McGill

# COMP 303

## Lecture 7

# Object state

Winter 2025

slides by Jonathan Campbell

# Announcements

- Brainstorming meetings.
- Server registration.
- Class diagrams.

# Plan for this week

- Objects
- State, state space, state diagrams, object life cycles and final fields
- Nullability and avoiding null values (Optional<T>, null design pattern)
- Object properties (ID, equality, uniqueness) and Singleton/Flyweight design patterns

# Recap

# Static vs. dynamic

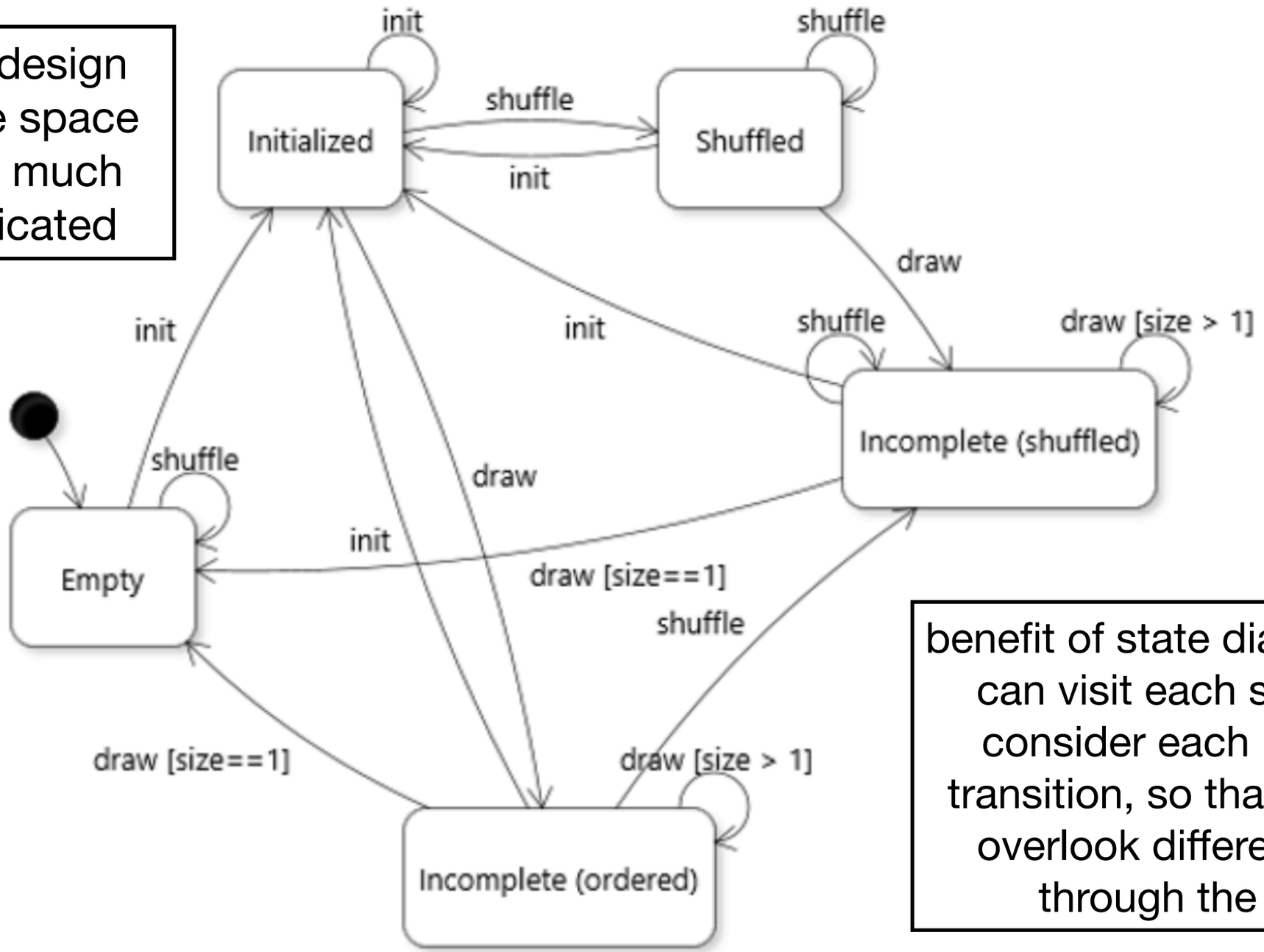
- Two complementary perspectives when modelling a software system:
  - Static (compile-time): the elements in the code and their relations.
    - look through source code, class diagrams.
  - Dynamic (run-time): the values and references held by variables at different points in the program.
    - make object, state and sequence diagrams.
    - think about object state.

