ECE 651
Lecture 3: OO Basics
Notes Outline

- Review of classes, fields, and methods:
  - Classes:
  - Methods:
  - Fields:
- Has-A (composition):
- Is-A (inheritance):
- Has-A vs Is-A practice:
  - A Person (has-A or is-A) Head
  - A SportsCar (has-A or is-A) Car
  - A Zoo (has-A or is-A) List¡Animal¿
  - A HashTable¡T¿ (has-A or is-A) Vector¡T¿
  - A WoodenDoor (has-A or is-A) PieceOfWood
  - A Shirt (has-A or is-A) PieceOfClothing
  - An EncryptedFileReader (has-A or is-A) FileReader

- Encapsulation:

- Avoiding get/set methods:

- Review of static

- Low-Coupling/High-Cohesion

  - Example of tight (high) coupling:

```
array_t * getSortedStrings(FILE * f) {
   array_t * arrayToSort = malloc(sizeof(*arrayToSort));
   arrayToSort->array = NULL;
   arrayToSort->n = 0;
   char * curr = NULL;
   size_t sz = 0;
   size_t i = 0;
   while (getline(&curr, &sz, f) >= 0) {
     arrayToSort->array =
        realloc(arrayToSort->array, ((i+1) * sizeof(*arrayToSort->array)));
     arrayToSort->array[i] = curr;
     curr = NULL;
     i++;
   }
   free(curr);
   arrayToSort->n = i;
   sortData(arrayToSort->array, arrayToSort->n);
   return arrayToSort;
}
```

(continued next page)

```
void printSortedFile(FILE * f){
   array_t * sortedArray = getSortedStrings(f);
   for (size_t i = 0; i < sortedArray->n; i++) {
      printf("%s", sortedArray->array[i]);
      free(sortedArray->array[i]);
   }
   free(sortedArray->array);
   free(sortedArray);
}
```

– Example of loose (low) coupling:

```
array_t * readStringsFromFile(FILE * f) {
   array_t * answer = malloc(sizeof(*answer));
   answer->array = NULL;
   answer->n = 0;
   char * curr = NULL;
   size_t sz = 0;
   size_t i = 0;
   while (getline(&curr, &sz, f) >= 0) {
     answer->array =
        realloc(answer->array, ((i+1) * sizeof(*answer->array)));
     answer->array[i] = curr;
     curr = NULL;
     i++;
   }
   free(curr);
   answer->n = i;
   return answer;
}

void printArray(array_t * data){
   for (size_t i = 0; i < data->n; i++) {
      printf("%s", data->array[i]);
   }
}
void freeArray(array_t * data){
   for (size_t i = 0; i < data->n; i++) {
      free(data->array[i]);
   }
   free(data->array);
   free(data);
}
```

- Decoupling: combining is ok, as long as you can separate

- Decoupling: input, operation, output, cleanup

- JSON input format:

```
{
"items": [{
          "type": "burger",
          "count": 2
          },
          {
          "type": "shake",
          "variety": "chocolate",
          "count": 1
          }
         ],
"payment": {
           "type": "cash",
           "amount": 7.93
           }
}
```

- Tightly coupled JSON processing (bad):

```
FoodBag packOrder(String orderString){
  FoodBag fb = new FoodBag();
  JsonObject order = JsonReader.readObject(orderStr);
  JsonArray items = order.getJsonArray(\items");
  for (int i = 0; i < items.size(); i++) {
     JsonObject food = items.getJsonObject(i);
     int count = food.getInt(\count");
     String variety = null;
     if (food.hasKey(\variety")) {
         variety = food.getString(\variety");
     }
     for (int i = 0; i < count; i++) {
        fb.addFood(food.getString(\type"), variety);
  }
  return fb;
}
```

- Decouple program state from input format
  - Program State:
- Model, View, Controller (MVC):

  Note: we'll return to this when we talk about GUIs
- Strings
- Hardcoding
- Objects vs primitives

- Decoupling error handling

  - 551 error handling: print and exit

```
Placement(String descr) { //Placement constructor that takes a String
     //some code
     if (some error) {
        System.err.println(\The placement was invalid");
        System.exit(1);
     }
    //other code that returns the answer
}
```

  - Using an exception instead:

```
  Placement(String descr) {
     //some code
     if (some error) {
        throw new IllegalArgumentException(\The placement was invalid");
     }
    //other code that returns the answer
}
```

  - Exception handler can choose what to do:

```
Placement playTurn() {
   output.println(\Where would you like to place a ship?");
   Placement p = null;
   while(p == null) {
      try {
         p = readPlacement(input); //reads line from input, does new Placement
      }
      catch(IllegalArgumentException iae) {
         output.println(iae.getMessage());
         output.println(\Please try again");
      }
   }
   return p;
}
```

- Testing exception behavior:

```
class PlacementTest {
  @Test
  public void test_constructor_valid() {
    Placement p1 = new Placement(\A1H");   //valid placement
    assertEquals(0, p1.getCoordinate().getRow());
    assertEquals(0, p1.getCoordinate().getColumn());
    assertEquals('H', p1.getOrientation());
    //more cases here...
  }
  @Test
  public void test_constructor_errors() {
    assertThrows(IllegalArgument.class, () -> new Placement(\A0")); //missing orientation
    assertThrows(IllegalArgument.class, () -> new Placement(\A0VV")); //two orientations
    assertThrows(IllegalArgument.class, () -> new Placement(\0AH")); //wrong order
    //more cases here...
  }
}
```

- Executable: a `@FunctionalInterface`

```
@FunctionalInterface
public interface Executable {
  public void execute() throws Throwable;
}
```

- What does assertThrows do?

```
public <T extends Throwable>
T assertThrows(Class<T> expected, Executable todo){
   try {
       todo.execute();
       fail(\Did not throw any exception");
   }
   catch(Throwable t){
       assertTrue(expected.isAssignableFrom(t.getClass());
       return expected.cast(t);
   }
}
```

- Lambdas:

```
assertThrows(IllegalArgument.class,  ()-> new Placement(\A0"));
```

  Shorthand for

```
class PlacementConstructorA0Executor implements Executable{
  public void execute()  {
      new Placement(\A0");
   }
}
....
assertThrows(IllegalArgument.class, new PlacementConstructorA0Executor());
```

- Exiting as error behavior:

- Differ in data vs differ in behavior:
  - Data:
  - Behavior:

  - Goal:

- Data vs behavior: Polymorphism and dynamic dispatch