

Jin L.C. Guo

## M3 (a) – Object State

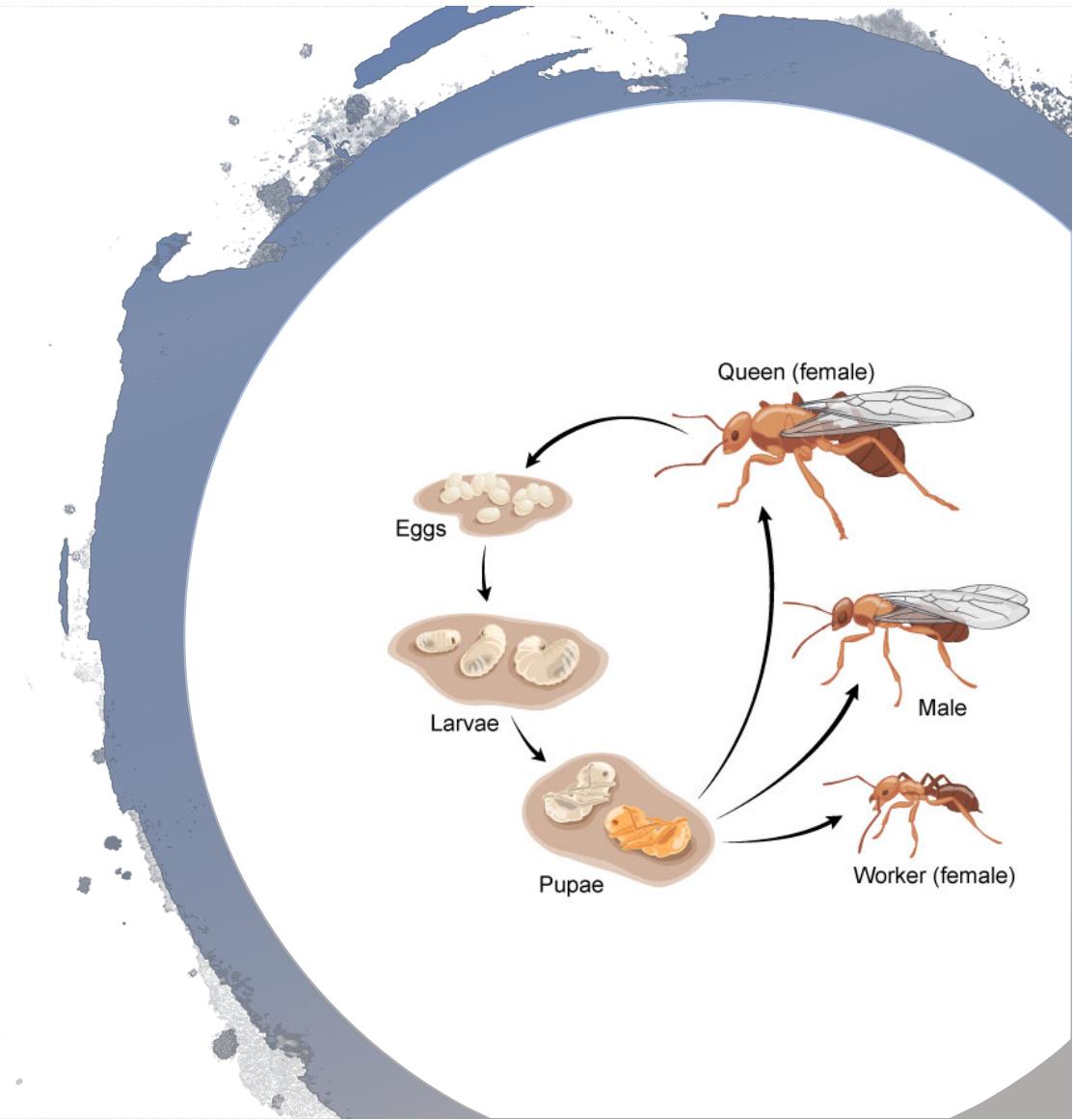
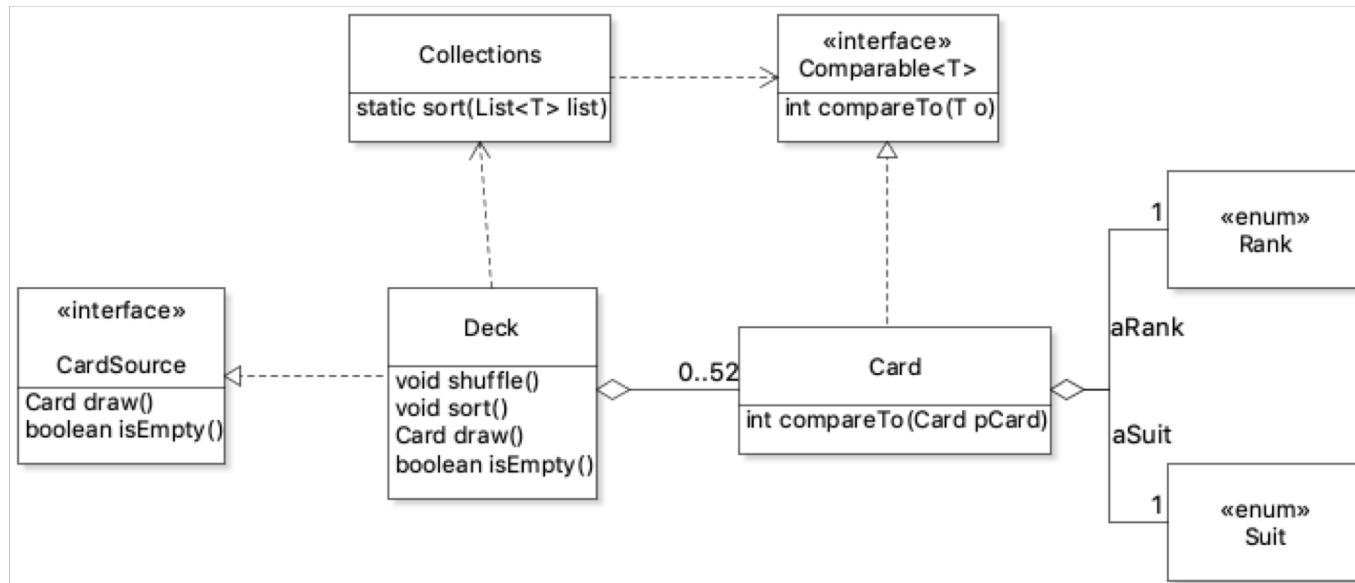


