M1 (a) – Encapsulation

Jin L.C. Guo

Logistics

- Lab Tests
- Office Hours
- Additional Java Resources

Lab Assessment Schedule

	Start Date	End Date	Focus	Note
Lab Test 1	Jan 23	Feb 3	Encapsulation	
Lab Test 2	Feb 13	Feb 24	Types and Polymorphism, Object State	
Lab Test 3	Mar 16	Mar 29	Design for Robustness, Unit Testing, Composition	
Lab Test 4	Mar 30	Apr 13	Inheritance, Inversion of Control	No lab test on Apr 7 and 10

Lab Test

- Four sessions
- Format: In-person (TR3120)
- Max 18 people for most time slot

Available Slots:

- *We will announce how to sign up the interactive assessment session later this week.*

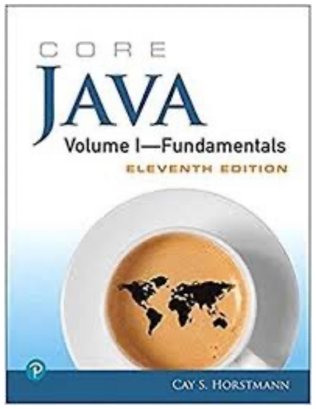
	Start Time	End Time
Monday	10:30 AM	11:30 AM
Monday	11:30 AM	12:30 PM
Tuesday	3:00 PM	4:00 PM
Tuesday	4:00 PM	5:00 PM
Thursday	1:00 PM	2:00 PM
Thursday	2:00 PM	3:00 PM
Friday	10:30 AM	11:30 AM
Friday	11:30 AM	12:30 PM

Office Hours

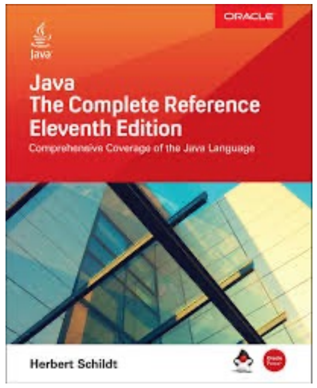
	Start Time	End Time	Location
Monday	9:00 AM	10:00 AM	MC 328
Monday	10:30 AM	11:30 AM	TR3120
Monday	11:30 AM	12:30 PM	
Tuesday	3:00 PM	4:00 PM	
Tuesday	4:00 PM	5:00 PM	
Tuesday	5:00 PM	6:00 PM	
Thursday	1:00 PM	2:00 PM	
Thursday	2:00 PM	3:00 PM	
Friday	10:30 AM	11:30 AM	
Friday	11:30 AM	12:30 PM	

Additional references for Java

- https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html



- Core Java Volume I—Fundamentals, Eleventh Edition



- Java: The Complete Reference, Eleventh Edition

Recap of last class

- The focus and definition of Software Design
- Role of Design in Software Engineering Process
- How to Store and Share Design Knowledge
- Objective of COMP 303

Objectives of this Module

- Programming mechanisms:
  - Scope and Visibility
- Concepts and Principles:
  - Information Hiding, Encapsulation, Escaping Reference, Immutability
- Design Techniques:
  - Object Diagrams
- Patterns and Antipatterns:
  - Primitive Obsession 👎

Very first task (Activity 1)

- Design the representation of a deck of playing cards.

For the purpose of building a card game with one deck of cards.

Definition of Software Design

(As a process) the construction of abstractions of data and computation and the organization of these abstraction into a working software application.

- Abstractions – variables, classes, objects, etc.
- Organization – modularized in a flexible and maintainable manner
- Working – correctly functioning (specification, testing)

Very first task (Activity 1)

- Design the representation of a deck of playing cards.

For the purpose of building a card game with one deck of cards.

Think about your own solutions first. Then, discuss with the students who sit close you, especially about why you designed differently and what are the pros and cons for each design.

