McGill

COMP 303

Lecture 6

Object state

Winter 2025

slides by Jonathan Campbell

Announcements

- Proposal guidelines out.
- Brainstorming meetings.
- Server registration.
- https://www.downtowndougbrown.com/2025/01/the-invalid-68030-instruction-that-accidentally-allowed-the-mac-classic-ii-to-successfully-boot-up/
- https://stratechery.com/2025/deepseek-faq/

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

Object state

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.

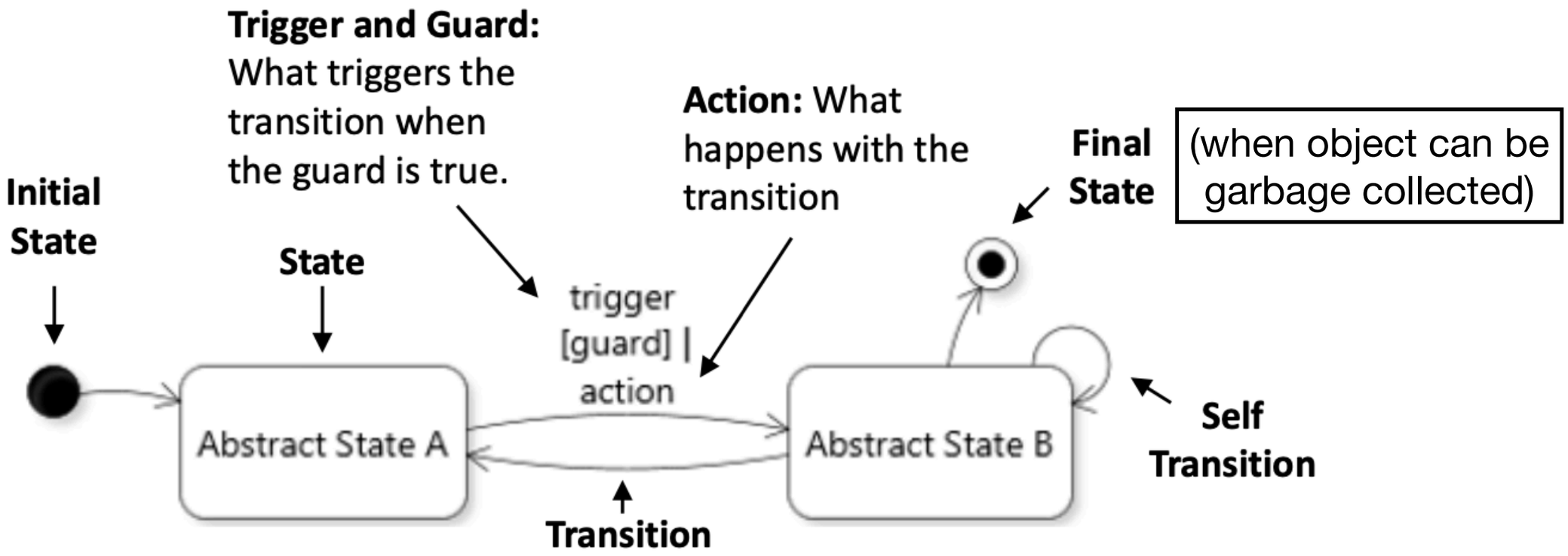
Object state

- We want to come up with meaningful abstract states, e.g., ones that impact how an object is used ("Empty" Deck).
    - "Three Kings" would therefore not be particularly meaningful.
    - States should not describe actions like "draw card" -- they should represent the actual contents of the fields.
- Note: objects with no fields are called stateless objects.

State diagrams

- Model how an object can transition from one abstract state to another via events like method calls.