Image Source: <https://askabiologist.asu.edu/individual-life-cycle>

# Objective

- Static vs. Dynamic Perspective on Software
- Concrete vs. Abstract Object State
- Object Life Cycle, State Diagram
- Nullability
- Object Identity
- Object Equality

# Static View of the Software

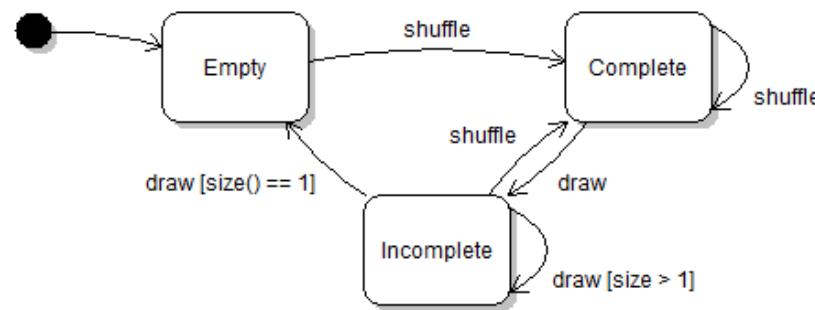


UML Class Diagram

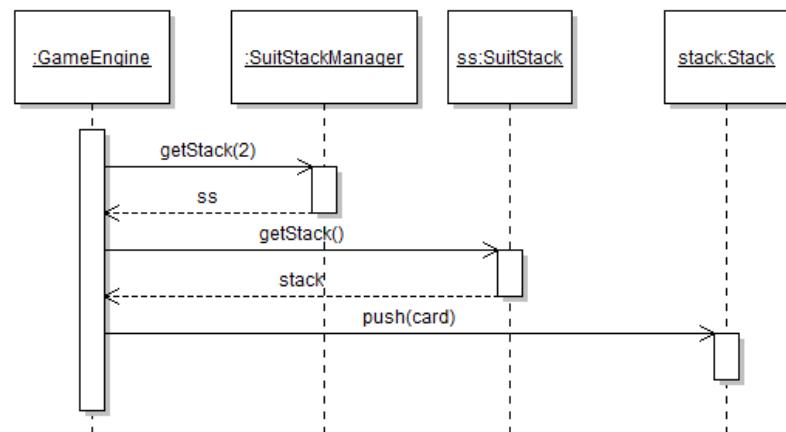
Illustrate how classes are defined

How about the software behaviors run-time?

# Dynamic View of the Software



State Diagram



Sequence Diagram

How about the software behaviors run-time?

