Jin L.C. Guo

M3 (b) – Object State

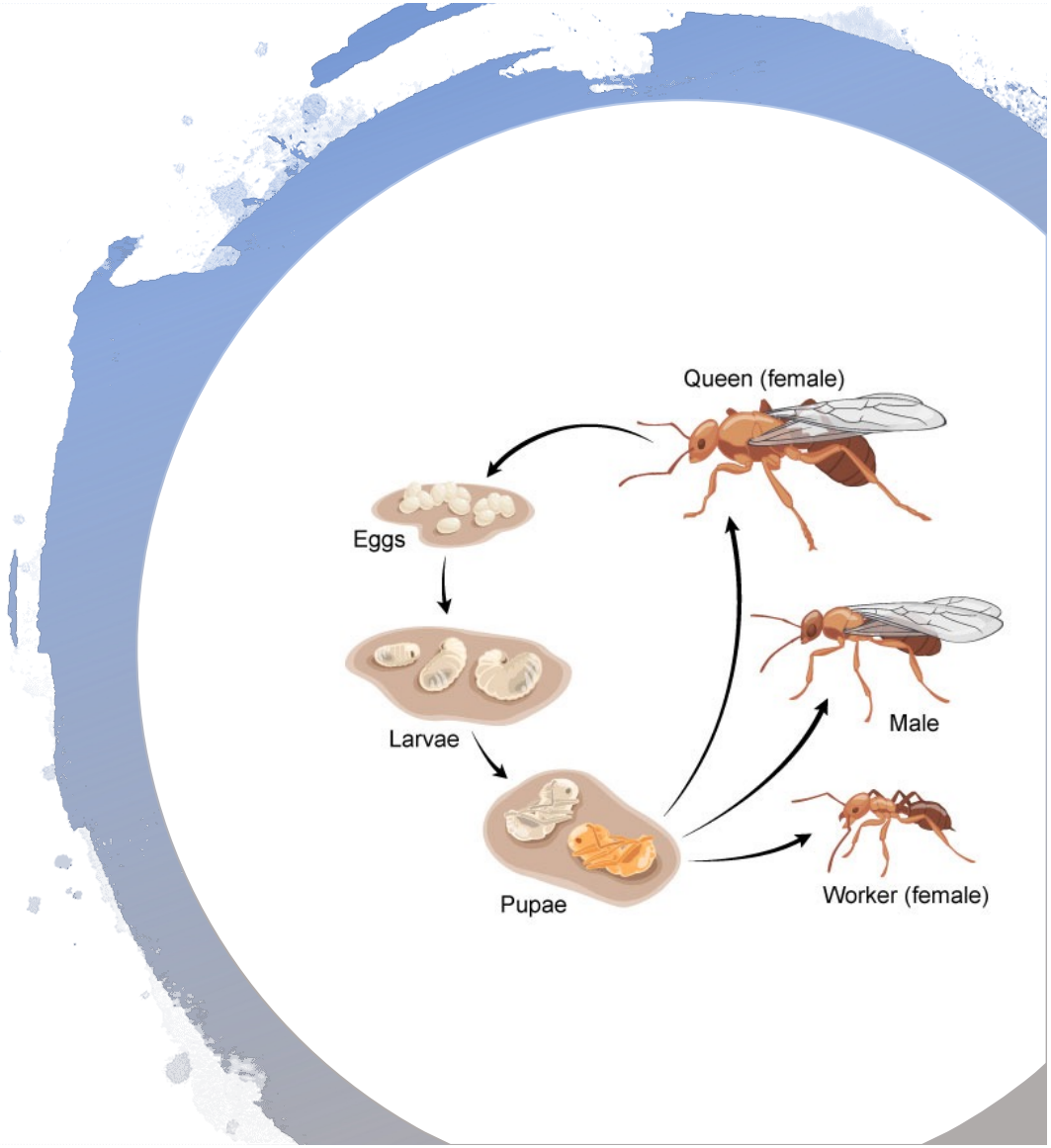


Image Source: https://askabiologist.asu.edu/individual-life-cycle

Recap - Objective

- Programming mechanism:

Null references, optional types

- Concepts and Principles:

Object life cycle

- Design techniques:

State Diagram

- Design Patterns and Antipatterns:

NULL OBJECT

Objective

- Concepts and Principles:

Object identity and equality, Object uniqueness

- Design Patterns and Antipatterns:

FLYWEIGHT, SINGLETON

Object Identity

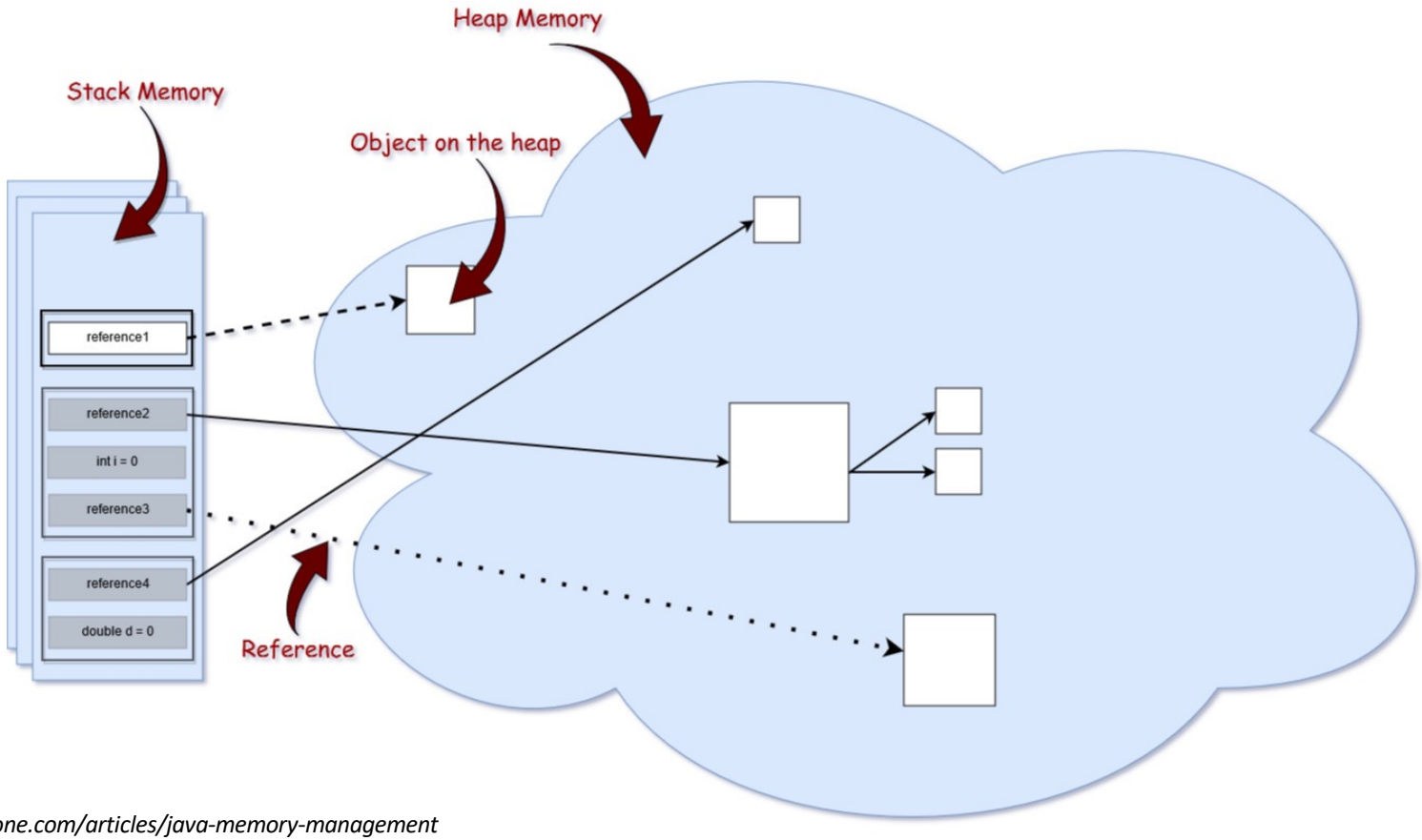


Image Source: https://dzone.com/articles/java-memory-management

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));
```

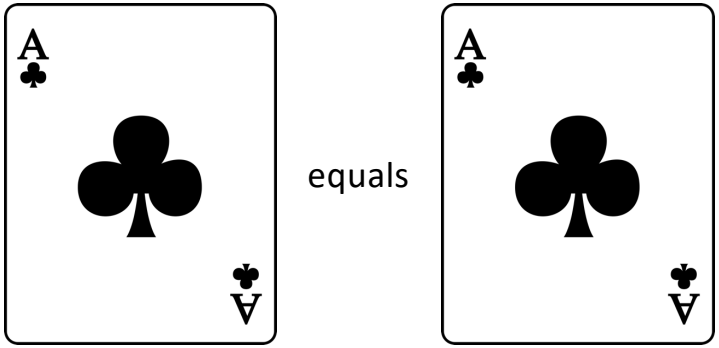
Reference equality

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) == true

Symmetric: x.equals(y) ⇔ y.equals(x)

Transitive: x.equals(y) ∧ y.equals(z) ⇔ x.equals(z)

