

Jin L.C. Guo

M3 (a) – Object State

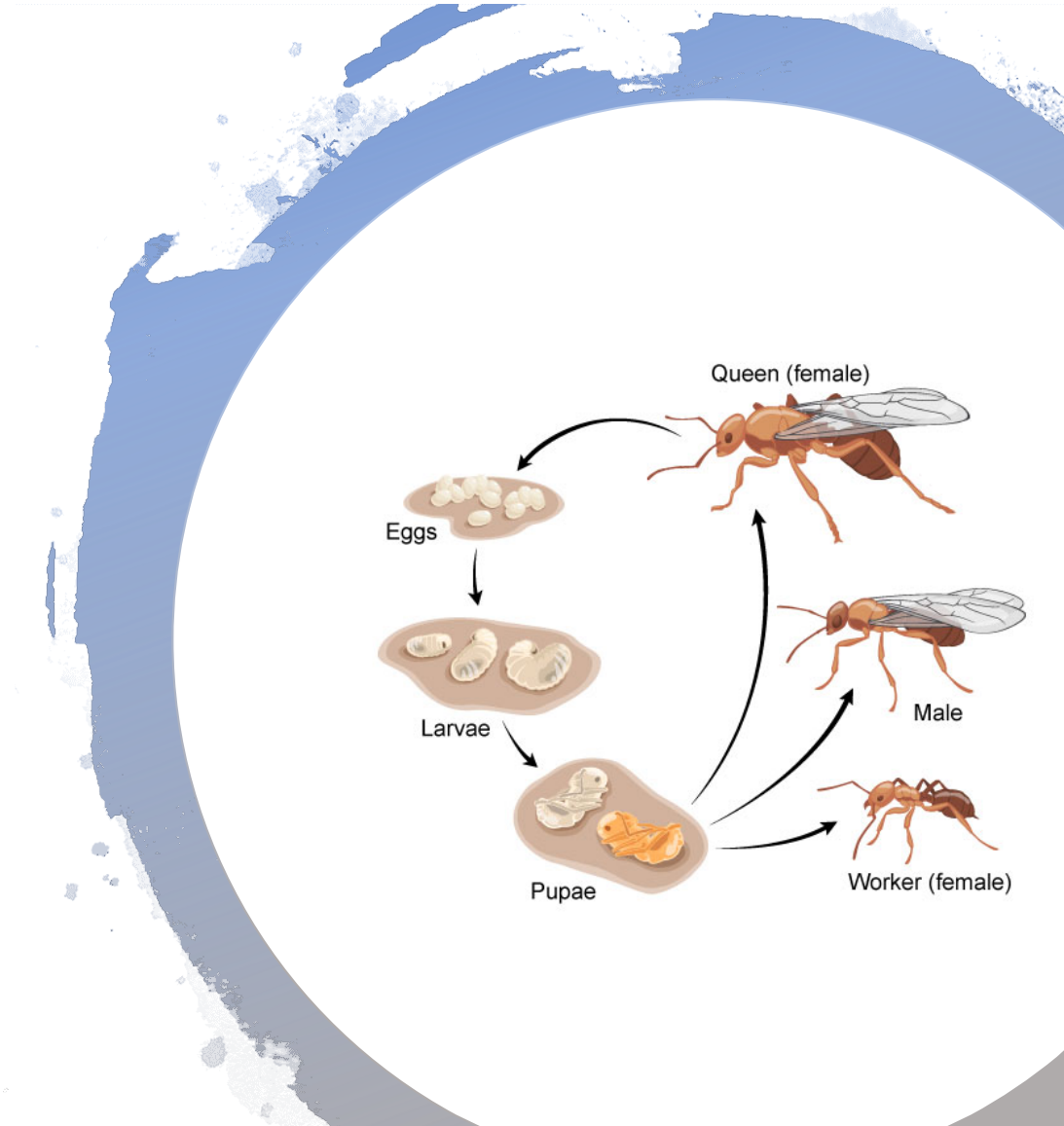


Image Source: <https://askabiologist.asu.edu/individual-life-cycle>

Objective

- Programming mechanism:

Null references, optional types

- Concepts and Principles:

Object life cycle, object identity and equality

- Design techniques:

State Diagram

Object at Run-time

```
public final class Card
```

```
{
```

```
    private final Rank aRank;
```

```
    private final Suit aSuit;
```

```
}
```

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

```
{CLUBS, DIAMONDS, HEARTS, SPADES}
```

13x4 possible state

Object at Run-time

Abstract State is needed

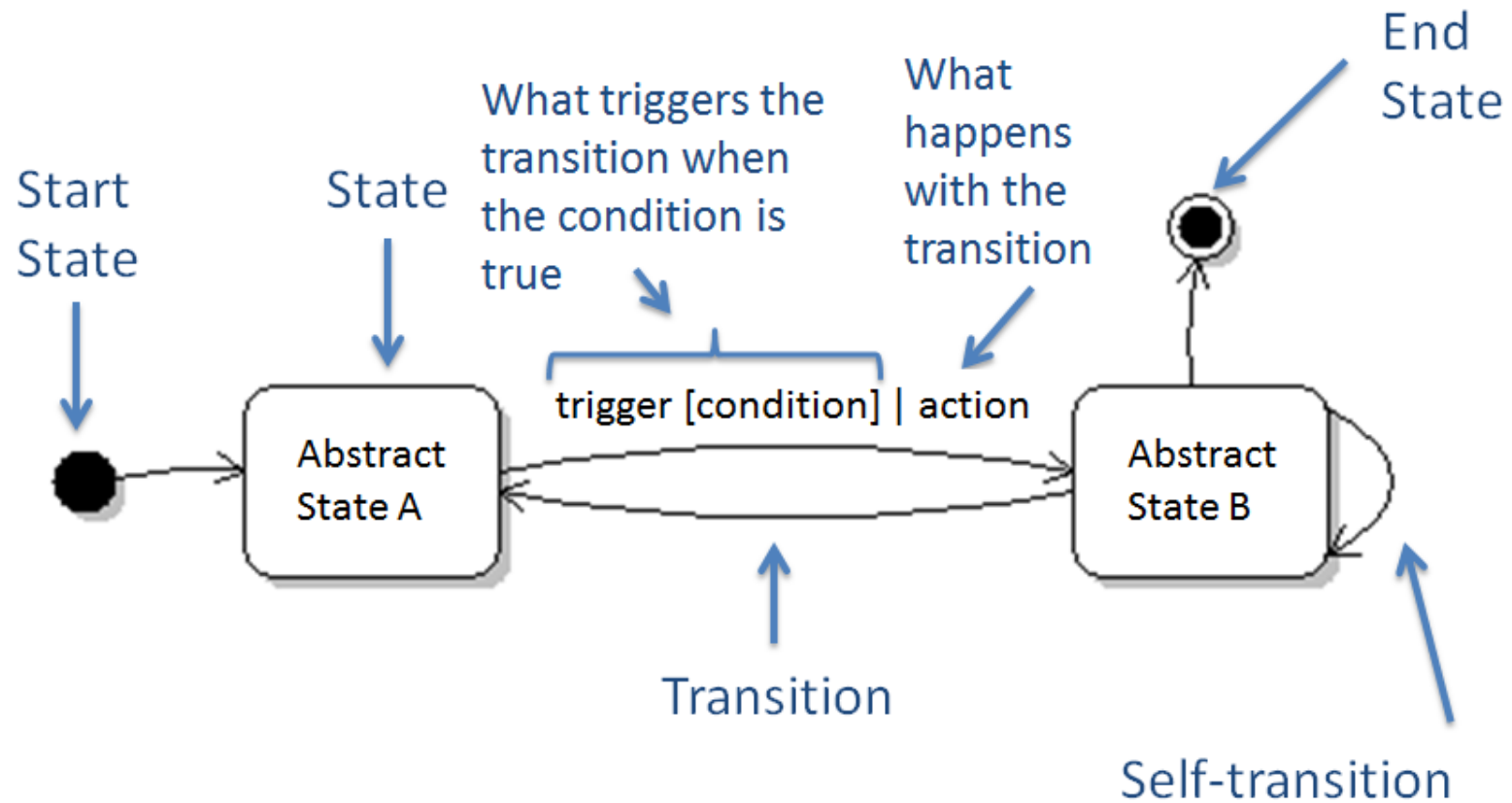
```
public class Student {  
    // Representation of a word in its original form  
    // as in one sentence.  
    final private String firstName;  
}
```

