Classes

Classes

Classes

Anatomy of a Class

By the end of next lecture, you'll understand everything in this class definition.

```
package edu.gatech.cs1331.card;

import java.util.Arrays;

public class Card {

  public static final String[] VALID_RANKS = {"2", ... , "ace"};
  public static final String[] VALID_SUITS = {"diamonds", ... };
  private String rank;
  private String suit;

  public Card(String aRank, String aSuit) {
    // ...
  }
  public String toString() {
    return rank + " of " + suit;
  }
  private boolean isValidRank(String someRank) { ... }
}
```

The Card Class Example

In this lecture we'll use a running example stored in a Git repo. Go to https://github.com/cs1331/card, click on "Clone or download" and copy the clone URL. Then open your terminal and do this:

```
cd cs1331
git clone https://github.com/cs1331/card.git
cd card
```

Note that if you've uplaoded your public SSH key you may use an SSH clone URL.

Now you're ready to follow along. Start by checking out v0.0 [1]:

```
git checkout v0.0
```

[1] Semantic Versioning

A Card Class, v0.0

Consider how to represent a Card ADT:

- rank - the rank of a playing card, e.g., 2, jack, ace
- suit - the suit of a playing card, e.g., spades, diamonds

```
public class Card {

    String rank;
    String suit;
}
```

- `rank` and `suit` are instance variables
- Every instance of `Card` has its own copy of instance variables.

Let's "do something" with `Card` by adding a main method and bumping to v0.1.

Card v0.1

After

```
git checkout v0.1
```

we have (we won't include the git commands in future slides):

```
public class Card {

    String rank;
    String suit;

    public static void main(String[] args) {
        Card c = new Card();
        System.out.println(c);
    }
}
```

Note that we can put a `main` method in any class. This is useful for exploratory testing, like we're doing here.

The `String` representation isn't very appealing. (What does it print?)

Card v0.2

```
public class Card {
    String rank;
    String suit;

    public String toString() {
        return rank + " of " + suit;
    }
    public static void main(String[] args) {
        Card swedishPop = new Card();
        swedishPop.rank = "ace";
        swedishPop.suit = "base";
        Card handy = new Card();
        handy.rank = "jack";
        handy.suit = "all trades";
        System.out.println(swedishPop);
        System.out.println(handy);
    }
}
```

Now we have a nice `String` representation, but we havve an "ace of base" card and a "jack of all trades", which aren't valid cards.

Encapsulation: Card, v1.0

Let's protect the instance variables by making them private:

```
public class Card {
    private String rank;
    private String suit;

    public String toString() {
        return rank + " of " + suit;
    }

    public static void main(String[] args) {
        Card c = new Card();
        c.rank = "ace";
        c.suit = "base";
        System.out.println(c);
    }
}
```

Why does this still compile?

- `main` method in `Card` – can see `Card` 's private parts

A Dealer Class, v1.1

```
public class Dealer {

    public static void main(String[] args) {
        Card c = new Card();
        c.rank = "ace";
        c.suit = "base";
        System.out.println(c);
    }
}
```

This won't compile (which is what we want). Why?

Mutators: Card, v1.2

```
public class Card {

    private String rank;
    private String suit;

    public void setRank(String rank) {
        rank = rank;
    }
    public void setSuit(String suit) {
        suit = suit;
    }
}
```

- Now client code can set the rank and suit of a card by calling `setRank` and `setSuit` .
- `setX` is the Java convention for a setter method for an instance variable named `x` .

Dealing Card , v1.2

Let's try out our new Card class.

```
public class Dealer {

    public static void main(String[] args) {
        Card c = new Card();
        c.setRank("ace");
        c.setSuit("base");
        System.out.println(c);
    }
}
```

Oops. Prints "null of null". Why?

Shadowing Variables

The parameters in the setters "shadowed" the instance variables:

```
public void setRank(String rank) {
    rank = rank;
}

public void setSuit(String suit) {
    suit = suit;
}
```

- `rank` in `setRank` refers to the local `rank` variable, not the instance variable of the same name
- `suit` in `setSuit` refers to the local `suit` variable, not the instance variable of the same name

Georgia Tech

Dealing with `this` : Card, v1.2.1

```
public class Card {
    private String rank;
    private String suit;

    public void setRank(String rank) {
        this.rank = rank;
    }
    public void setSuit(String suit) {
        this.suit = suit;
    }
}
```

- Every instance of a class has a `this` reference which refers to the instance on which a method is being called.
- `this.rank` refers to the `rank` instance variable for the `Card` instance on which `setRank` is being called.
- `this.rank` is different from the local `rank` variable that is a parameter to the `setRank` method.