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

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

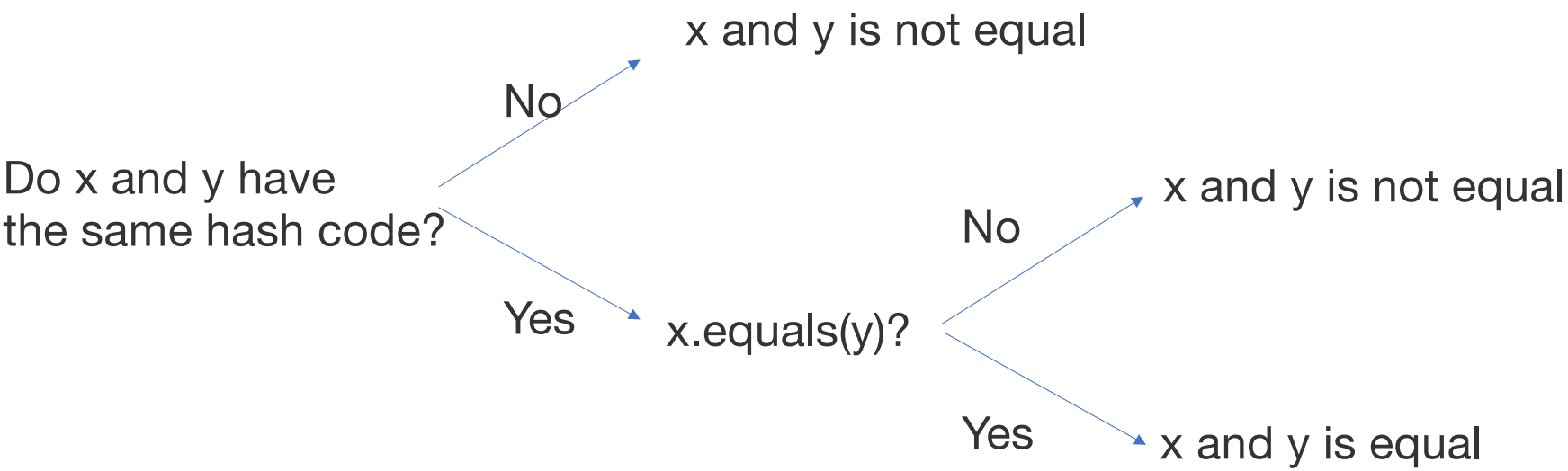
Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by HashMap.

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

Consistent with Equals:

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

Prefiltering for equality



Override hashCode() method

```java
@Override
public int hashCode() {
    return 1;
}
```

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

Activity: design the comparison methods for `CardWithDesign` and `Card` classes