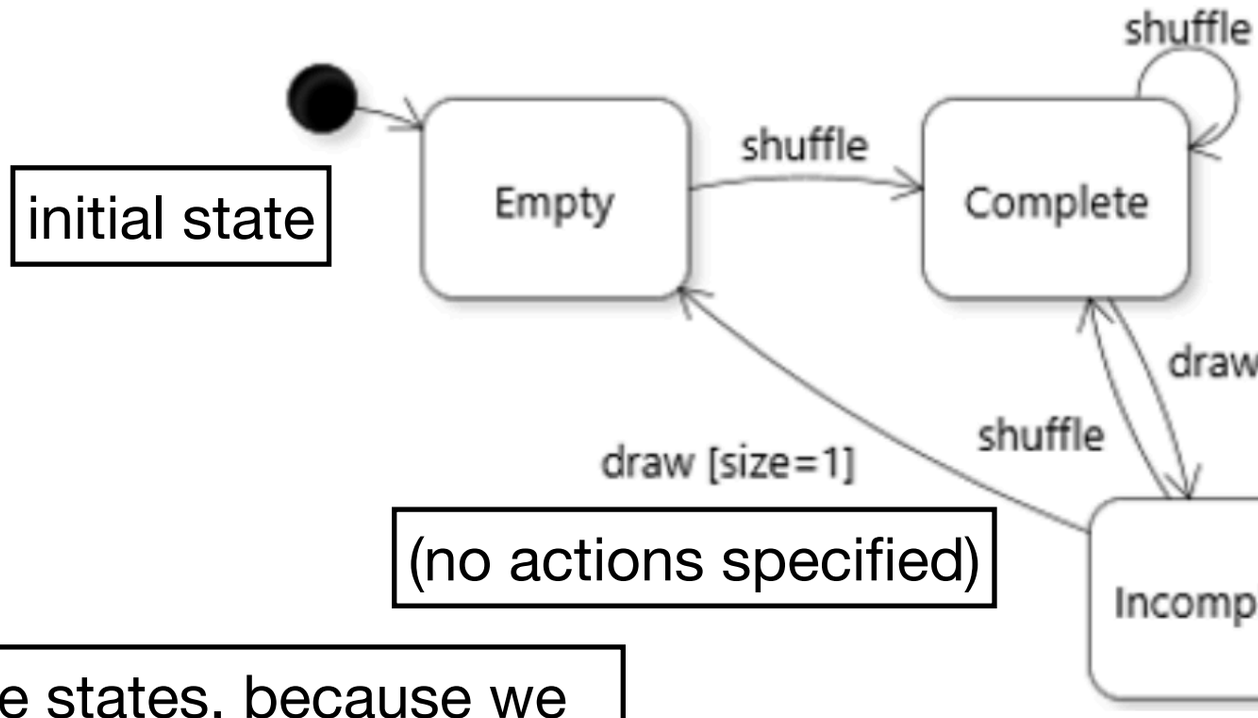
State diagrams



State diagram for a Deck

can only shuffle, but not draw, from an empty deck

self-transition



guard

(no actions specified)