# Object state

- Concrete state: values stored in an object's fields.
  - State space: the set of possible states for a variable. (Very large for primitive objects.)
- Abstract state: arbitrarily-defined subset of concrete state space.
  - E.g., the abstract state "Three Kings" could represent any possible configuration of a Deck where exactly three cards of Rank.King are present. Or, "Empty" for a Deck.
  - Or "Non-zero score" for a Player object.

# State diagram for a Deck

with a small design change, state space can become much more complicated




benefit of state diagrams: we can visit each state and consider each possible transition, so that we don't overlook different paths through the code

# SPECULATIVE GENERALITY

- We can try to eliminate certain states from the life cycle of an object.

- E.g., does a Deck need to have an uninitialized state? Or a "complete and unshuffled" state?

- SPECULATIVE GENERALITY anti-pattern ("one day it may be useful").

# TEMPORARY FIELD

- We may try to improve performance by caching certain information. But, oftentimes, the time savings doesn't make up for the decrease in simplicity in the code.

- TEMPORARY FIELD anti-pattern.

# Final fields and variables

- We want to keep the size of the abstract state space for objects of a class to be the smallest possible, to keep the number of meaningful abstract states small.

- One way to do so is to prevent changing the value of a field after initializing, making the value remain constant after its first assignment.

# Meaning of a null reference

- Variable is temporarily un-initialized, but will be initialized in a later method.
- Variable is incorrectly initialized, because we forgot to call a method.
- A flag that purposefully represents the absence of a value.
- A flag that has some other sort of special interpretation.

# Absent values without null

- How could we write the Card class to represent a Joker, a special card with no rank nor suit?

# Object state pt. 2

# Absent values without null

- Two ways:
  - Optional<T> type
  - Null object design pattern
- Note: These don't really exist in Python, so on the exams, we would ask for a Java solution.

# Solution 1: optional types

- Optional<T> library class.

- Generic type that acts as a wrapper for an instance of type T, but can also be empty.

# Solution 1: optional types

```
public class Card {
   private Optional<Rank> aRank;
   private Optional<Suit> aSuit;
   public Card() {
      aRank = Optional.empty();
      aSuit = Optional.empty();
         // or Optional.of(value)
         // or Optional.ofNullable(value)
   }
   public boolean isJoker() {
      return !aRank.isPresent();
   }
}
```

To get the value stored within, we call get().

# Handling optional types

```
public class Card {
   private Optional<Rank> aRank;
   private Optional<Suit> aSuit;

   public Optional<Rank> getRank() {
      return aRank;
   }

   public Optional<Suit> getSuit() {
      return aSuit;
   }
}
```

Getters could return Optional<T>, meaning that the client code must deal with it.

Makes client code more messy, but lets client handle absent value.

# Handling optional types

```
public class Card {
  private Optional<Rank> rank;
  private Optional<Suit> suit;

  public Rank getRank() {
    return rank.orElseThrow(() -> new
IllegalStateException("Card has no rank"));
  }

  public Suit getSuit() {
    return suit.orElseThrow(() -> new
IllegalStateException("Card has no suit"));
  }
}
```

Alternative: Getters can "unwrap" the optional object and throw an exception to make sure an invalid value can never be returned.

Good if most Card instances have rank and suit, or if it is preferred to stop immediately on an incorrect usage.

# Optional<T> vs null

- Using Optional<T> is more work than using null.
  - Client code still has to check if there is an absent value either way.
- But, it explicitly specifies that there is an absent value, vs. all the other potential meanings of the null reference. That helps us (or the client code) to know what is going on.
- It also lets the type system tell us if we made any errors, and so avoids the NullPointerException.