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

State Diagram

- Abstract States
- Transitions between states

State Diagram



Activity 1: Sketch the state diagram of **Course**

```
public class Course {  
    private String      aID; // e.g. "COMP 303"  
    private boolean     aIsActive;  
    private int         aCap;  
    private List<Student> aEnrollment;  
    private CourseSchedule aSchedule;  
}
```

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

Objective

- Programming mechanism:

Null references, optional types

- Concepts and Principles:

Object life cycle, object identity, equality, and uniqueness

- Design techniques:

State Diagram

Objective

- Programming mechanism:

Null references, optional types

- Concepts and Principles:

Object life cycle, object identity and equality

- Design techniques:

State Diagram

Nullability (absence of value)

```
Card card = null;
```

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

A variable is incorrectly initialized. The code of initialization 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!

Avoid *null* values when designing classes?

```
public class Course {
```

```
    private String aID;  
    private boolean aIsActive;  
    private int aCap;  
    private List<Student> aEnrollment;  
    private CourseSchedule aSchedule;
```

What about Schedule?

It might be a valid state when the class is created but not scheduled.

```
    public Course(String pID, int pCap) {  
        aID = pID;  
        aCap = pCap;  
        aEnrollment = new ArrayList<>();  
        aIsActive = false;  
    }
```

Avoid *null* values when designing classes?

- Sometimes it's necessary to model absence of value

Activity 2:

- Discuss your design of the extension of class Card where one instance can also represent a "Joker". (Textbook Chapter 2 - Exercise #4)

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

- How did you handle the fields of Rank and Suit for "Joker"?



Image source: 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

Go back to the **Course** class

```
public class Course {  
    .....  
    public Course(String pID, int pCap) {  
        aID = pID;  
        aCap = pCap;  
        aEnrollment = new ArrayList<>();  
        aIsActive = false;  
        aSchedule = Optional.empty();  
    }  
  
    public void setSchedule(CourseSchedule pSchedule) {  
        aSchedule = Optional.of(new CourseSchedule(pSchedule));  
    }  
  
    public Optional<CourseSchedule> getSchedule(){  
        return aSchedule;  
    }  
  
    private static void printSchedule(Course p  
        if(pCourse.getSchedule().isPresent())  
            CourseSchedule schedule = pCourse.  
            System.out.println(schedule);  
        } else {  
            System.out.println("Schedule unava  
        }  
    }
```

Client code of the **Course** class

```
private static void printSchedule(Course pCourse) {  
    if(pCourse.getSchedule().isPresent()) {  
        CourseSchedule schedule = pCourse.getSchedule().get();  
        System.out.println(schedule);  
    } else {  
        System.out.println("Schedule unavailable.");  
    }  
}
```

Objective

- Programming mechanism:

Null references, optional types

- Concepts and Principles:

Object life cycle, object identity and equality

- Design techniques:

State Diagram

Object Identity

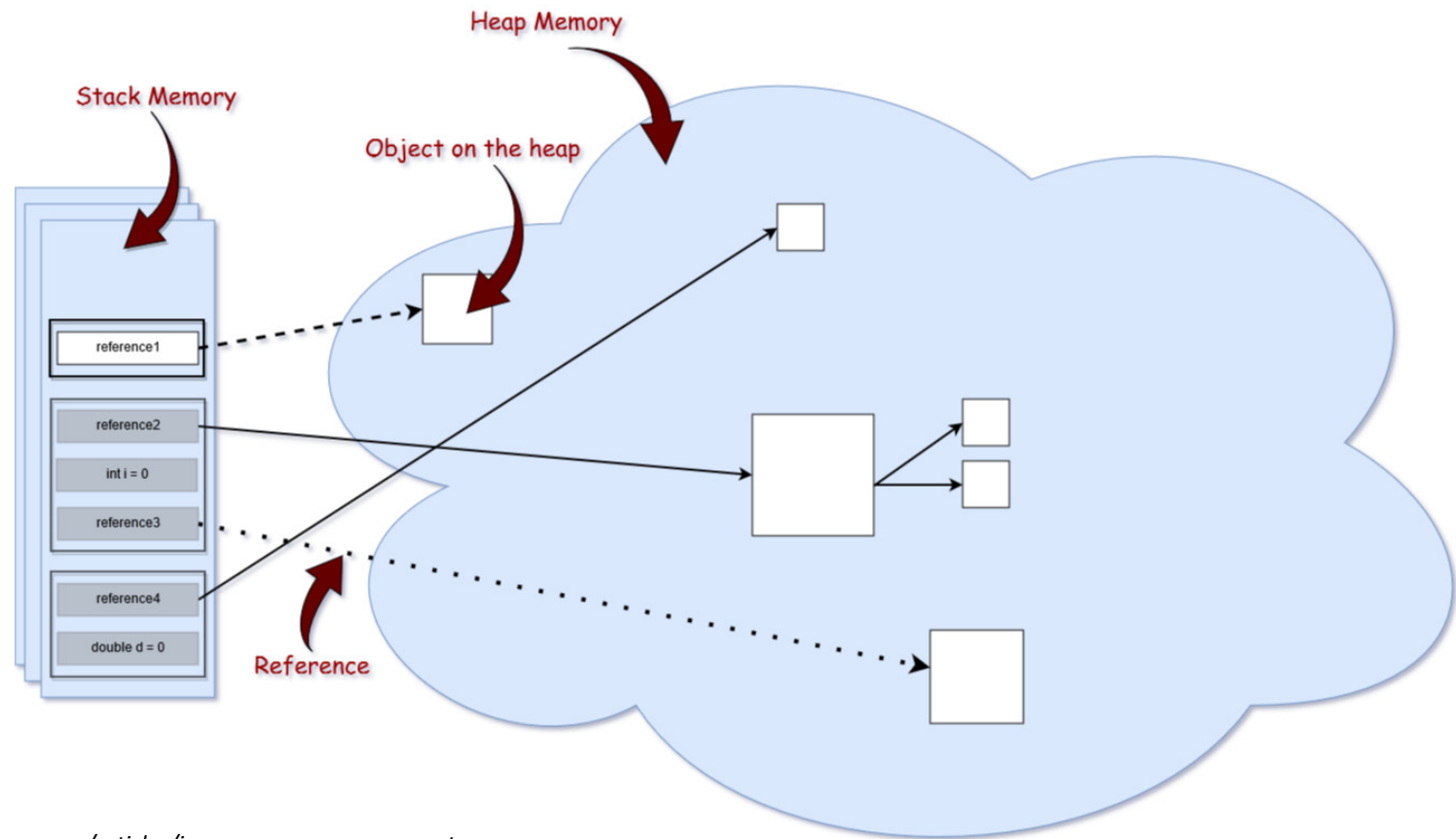


Image Source: <https://dzone.com/articles/java-memory-management>

Object Identity

```
private static CourseSchedule createSchedule() {  
    DayOfWeek[] pDayOfWeek = new DayOfWeek[2];  
    pDayOfWeek[0] = DayOfWeek.WEDNESDAY;  
    pDayOfWeek[1] = DayOfWeek.FRIDAY;  
    LocalTime startTime = LocalTime.of( hour: 14, minute: 35, second: 00);  
    LocalTime endTime = LocalTime.of( hour: 15, minute: 55, second: 00);  
  
    CourseSchedule schedule = new CourseSchedule(new Semester(Semester.Term.Fall, pYear: 2020), pDayOfWeek,  
        startTime, endTime);  
    return schedule;  
}
```

Variables

- + ▶ `pDayOfWeek` = {DayOfWeek[2]@497}
- ▶ `startTime` = {LocalTime@498} "14:35"
- ▶ `endTime` = {LocalTime@499} "15:55"
- ▼ `schedule` = {CourseSchedule@506} "Schedule: Fall-2020, [WEDNESDAY, FRIDAY], from 14:35 to 15:55"
 - ▶ `aSemester` = {Semester@507} "Fall-2020"
 - ▶ `aDayOfWeek` = {DayOfWeek[2]@519}
 - ▶ `aStartTime` = {LocalTime@498} "14:35"
 - ▶ `aEndTime` = {LocalTime@499} "15:55"