# Object at Run-time

```
public final class Card
{
    private final Rank aRank;           {ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN,
    private final Suit aSuit;          {EIGHT, NINE, TEN, JACK, QUEEN, KING}
}
```

{CLUBS, DIAMONDS, HEARTS, SPADES}

13x4 possible state

Concrete state of the object

# Object at Run-time

Abstract State is needed

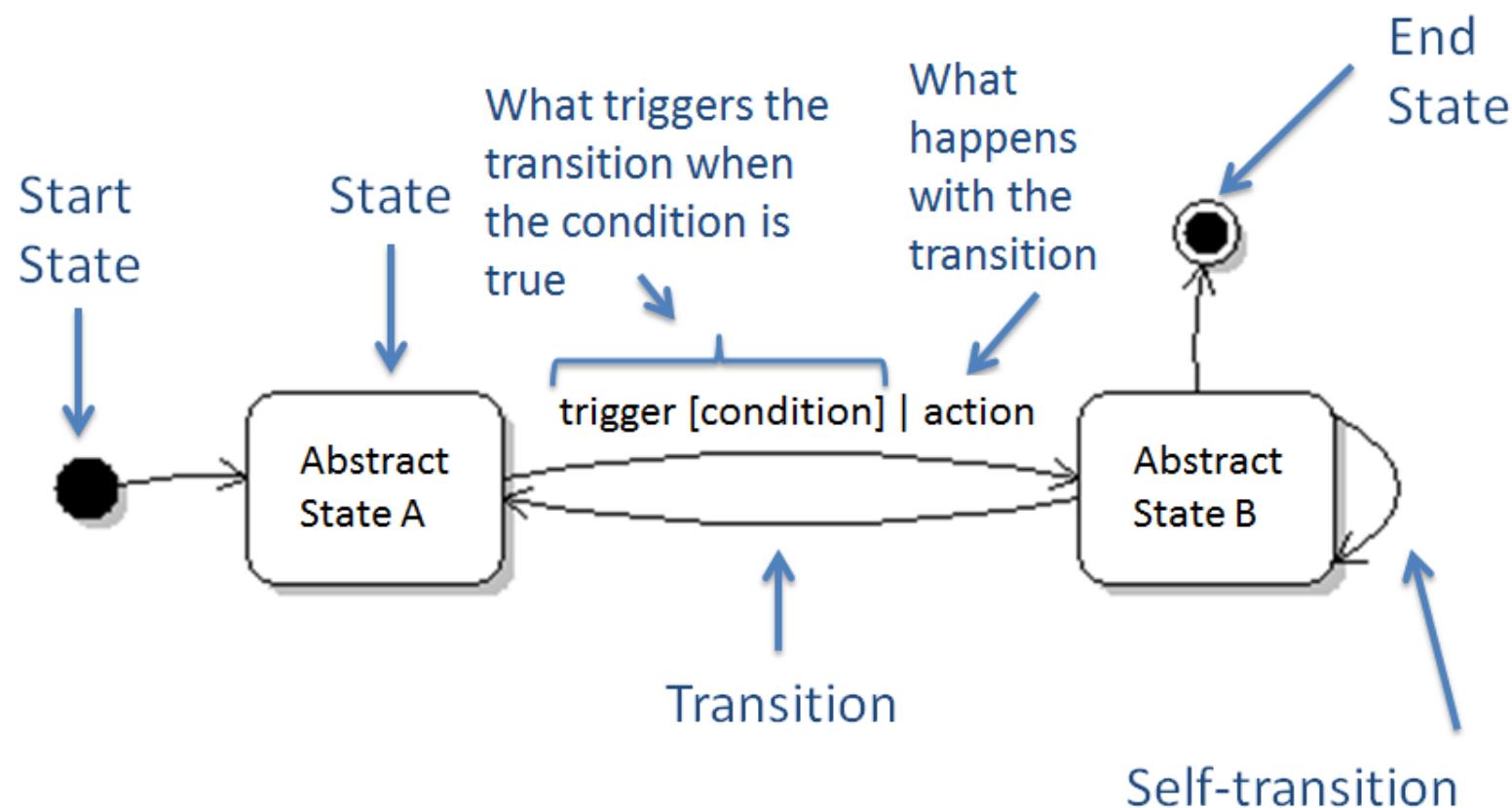
```
public class Word {  
    // Representation of a word in its original form  
    // as in one sentence.  
    final private String orignialForm;  
}
```