```
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;
    }
}
```

Equality during Inheritance

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

Violate Transitive property

Solution?

Make the comparison between supertype and subtype return false

Favor composition over inheritance (More during Module-Composition)

Objective

- Concepts and Principles:

Object identity and equality, Object uniqueness

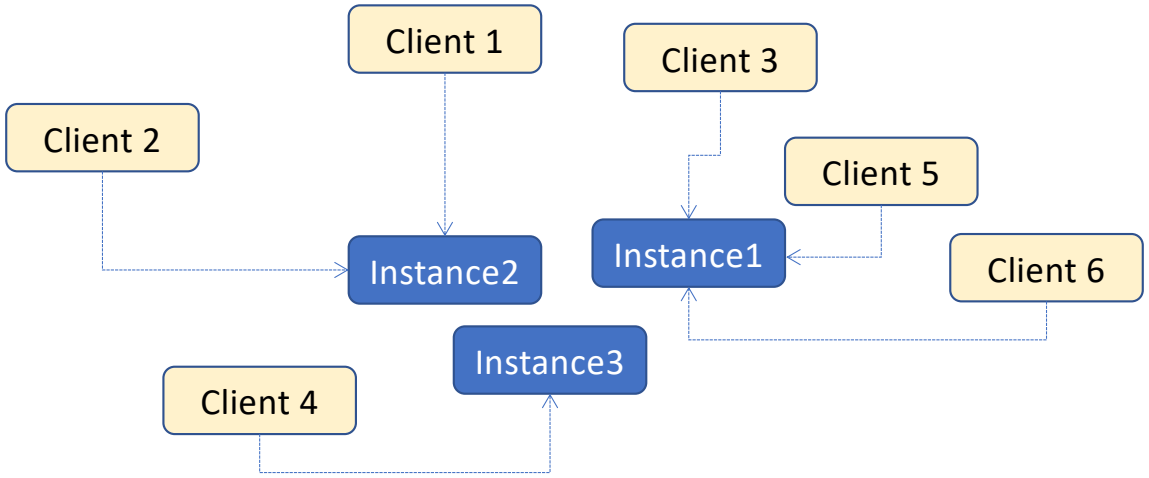
- Design Patterns and Antipatterns:

FLYWEIGHT, SINGLETON

Object Uniqueness

Do we even need to have more than one object
to represent the cards with the same rank and suit?

Flyweight Design Pattern



Immutable objects are safe to shared

Flyweight Design Pattern

Application uses a large number of objects;

Storage of those objects is high;

Objects are immutable;

Application doesn’t depend on the object identity.

Flyweight Design Pattern

How to control the creation of instances?	Change access to constructor
How to store the flyweight instances?	Choose a data structure and its location
How to supply the flyweight instances?	Use a static factory method

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

```
public class CardFactory
{




   ……

   public static Card getCard(Card.Rank rank, Card.Suit suit) {
      assert rank!=null && suit!=null;
      return CARDS[suit.ordinal()][rank.ordinal()];
   }


}
```

Flyweight in Java

- java.lang.Integer#valueOf(int)

Returns an Integer instance representing the specified int value.

If a new Integer instance is not required, this method should generally be used in preference to the constructor Integer(int), as this method is likely to yield significantly better space and time performance by caching frequently requested values.

This method will always cache values in the range -128 to 127, inclusive, and may cache other values outside of this range.

```
public static Integer valueOf(int i) {
     if (i >= IntegerCache.low && i <= IntegerCache.high)
         return IntegerCache.cache[i + (-IntegerCache.low)];
     return new Integer(i);
 }

 private static class IntegerCache {
     static final int low = -128;
     static final int high;
     static final Integer cache[];