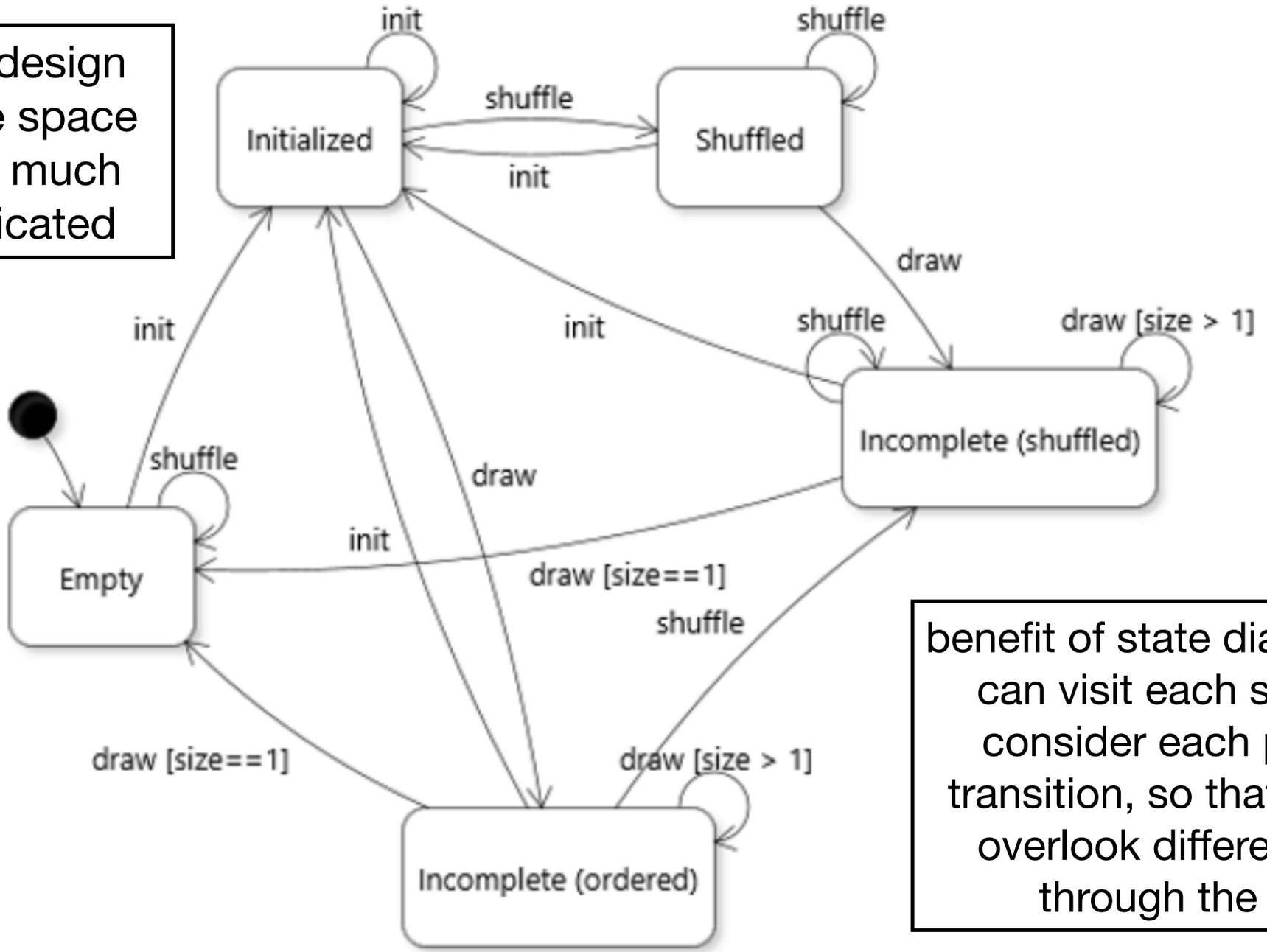
(no final states specified -- object can stop being used in any state)

three states, because we bundled deck initialization + shuffling

State diagrams help us to evaluate the impact of design decisions on the complexity of the state space.

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

Object life cycle

- A state diagram can also be thought of as an object's life cycle:
  - initialization
  - transition between different abstract states (if stateful and mutable)
  - abandonment
  - eventual destruction by garbage collector

Object life cycle

- Objects with a complex life cycle are difficult to use, difficult to test, and their design and implementation are error-prone.

- Design principle: **minimize** the state space of objects to what is absolutely necessary.

- Two relevant anti-patterns: Speculative Generality and Temporary Field.

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.

TEMPORARY FIELD

```
public class Deck {
   private List<Card> aCards = new ArrayList<>();
   private int aSize = 0;
   public int size() {
      return aSize;
   }
   public Card draw() {
      aSize--;
      return aCards.remove(aCards.size());
   }
}
```

have to always keep in mind to update aSize when needed -- extra work and can lead to bugs

Temporary Field

- Information should not be stored in an object unless it uniquely contributes to the intrinsic value represented by the object.

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.

Final fields and variables

```
public class Card {
   private final Rank aRank;
   private final Suit aSuit;
   public Card(Rank pRank, Suit pSuit) {
      aRank = pRank;
      aSuit = pSuit;
   }
}
```

cannot be assigned a second time

Final fields and variables

- Making all fields final is the same as making the class immutable, unless the referenced objects are **mutable**.