Concrete state of the object  
 $(2^{31} - 1) \times 2^{16}$  !

# State Diagram

- Abstract States
- Transitions between states

# State Diagram



# State Diagram for Deck Object

- Demo of using JetUML for two Deck Designs

# Design Constructor

- A constructor should fully initialize the object
  - The class invariant should hold
  - Shouldn't need to call other methods to “finish” initialization

# Design Field

- Has a value that retains meaning throughout the object's life
- Its state must persist between public method invocations

# General Principle

- Minimize the state space of object to what is absolutely necessary
  - It's impossible to put the object in an invalid or useless state
  - There's no unnecessary state information

# Nullability (absence of value)

```
Card card = null;
```

A viable is temporarily un-initialized and will be initialized in a different state.

A viable is incorrectly initialized. The code of initiation is not executed properly.

As a flag that represents the absence of a useful value

Special use.

```
Card.Rank rank = card.getRank();
```

Avoid *null* values when designing classes!

# How to avoid Null

- Through Contract

Make Null invalid for a variable

```
/**  
 * @param pRank The index of the rank in RANKS  
 * @param pSuit The index of the suit in SUITS  
 * @pre pRank != null && pSuit != null  
 */  
public Card(Rank pRank, Suit pSuit)  
{  
    assert pRank != null && pSuit != null;  
    aRank = pRank;  
    aSuit = pSuit;  
}
```

# How to avoid Null

- Sometimes it's necessary to model absence of value

# Activity

- Discuss your design of the extension of class Card where one instance can also represent a "Joker". (M1-Exercise, #2)

Note: Joker is special card with no rank and no suit.



Image source: [https://upload.wikimedia.org/wikipedia/commons/6/6f/Joker\\_Card\\_Image.jpg](https://upload.wikimedia.org/wikipedia/commons/6/6f/Joker_Card_Image.jpg)

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

Arbitrary (valid) value for rank and suit?

Add special value for Rank and Suit enums?

## java.util.Optional<T>

- A container object which may or may not contain a non-null value.
- If a value is present, isPresent() will return true and get() will return the value.