# Solution 2: Null Object pattern

- Use a special object to represent the null value,
  and test for absence using a polymorphic method call.

- Avoids the issue of unpacking wrapper objects.

- But, only applicable to situations where a type hierarchy is available.

  - (Cannot be used on Cards because they are not a subtype of any other user-defined type.)

# Solution 2: NULL OBJECT pattern

- As an example usage scenario, suppose some class (e.g., SolitaireGame) stores an object of type CardSource.

- Now suppose that at some time(s) during execution, a CardSource may not be available, and we may then want to assign the null reference to the field.

- How can we avoid using null?

# Solution 2: Null Object pattern

- Instead of handling absent card sources like this:

```
CardSource aSource = null;
```

- we can do this:

```
CardSource aSource = new NullCardSource();
```

- then, later, assuming that the field has a getter getSource:

```
if (source.isNull()) {...}
```

# Solution 2: NULL OBJECT pattern


isNull is implemented in all CardSource objects,
but only in NullCardSource will the method return true.

# Solution 2: NULL OBJECT pattern

- To create the NullCardSource class, we could define a constant in the CardSource interface with an anonymous class.

# Solution 2: NULL OBJECT pattern

defining a constant implementation

client code can access it through CardSource.NULL

```
public interface CardSource {
  CardSource NULL = new CardSource() {
    public boolean isEmpty() { return true; }
    public Card draw() {
      throw new UnsupportedOperationException(
                  "Cannot draw from a NULL CardSource")
    }
    public boolean isNull() { return true; }
  };
  Card draw();
  boolean isEmpty();
  default boolean isNull() { return false; }
}
```

overriding default method

default method in interface --
applies to all implementing types

# Solution 2: NULL OBJECT pattern

```
public interface CardSource {
  CardSource NULL = new CardSource() {
    public boolean isEmpty() { return true; }
    public Card draw() {
      throw new UnsupportedOperationException(
                  "Cannot draw from a NULL CardSource")
    }
    public boolean i
  };
  Card draw();
  boolean isEmpty();
  default boolean is
}
```

We can call draw() on a NULL CardSource without getting a NullPointerException (like we would if we tried to call it on a null reference).

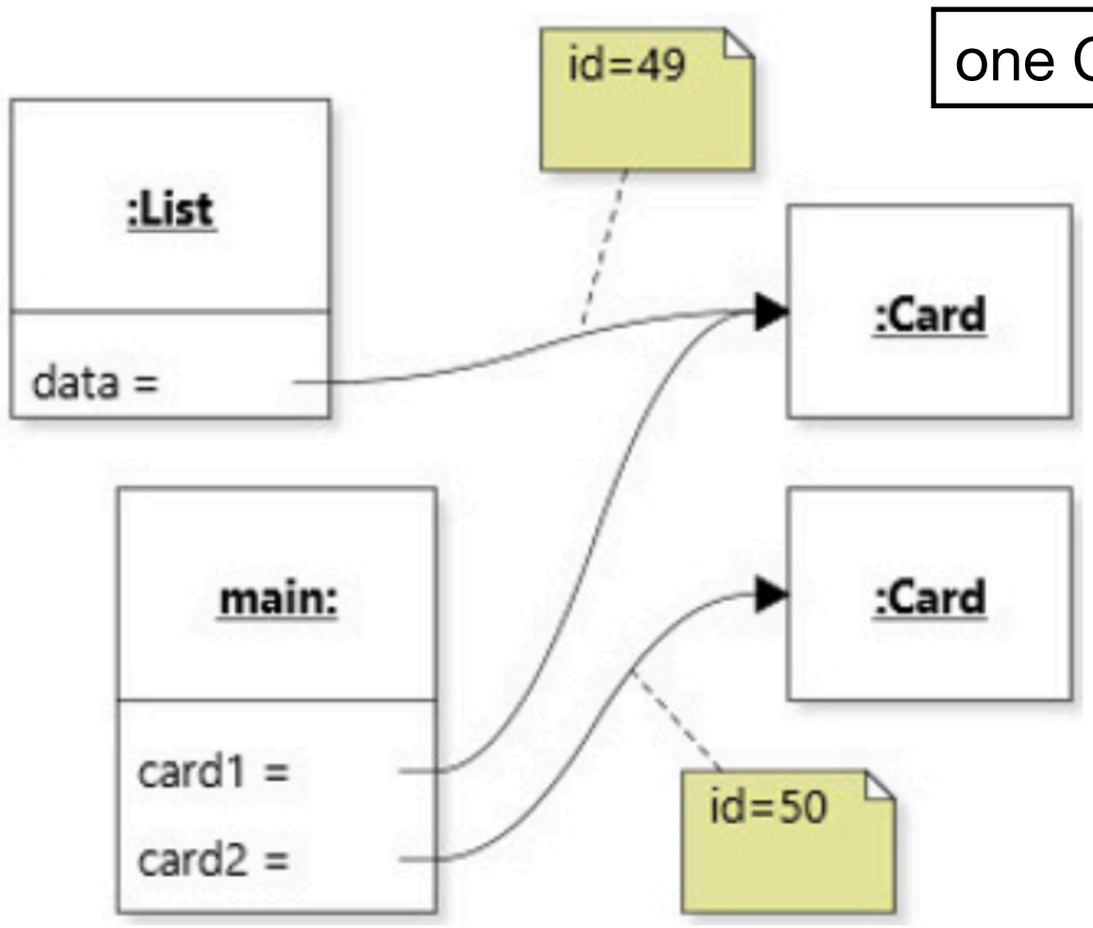
draw() will end up throwing an error too, but it is a controlled failure.