Georgia Tech

Dealing Card , v1.2.1

```
public class Dealer {

    public static void main(String[] args) {
        Card c = new Card();
        c.setRank("ace");
        c.setSuit("base");
        System.out.println(c);
    }
}
```

Now we have encapsulation, but we can still create invalid Card s, e.g., "base" is not a valid suit. How to fix?

Class Invariants

Class invariant: a condition that must hold for all instances of a class in order for instances of the class to be considered valid.
Invariants for Card class:

- `rank` must be one of {"2", "3", "4", "5", "6", "7", "8", "9", "10", "jack", "queen", "king", "ace"}
- `suit` must be one of {"diamonds", "clubs", "hearts","spades"}

Georgia Tech

Class Invariants: Card v1.3

`rank` invariant can be maintained by adding:

```
public class Card {
    private final String[] VALID_RANKS =
        {"2", "3", "4", "5", "6", "7", "8", "9",
         "10", "jack", "queen", "king", "ace"};
    public void setRank(String rank) {
        if (!isValidRank(rank)) {
            System.out.println(rank + " is not a valid rank.");
            System.exit(0);
        }
        this.rank = rank;
    }
    private boolean isValidRank(String someRank) {
        return contains(VALID_RANKS, someRank);
    }
    private boolean contains(String[] array, String item) {
        for (String element: array) {
            if (element.equals(item)) {
                return true;
            }
        }
        return false;
    }
    // ...
}
```

Class Invariants Ensure Consistent Objects

Now we can't write code that instantiates an invalid `Card` object:

```
public class Dealer {
    public static void main(String[] args) {
        Card c = new Card();
        c.setRank("ace");
        c.setSuit("base");
        System.out.println(c);
    }
}
```

yields:

```
$ java Dealer
base is not a valid suit.
```

Dealer v1.3.1

Version 1.3.1 fixes the invalid suit:

```
public class Dealer {

    public static void main(String[] args) {
        Card c = new Card();
        c.setRank("ace");
        c.setSuit("spades");
        System.out.println(c);
    }
}
```

Georgia Tech

Initializing Instances (v1.4)

`Card` now ensures that we don't create card objects with invalid ranks or suits. But consider this slight modification to `Dealer` in v1.4:

```
public class Dealer5 {

    public static void main(String[] args) {
        Card c = new Card();
        System.out.println(c); // Printing a new Card instance
        c.setRank("ace");
        c.setSuit("base");
        System.out.println(c);
    }
}
```

What if we printed our `Card` instance, `c` , before we called the setters?

Object Initialization

Two ways to initialize the instance variables of an object:

- Declaration point initialization:

```
public class Card {
    private String rank = "2";
    // ...
}
```

- Constructors

```
public class Card {
    public Card() {
      rank = "2";
    }
    // ...
}
```

A constructor is what's being called when you invoke operator `new` .

Initializing Objects

Since we didn't write our own constructor, Java provided a default no-arg constructor

- default no-arg ctor sets instance variables (that don't have their own declaration-point intializations) to their default values.

That's why a `Card` object's instance variables are `null` ("`null of null`") after they're instantiated. We have to call the setters on a `Card` instance before we have a valid object.

Georgia Tech

Innitialization Style

In general, it's poor style to require multi-step initialization.

- After `new Card()` is called, instance variables have useless defaults.
- Client programmer must remember to call setter methods.
- Often there can be order dependencies that we don't want to burden client programmers with.

The way to fix this is by writing our own constructor.

Georgia Tech

A Constructor for Card, v2.0

If we write a constructor, Java won't provide a default no-arg constructor.

```
public class Card {
    // ...
    public Card(String rank, String suit) {
        setRank(rank);
        setSuit(suit);
    }
    // ...
}
```

If we want a no-arg constructor in addition to other constructors, we must write it explicitly.

Dealer v2.0

Now this won't even compile:

```
public class Dealer {

    public static void main(String[] args) {
        Card c = new Card();
        // ...
    }
}
```

```
$ javac Dealer.java
Dealer.java:4: error: constructor Card in class Card cannot be
     applied to given types;
        Card c = new Card();
                 ^
  required: String,String
  found: no arguments
  reason: actual and formal argument lists differ in length
1 error
```

Using the Card v2.0.1 Constructor

Now we have a safer, more consistent way to initialize objects:

```
public class Dealer {

    public static void main(String[] args) {
        Card c = new Card("queen", "hearts");
        System.out.println(c);
    }
}
```

Georgia Tech

Progress Check

Let's review our progress with our Card class design:

- We have a nice string representation of Card objects.
- We have encapsulated the rank and suit in private instance variables with mutator methods to set their values.
- We validate the rank and suit in the mutator methods so we can't set invalid ranks and suits in Card objects.
- `Card` has a constructor, which ensures that instance variables are initialized when an instance of `Card` is created.