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

```
public Card(Rank pRank, Suit pSuit)
{
    assert pRank != null && pSuit != null;
    aRank = Optional.of(pRank);
    aSuit = Optional.of(pSuit);
}

public Card()
{
    aIsJoker = true;
    aRank = Optional.empty();
    aSuit = Optional.empty();
}
```

# What about getter methods?

- Return Optional<T> types
- Up-wrap Optional and return T

# Object Identity

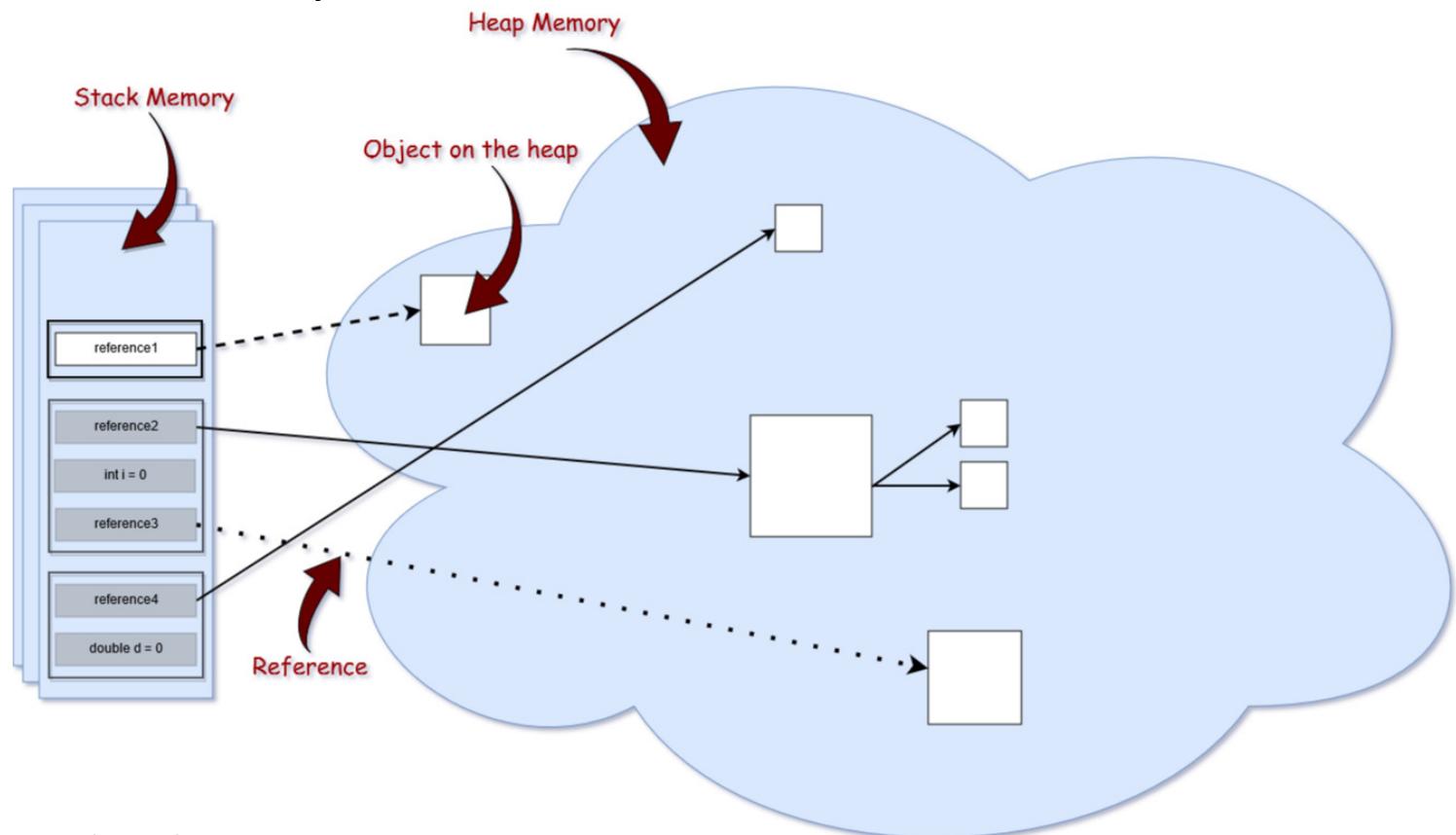


Image Source: <https://dzone.com/articles/java-memory-management>

# Object Identity

The screenshot shows a Java IDE interface. On the left is the code editor with the following Java code:

```
public static void main(String[] args)
{
    Card card1 = new Card( Rank.ACE, Suit.CLUBS );
    Card card2 = new Card( Rank.ACE, Suit.CLUBS );
    System.out.println(card1 == card2);
}
```