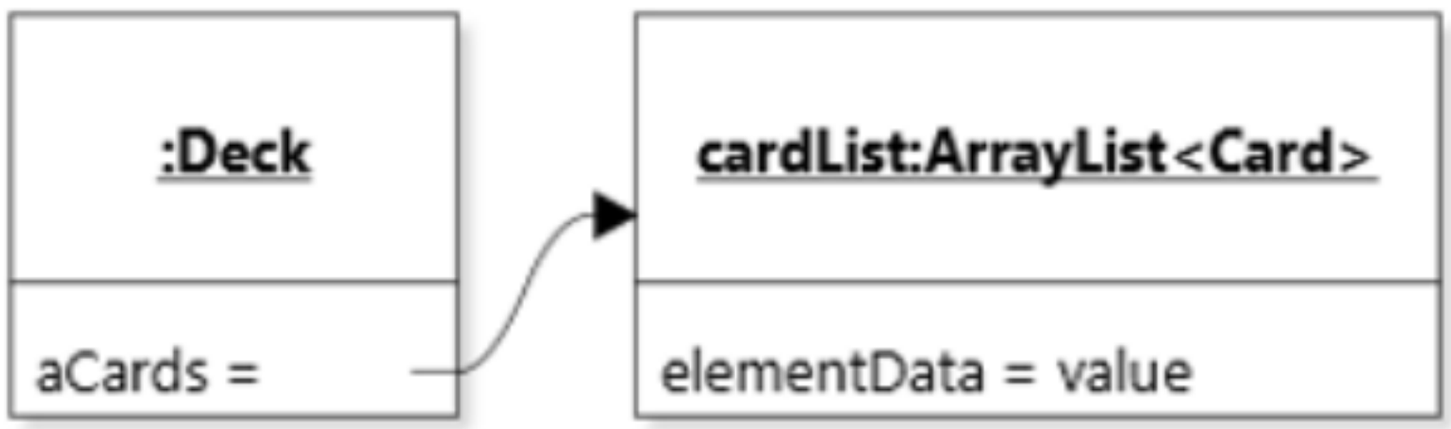
- In that case, although we can't change the reference to the object itself, we can call methods on the object to modify its contents.

Final fields and variables

```
public class Deck {
  private final List<Card> aCards = new ArrayList<>();
}
```

Final fields and variables

If aCards is final, we can't change this arrow after assignment.



But, keep in mind that we can modify the elements inside.

Nullability

```
Card card = null;

System.out.println(card.getRank());

// NullPointerException
```

avoiding null references
can help us avoid bugs

also reduces the state space size

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.

Avoiding null references

```
public Card(Rank pRank, Suit pSuit) {
  // input validation
  if (pRank == null || pSuit == null) {
    throw new IllegalArgumentException();
  }
  aRank = pRank;
  aSuit = pSuit;
}
```

(if constructor is the only place to set these fields)

Avoiding null references

```
/**
* @pre pRank != null && pSuit != null;
*/
public Card(Rank pRank, Suit pSuit) {
   // design by contract
   assert pRank != null && pSuit != null;
   aRank = pRank;
   aSuit = pSuit;
}
```

(if constructor is the only place to set these fields)

Absent values without null

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

Absent values without null

```
public class Card {
   private Rank aRank;
   private Suit aSuit;
   private boolean aIsJoker;

   public boolean isJoker() { return aIsJoker; }
   public Rank getRank() { return aRank; }
   public Suit getSuit() { return aSuit; }
}
```

Bad practice 1: null reference

```
public class Card {
   private Rank aRank;
   private Suit aSuit;

   public Card(Rank pRank, Suit pSuit) {
      aRank = pRank;
      aSuit = pSuit;
   }

   public boolean isJoker() {
      return aRank == null;
   }
}
```

same problem as before: could dereference null accidentally

Bad practice 2: bogus values

```
public class Card {
  private Rank aRank;
  private Suit aSuit;
  private boolean aIsJoker;

  public Card(Rank pRank, Suit pSuit) {
    aRank = pRank;
    aSuit = pSuit;
    aIsJoker = false;
  }
  public Card() {
    aRank = Rank.ACE;
    aSuit = Suit.CLUBS;
    aIsJoker = true;
  }
  public boolean isJoker() {
    return aIsJoker;
  }
}
```

confusing and dangerous!

Bad practice 3: extra enum value

```
public class Card {
  private Rank aRank;
  private Suit aSuit;
  private boolean aIsJoker;

  public Card(Rank pRank, Suit pSuit) {
    aRank = pRank;
    aSuit = pSuit;
    aIsJoker = false;
  }
  public Card() {
    aRank = Rank.INAPPLICABLE;
    aSuit = Suit.INAPPLICABLE;
    aIsJoker = true;
  }
  public boolean isJoker() {
    return aIsJoker;
  }
}
```

```
enum Suit {
  CLUBS,
  DIAMONDS,
  HEARTS,
  SPADES,
  INAPPLICABLE
}
```

better, but an abuse of Enum

also: anything that iterates over Suit/Rank may get messed up

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.

References

- Robillard ch. 4-4.5 (p. 67-80)

Coming up

- Next lecture:
  - More about object state