object myObj = new ...
Class classObject = myObj.getClass();
Field f = classObject.getDeclaredField("id");
Object value = f.get(myObj);
```

  - If the field type is primitive, the returned value is wrapped in the appropriate wrapper object
  - If you know the type of the primitive you can access the value directly using methods like
    `boolean getBoolean(Object obj)`, `double getDouble(Object obj)`
    `int value = f.getInt(myObj)`
- Fields can be set reflectively using `void set(Object obj, Object value)`
  - Eg. `f.set(myObj, newValue);`
  - You must wrap primitive values, or use methods like
    `void setBoolean(Object obj, boolean value)`, `void setDouble(Object obj, double value)`, etc
    `f.setInt(myObj, 37);`

Modifiers:

- Any Class, Method, or Field object can be queried using `getModifiers()`
  - Returns an int where particular bits represent one of the 11 modifiers in java
    - `public`, `protected`, `private`, `static`, `abstract`, etc
  - Can be decoded using static methods in `java.lang.reflect.Modifier`
    - `boolean isPublic(int mod)`
    - `boolean isProtected(int mod)`
    - etc

```
Field f = classObject.getField("id");
int mod = f.getModifiers();

if (Modifier.isStatic(mod)) { ... }
```

- Can print out all modifiers using `toString(int mod)`
  `System.out.println(Modifier.toString(mod));`
- Normally, non-public fields and methods cannot be accessed from outside the class
  - Access checking can be bypassed using `void setAccessible(boolean flag)`

```
f.setAccessible(true);
Object value = f.get(myObj)
```

Arrays:

- `java.lang.reflect.Array` provides static methods to operate reflectively on array objects
  - `Object newInstance(Class componentType, int length)`
    `Object myArray = Array.newInstance(int.class, 10);`
  - `int getLength(Object array)`
    `int length = Array.getLength(anObj);`
- `Object get(Object array, int index)`
  - Returns the element at index, wrapping primitives if necessary
  - `Object obj = Array.get(myArray, 3);`
  - Wrapper methods like `getBoolean(...)`, `getDouble(...)`, etc
    - `int i = Array.getInt(myArray, 3);`
- `void set(Object array, int index, Object value)`
  - Sets the element at index to a specified value, unwrapping primitives if necessary
  - Also has methods like `setBoolean(...)`, `setDouble(...)`, etc
  - `Array.setInt(myArray, i, iVal);`

Constructors:

- Constructors for a class are represented with metaobjects of the type `java.lang.reflect.Constructor`
- Constructors can be found at runtime by querying the class object
  - To find a public constructor (either declared or inherited) use
    `Constructor getConstructor(Class[] parameterTypes)`

```
Constructor c;
c = ClassObject.getConstructor(new Class[] {int.class, double.class});
```

  - If no parameters, use `null` or zero-length array for the argument
  - Throws `NoSuchMethodException` if not found
- Use `getDeclaredConstructor(...)` to find a constructor (of any visibility) explicitly declared by the class
- To find *all* public constructors of a class (inherited or declared) use
  `Constructor[] getConstructors()` `Constructor cArray[] = classObject.getConstructors();`
- To find all declared constructors of any visibility, use `Constructor[] getDeclaredConstructors()`
- A constructor object can be queried with:
  - `String getName()`
  - `Class getDeclaringClass()`
  - `Class[] getExceptionTypes()`
  - `Class[] getParameterTypes()`
  - `int getModifiers()`

Reflective Instantiations:

- Can be done using `newInstance()` on the class object

```
class classObject = ...
Object myObj = classObject.newInstance();
```

- Implicitly uses the no-arg constructor
- Can be done using a constructor metaobject and the method:
  `Object newInstance(Object[] initargs)`

```
Constructor c;
int iVal;
...
c = classObject.getConstructor(new Class[] {int.class});
Object myObj = c.newInstance(new Object[] {new Integer(iVal)});
```

Simple Example:

```
import java.lang.reflect.*;

public class Test {
        public static void main(String[] args) {
                Object object = null;
                Class classObject = null;

                try {
                        // load the class dynamically using 1st command-line arg
                        classObject = Class.forName(arg[0]);

                        // Create an instance of the class
                        object = classObject.newInstance();
                } catch (...) {
                        // InstantiationException IllegalAccessException ClassNotFoundException
                }

                try {
                        // Find the no-arg method named by 2nd command-line arg
                        Method m = classObject.getMethod(arg[1], null);

                        // Invoke the method on the object
                        m.invoke(object, null);
                } catch (...) {
                        // NoSuchMethodException IllegalAccessException InvocationTargetException
                }
        }
}
```

- Can be used on any class
- Example:

```
public class myClass {
        public void print() {
                System.out.println("Hello, world!");
        }
        public void display() {
                System.out.println("Goodbye, cruel world!");
        }
}
```

- Can be invoked like so:

```
> java Test MyClass print
Hello, world!
> java Test MyClass display
Goodbye, cruel world!
```