On the right is the "Variables" window, which displays the current state of variables:

Name	Value
args	String[0] (id=16)
card1	Card (id=21)
card2	Card (id=22)

The variable "card2" has its value highlighted with a red box.

# True or False?

```
Card card1 = new Card(Card.Rank.FOUR, Card.Suit.CLUBS);
Card card2 = new Card(Card.Rank.FOUR, Card.Suit.CLUBS);
Card card3 = card1;

System.out.println(card1 == card2);
System.out.println(card1 == card3);
System.out.println(card1.equals(card2));
System.out.println(card1.equals(card3));
```

# Object Equality

```
Card card1 = new Card(Card.Rank.FOUR, Card.Suit.CLUBS);
Card card2 = new Card(Card.Rank.FOUR, Card.Suit.CLUBS);
Card card3 = card1;
```

```
System.out.println(card1 == card2); Reference equality
System.out.println(card1 == card3);
System.out.println(card1.equals(card2));
System.out.println(card1.equals(card3));
```

Variables refer to (point to) the same object in the memory

# Object.equals method

```
public class Object {  
    public boolean equals(Object o) {  
        return this == o; // reference equality  
    }  
}
```

Implements an equivalence relation on non-null object references.

*Reflexive:*  $x.equals(x) == true$

*Symmetric:*  $x.equals(y) \Leftrightarrow y.equals(x)$

*Transitive:*  $x.equals(y) \wedge y.equals(z) \Leftrightarrow x.equals(z)$

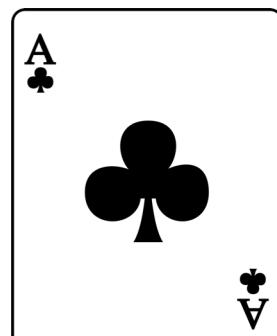
*Consistent:*  $x.equals(x) == x.equals(x)$

For non-null reference value  $x$      $x.equals(null) == false$

# Object.equals method

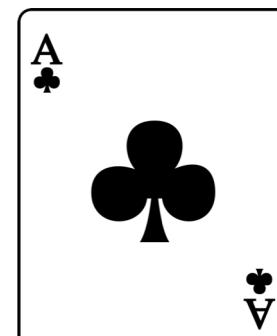
- The most discriminating possible equivalence relation on objects

What about when logical equality is needed?



card1 (id = 21)

equals



card2 (id = 22)