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

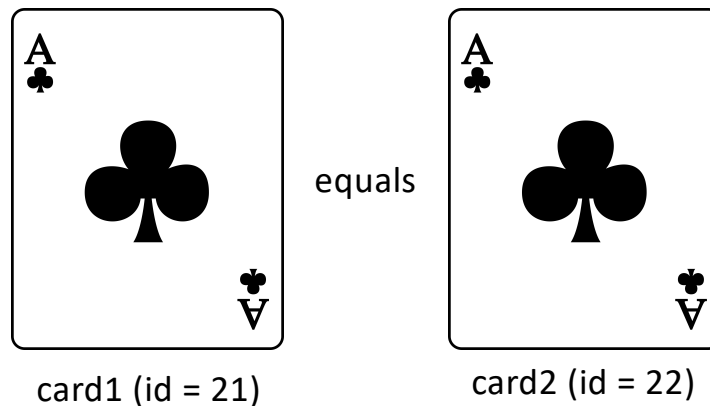
Reference equality

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

Reference Equality

- The most discriminating possible equivalence relation on objects

What about when logical equality is needed?



Logical equality: Using `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: $x.equals(x) == \text{true}$

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 $x.equals(\text{null}) == \text{false}$

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));
```

Also override **Object.hashCode** method

```
public int hashCode()
```

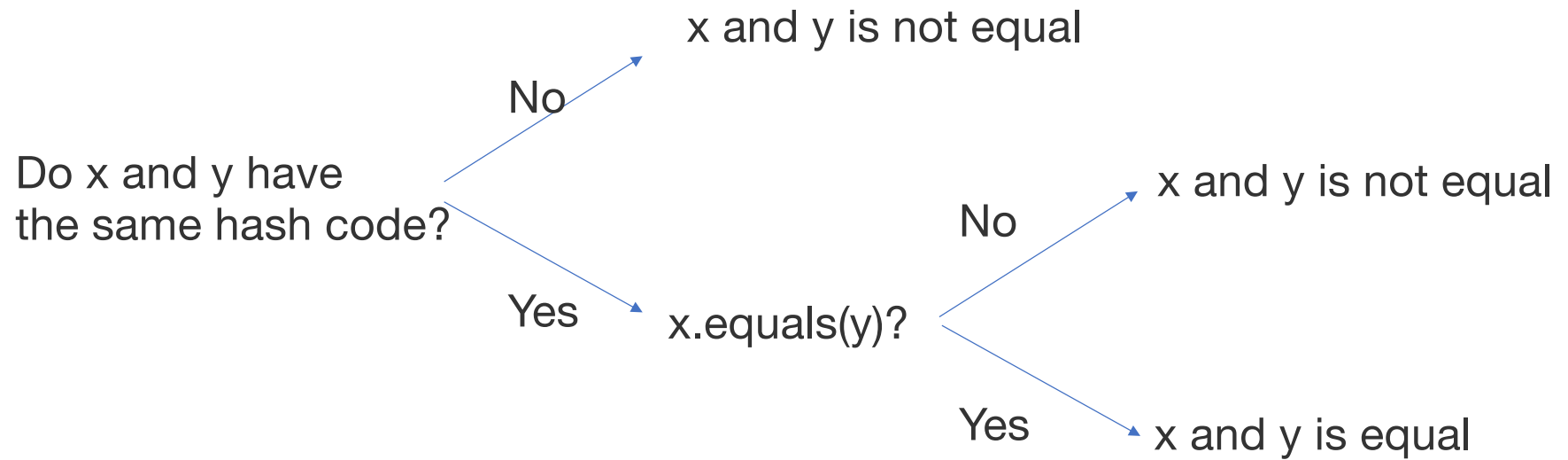
Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by [HashMap](#).

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

Consistent with Equals:

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

Prefiltering for equality



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 during Module-Composition)