(We could also provide some other behaviour that does not throw an error, if it made sense for the context.)

# Object properties

- Identity
  - E.g., memory location.
- Equality
  - Depends on identity for reference objects, unless otherwise specified.
- Uniqueness
  - Should there be duplicate objects with identical field values?

# Object identity




two new Cards created

# Object equality

- == operator for reference types returns True if the two operands share the same identity (memory location).

```
Card card1 = new Card(Rank.ACE, Suit.CLUBS);
Card card2 = new Card(Rank.ACE, Suit.CLUBS);
System.out.println(card1 == card2); // False

Card card1 = new Card(Rank.ACE, Suit.CLUBS);
Card card2 = card1;
System.out.println(card1 == card2); // True
```

# Object equality

- To override this general behaviour, we can provide an implementation for the equals(Object) method, and then use that method for comparisons.

- In Java, we also have to override hashCode, so that it returns the same result for two objects which are equal to each other.

# Object equality

```
public boolean equals(Object pObject) {
  if (pObject == null) {
    return false; // As required by the specification
  }
  else if (pObject == this) {
    return true; // Standard optimization
  }
  else if (pObject.getClass() != getClass()) {
    return false;
  }
  else {
    // Actual comparison code
    return aRank == ((Card)pObject).aRank &&
      ((Card)pObject).aSuit == aSuit;
  }
}
```

# Object uniqueness

- Should there be two separate Card objects for the same playing card (e.g., Ace of Clubs)?

- If we can guarantee that two distinct objects cannot be equal to each other, then we don't have to define equality!

  - We can do so by using the Flyweight and/or Singleton design patterns.

# FLYWEIGHT design pattern

- Cleanly manage collections of low-level immutable objects, ensuring uniqueness of objects of a class.

- Used when immutable instances of a class are heavily shared. E.g., Card objects in a Solitaire program.

# FLYWEIGHT design pattern

- Main idea: control the creation of objects of a class (called the flyweight class) through an access method, ensuring that no duplicate objects can ever exist.

# FLYWEIGHT design pattern

- Three components:
  - private constructor, so that clients cannot construct their own objects.
  - static flyweight store (collection of flyweight objects).
  - static access method that returns the appropriate flyweight object. It checks if the object exists yet; if not, it creates it.