     static {
        /* … initialize cache … */
     }

     private IntegerCache() {}
 }
```

Flyweight that enables sharing,
but does not enforce it.

Flyweight in Java

- java.lang.Integer#valueOf(int)

- Similarly on Boolean, Byte, Character, Short, Long and BigDecimal

Flyweight Design Pattern

- When to instantiate flyweight object?
    - Pre-instantiate
    - Create instance upon request

Objective

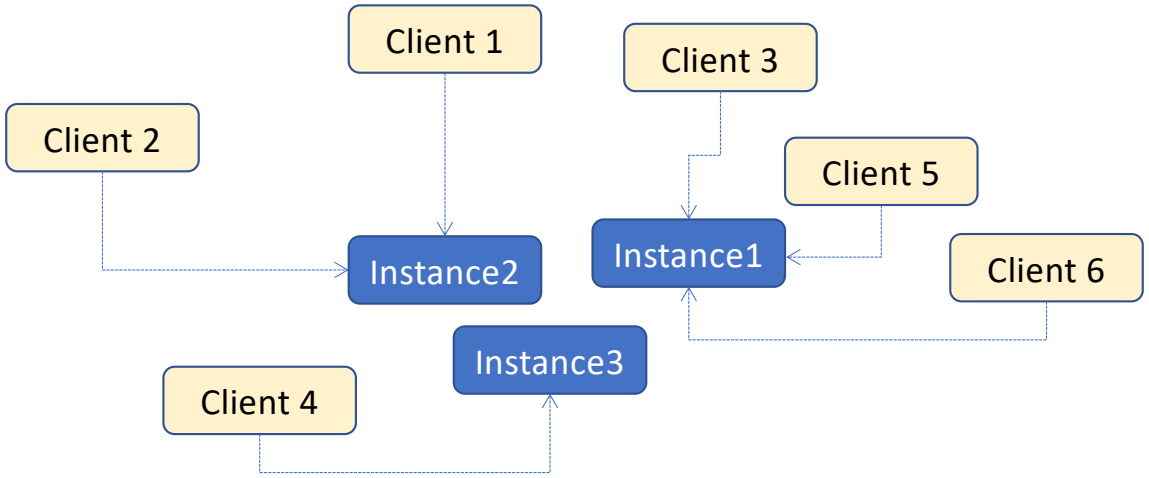
- Concepts and Principles:

Object identity and equality, Object uniqueness

- Design Patterns and Antipatterns:

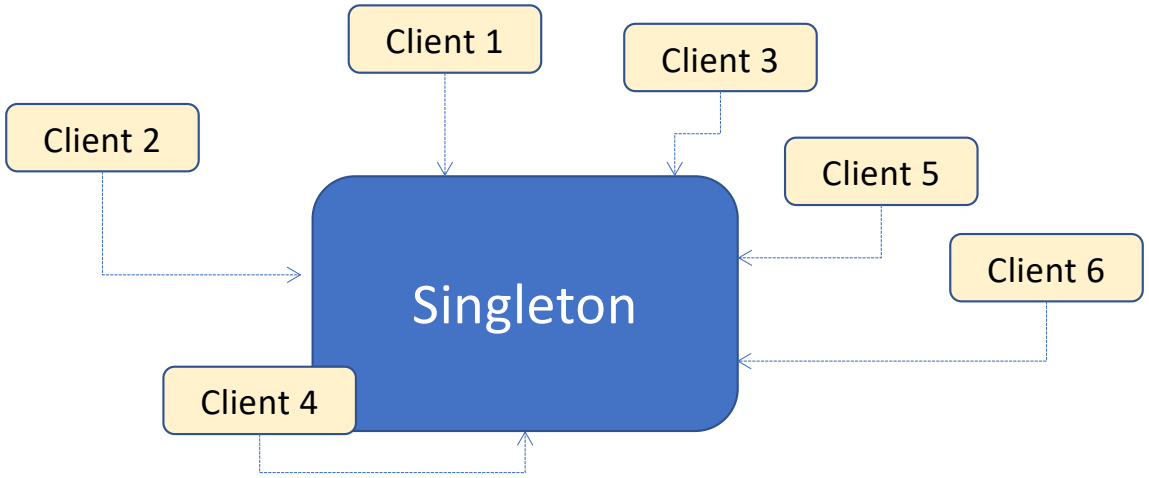
FLYWEIGHT, SINGLETON

Flyweight Design Pattern



Singleton Design Pattern

- Guarantee there is *a single instance* of a class



Singleton Design Pattern

How to control the creation of instance?	Change access to constructor
How to store the single instance?	A static final variable holding the reference to the instance
How to supply the single instances?	public static method

Singleton in Java

- java.lang.Runtime

Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running.

The current runtime can be obtained from the getRuntime method.

An application cannot create its own instance of this class.

```
public class Runtime {
    private static Runtime currentRuntime = new Runtime();


    /**
    * Returns the runtime object associated with the current Java application.
    * Most of the methods of class <code>Runtime</code> are instance
    * methods and must be invoked with respect to the current runtime object.
    *
    * @return  the <code>Runtime</code> object associated with the current
    *          Java application.
    */
    public static Runtime getRuntime() {
        return currentRuntime;
    }


    /** Don't let anyone else instantiate this class */
    private Runtime() {}
}
```

Code Exploration for Singleton

- `GameModel` in Solitaire
- `ApplicationReources` in JetUML