Georgia Tech

Valid Ranks and Suits

Recall the declarations and definitions of our `VALID_RANKS` and `VALID_SUITS` :

```
public final String[] VALID_RANKS =
    {"2", "3", "4", "5", "6", "7", "8", "9",
     "10", "jack", "queen", "king", "ace"};

public final String[] VALID_SUITS =
    {"diamonds", "clubs", "hearts","spades"};
```

Do we need a separate instance of `VALID_RANKS` and `VALID_SUITS` for each instance of our Card class?

Static Members, Card v2.1

`static` members are shared with all instances of a class:

```
public static final String[] VALID_RANKS =
    {"2", "3", "4", "5", "6", "7", "8", "9",
     "10", "jack", "queen", "king", "ace"};
public static final String[] VALID_SUITS =
    {"diamonds", "clubs", "hearts","spades"};
```

Given the declarations above:

- Each instance shares a single copy of `VALID_RANKS` and a single copy of `VALID_SUITS`
- Since they're `final` , we can safely make them `public` so clients of our Card class can use them

Stateful Card Objects?

Card v2.1 is pretty good, but we can write code like this:

```
public class Dealer {

    public static void main(String[] args) {
        Card c = new Card("queen", "hearts");
        System.out.println(c);
        c.setRank("jack"); // modifying c
        System.out.println(c);
    }
}
```

Does this make sense? Should Card objects be mutable?

Immutable Objects

Card objects don't change. We can model this behavior by removing the setters and putting the initialization code in the constructor (or making the setters private and calling them from the constructor):

```
public Card(String aRank, String aSuit) { // constructor
  if (!isValidRank(rank)) {
    System.out.println(aRank + " is not a valid rank.");
    System.exit(0);
  }
  rank = aRank;
  if (!isValidSuit(aSuit)) {
    System.out.println(aSuit + " is not a valid suit.");
    System.exit(0);
  }
  suit = aSuit;
}
```

Note the use of another idiom for disambiguating constructor paramters from instance variables (as opposed to using `this`).

A Few Final Bits of Polish

Take a look at the final evolution of our Card class.

```
git checkout master
```

It contains a few more enhancements:

- It has an `equals()` method for comparing cards of equal value.
  - We'll learn the role of the `equals` method today and learn how to write one in a couple of weeks.
- It uses `enum` s for rank and suit.

Georgia Tech

Enums

```
public enum Rank {
    TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK,
        QUEEN, KING, ACE
}
```

```
public enum Rank {
    TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK,
        QUEEN, KING, ACE
}
```

Now we get static type checking for `Rank` and `Suit` – no need for run-time validity checks.

```
public class Card {

    private Rank rank;
    private Suit suit;

    public Card(Rank aRank, Suit aSuit) {
        rank = aRank;
        suit = aSuit;
    }
...
}
```

Equality

- For reference types:
  - == means identity equality (aliasing testing).
  - `equals(Object)` tests value equality, as defined by the class.
- For primitive types == means value equality and is the only equality test.

Here's the final `Card` class, with a definition of value equality for `Card` instances:

```
public class Card {
    private Rank rank;
    private Suit suit;
    public Card(Rank aRank, Suit aSuit) {
        rank = aRank;
        suit = aSuit;
    }
    public String toString() { return rank + " of " + suit; }

    public boolean equals(Object other) {
        if (null == other) { return false; }
        if (this == other) { return true; }
        if (!(other instanceof Card)) { return false; }
        Card that = (Card) other;
        return this.rank.equals(that.rank) &&
            this.suit.equals(that.suit);
    }
}
```

Equality Tests

Given our finished Card class with a properly implemented `equals(Object)` method, this code:

```
Card c1 = new Card(Rank.ACE, Suit.SPADES);
Card c2 = new Card(Rank.ACE, Suit.SPADES);
Card c3 = c1;
System.out.println("c1 == c2 returns " + (c1 == c2));
System.out.println("c1.equals(c2) returns " + c1.equals(c2));
System.out.println("c1 == c3 returns " + (c1 == c3));
System.out.println("c1.equals(c3) returns " + c1.equals(c3));
```

produces this output:

```
c1 == c2 returns false
c1.equals(c2) returns true
c1 == c3 returns true
c1.equals(c3) returns true
```

By the way, what if we left off the parentheses around `c1 == c2` in `System.out.println("c1 == c2 returns " + (c1 == c2))` ?

Exercise: Treating People as Objects

Using the encapsulation techniques we just learned, write a class named `Person` with a `name` field of type `String` and an age field of type `int` .
Write a suitable `toString` method for your `Person` class.
Add a main method that:

- Creates an array of `Person` objects
- Iterates through the array and prints each Person object who's age is greater than 21

Georgia Tech