# Override equals method

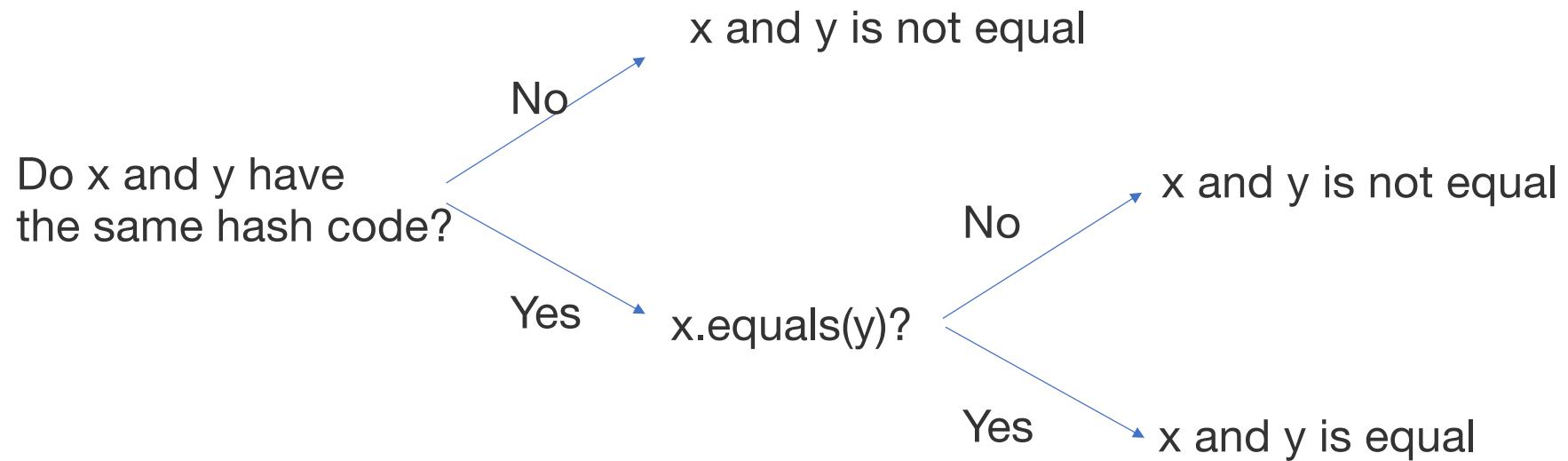
```
@Override  
public boolean equals(Object obj) {  
    if (this == obj) return true;  
    if (obj == null) return false;  
  
    if (getClass() != obj.getClass())  
        return false;  
  
    Card other = (Card) obj;  
  
    return aIsJoker == other.aIsJoker  
        && aRank.equals(other.aRank)  
        && aSuit.equals(other.aSuit)  
}
```

## True or False (after overriding equals)?

```
Card card1 = new Card(Card.Rank.FOUR, Card.Suit.CLUBS);
Card card2 = new Card(Card.Rank.FOUR, Card.Suit.CLUBS);
Card card3 = card1;

System.out.println(card1 == card2);
System.out.println(card1 == card3);
System.out.println(card1.equals(card2));
System.out.println(card1.equals(card3));
```

# Prefiltering for equality



# Object.hashCode method

**public int hashCode()**

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by [HashMap](#).

*Self-Consistent:* `o.hashCode() == o.hashCode()`

*Consistent with Equals:*

`x.equals(y) => x.hashCode() == y.hashCode()`

# Override hashCode() method

```
@Override  
public int hashCode() {  
    return 1;  
}  
  
@Override  
public int hashCode() {  
    return Boolean.hashCode(aIsJoker)  
        + aRank.hashCode()  
        + aSuit.hashCode();  
}
```

# Equality during Inheritance

```
public class CardWithDesign extends Card {  
    public enum Design{ CLASSIC, ARTISTIC, FUN}  
  
    Design aStyle;  
  
    public CardWithDesign(Rank pRank, Suit pSuit, Design pStyle) {  
        super(pRank, pSuit);  
        this.aStyle = pStyle;  
    }  
    public CardWithDesign(Design pStyle) {  
        super();  
        this.aStyle = pStyle;  
    }  
}
```

```
Card card1 = new CardWithDesign(Card.Rank.FOUR, Card.Suit.CLUBS,  
    CardWithDesign.Design.ARTISTIC);  
Card card2 = new CardWithDesign(Card.Rank.FOUR, Card.Suit.CLUBS,  
    CardWithDesign.Design.CLASSIC);  
  
System.out.println(card1.equals(card2));
```

```
Card card3 = new Card(Card.Rank.FOUR, Card.Suit.CLUBS);  
System.out.println(card1.equals(card3));  
System.out.println(card3.equals(card1)); Violate Symmetric property
```

```
System.out.println(card2.equals(card3)); Violate Transitive property
```

# Solution?

Make the comparison between supertype and subtype return false

Favor composition over inheritance (More on Module 5)