# FLYWEIGHT design pattern

```
public class Card {
   private final Rank aRank;
   private final Suit aSuit;
   public Card(Rank pRank, Suit pSuit) {
      aRank = pRank;
      aSuit = pSuit;
   }
   /* Includes equals and hashCode
      implementations */
}
```

not flyweight -- anyone can call constructor and make new Cards of the same rank/suit

# FLYWEIGHT design pattern

```
public class Card {
   private final Rank aRank;
   private final Suit aSuit;
   private Card(Rank pRank, Suit pSuit) {
      aRank = pRank;
      aSuit = pSuit;
   }
   /* Includes equals and hashCode
      implementations */
}
```

step 1: private constructor

# FLYWEIGHT design pattern

we use array here to index by suit and rank -- it depends on what you want to key on

```
public class Card {
  private static final Card[][] CARDS =
                  new Card[Suit.values().length]
                          [Rank.values().length];
  static {
    for (Suit suit : Suit.values()) {
      for (Rank rank : Rank.values()) {
        CARDS[suit.ordinal()][rank.ordinal()] =
                                  new Card(rank, suit);
      }
    }
  }
}
```

step 2: flyweight store

(in this case, initialized at start, since there are only 52 possibilities)

# FLYWEIGHT design pattern

```
public class Card {
  public static Card get(Rank pRank, Suit pSuit) {
    assert pRank != null && pSuit != null;
    return CARDS[pSuit.ordinal()][pRank.ordinal()];
  }
}
```

step 3: access method

static, so that no instance is required to call

# FLYWEIGHT design pattern

```
public class Card {
  private static final Card[][] CARDS =
                    new Card[Suit.values().length]
                            [Rank.values().length];
  public static Card get(Rank pRank, Suit pSuit) {
    if (CARDS[pSuit.ordinal()][pRank.ordinal()] == null) {
      CARDS[pSuit.ordinal()][pRank.ordinal()] =
                                    new Card(pRank, pSuit);
    }
    return CARDS[pSuit.ordinal()][pRank.ordinal()];
  }
}
```

alternative: flyweight objects created in access method as needed

# SINGLETON pattern

- Goal: ensure there is only a **single** instance of a given class at any point in time.

- Such classes may store a cohesive amount of information needed in different parts of the code.

# SINGLETON pattern examples

- A global state manager, like a CardGame class that stores the full state of the game (the deck, players, etc.).

- A Logger class that stores records; different parts of the code may need to call out to it to log various events.

- A class that represents a cache of resources, which may need to be accessed by lots of different classes.

# SINGLETON pattern

- Three components:
  - Private constructor, so that clients cannot create multiple objects.
  - Global variable that holds the reference to the single class instance.
  - Accessor method, to retrieve the singleton instance.

# SINGLETON pattern

```
public class GameModel {
   private static final GameModel INSTANCE = new GameModel();
   private GameModel() { ... }
   public static GameModel instance()
   {
      return INSTANCE;
   }
}
```

# SINGLETON pattern

```
public class GameModel {
   private static final GameModel INSTANCE = new GameModel();
   public GameModel() { ... }
   public static GameModel instance()
   {
      return INSTANCE;
   }
}
```

if constructor is public, then it's not a proper Singleton pattern!

# SINGLETON pattern

- All design decisions involve trade-offs. When we choose to use a design pattern, we have to think about whether the benefits outweigh the disadvantages.

- A singleton is a global variable accessible from anywhere in the code, which can lead to messy code with lots of dependencies.

- Singleton objects live for the entire duration of the application and control their own life cycle, so they are difficult to test.

# SINGLETON VS FLYWEIGHT

- Flyweight pattern:
  - unique instances of a class (e.g., unique Card objects).
  - typically for immutable objects.
- Singleton pattern:
  - only ever one single instance of a class.
  - the instance is typically stateful and mutable.

# Flyweight in Python

```
class Card:
    _instances = {}

    def __new__(cls, rank: Rank, suit: Suit):
        self = cls._instances.get((rank, suit))
        if self is None:
            self = cls._instances[(rank, suit)] =
                                object.__new__(Card)
            self.rank = rank
            self.suit = suit
        return self
```

# Singleton in Python

```
class GameModel:
    instance = None

def __new__(cls):
  if cls._instance is None:
    cls._instance = super(GameModel, cls).__new__(cls)
     # Initialization code here.
  return cls._instance
```

# References

- Robillard ch. 4.5-4.9 (p. 80-91)

  - Exercises #1-11: https://github.com/prmr/DesignBook/blob/master/exercises/e-chapter4.md

# Coming up

- Next lecture:
  - Chapter 6: Composition