Code under design
Client code

Programming Mechanism Review

- Java static type system

Interfaces/Annotations

Classes/Enums $\Bigg\}$ Reference types

Arrays

Primitives *byte, short, int, long, float, double, boolean, char*

Java Memory Organization

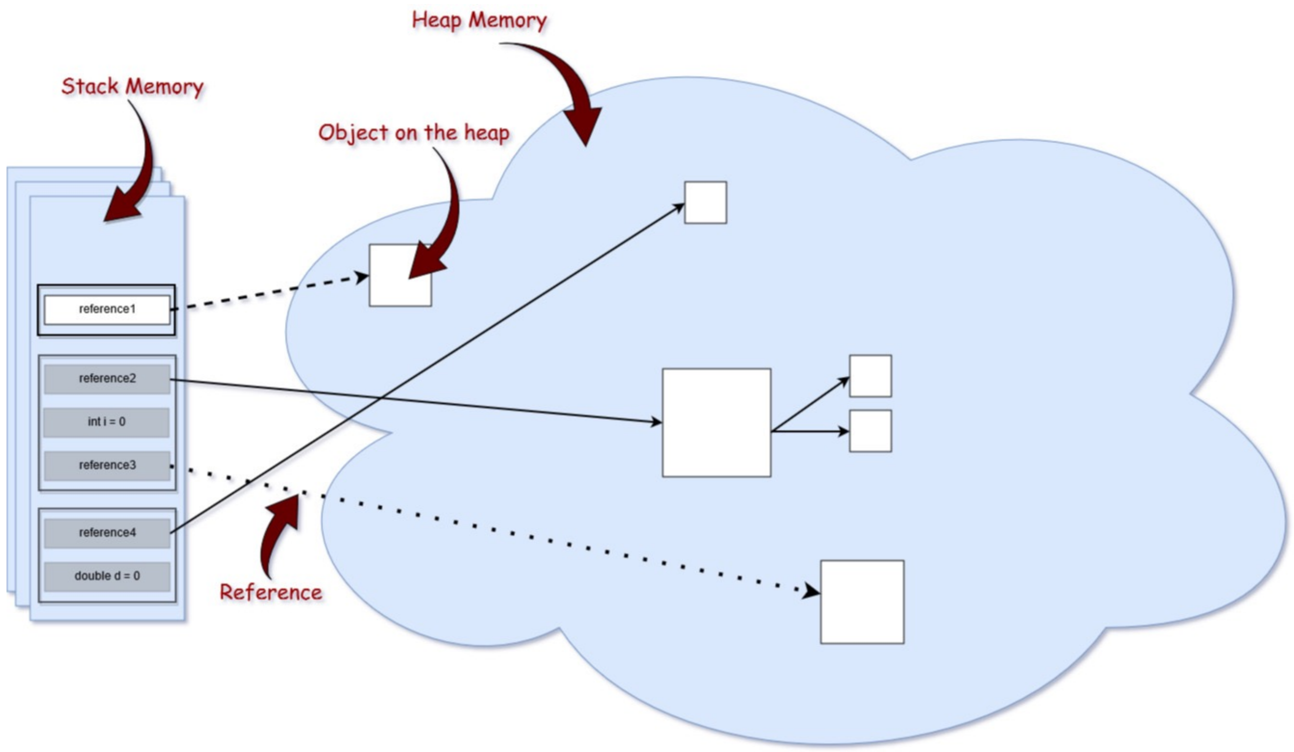


Image Source: https://dzone.com/articles/java-memory-management

Options: using primitive data types

Use integer

- Clubs 0-12
- Hearts 13-25
- Spades 26-38
- Diamonds 39-51

```
int card = 13; // The Ace of Hearts
int suit = card / 13; // 1 = Hearts
int rank = card % 13; // 0 = Ace
```

Problems?

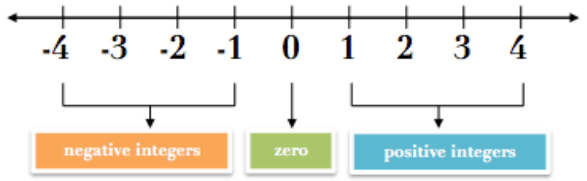
Options: using arrays

- Use pair of values [int, int]
  - Rank 0-12
  - suit 0-4

```
int[] card = {1,0}; // The Ace of Hearts
int suit = card[0];
int rank = card[1];
```

Problems?

Representation



Domain Concept

What about representing phone number with string?

- *Note: the String class is not technically a primitive data type, but considering the special support given to it by the language, you'll probably tend to think of it as such. [Java Primitive Data Types]*

Anti-pattern

- Primitive Obsession
  - **Symptoms**

    Use of primitives for “simple” tasks (such as currency, ranges, special strings for phone numbers, etc.)

Anti-pattern

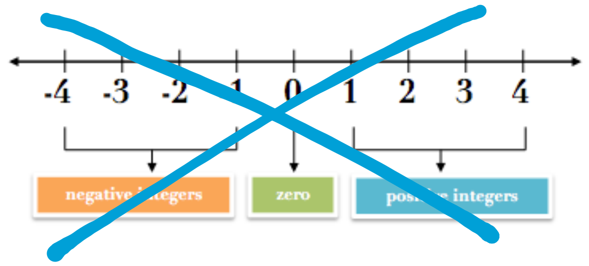
- Primitive Obsession
  - **Symptoms**

    Use of primitives for “simple” tasks (such as currency, ranges, special strings for phone numbers, etc.)

  - **Treatment**

    Replace Primitive with Object (if you are doing things other than simple printing)

Representation
Implementation



Representation
Implementation

loosely coupled

Define our own Card type

```
public class Card
{

        …

}
```

Define our own Card type

```
public class Card
{
        int aId = 0;
        …
}
```

Good choice?

```
Card myCard;

// Initiate and use myCard.
```

Characterizing the Card

int constant? string constant?

- Suit
  - Clubs, Hearts, Spades, Diamonds
- Rank
  - Ace, Two, …, Jack, Queen, King

Characterizing the Card

- Suit
  - Clubs, Hearts, Spades, Diamonds

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

- Rank
  - Ace, Two, …, Jack, Queen, King

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

Java Enum Type

- For predefined constants
- Compile-time type and value safety

  Suit can only be one of *CLUBS, DIAMONDS, SPADES, HEARTS*

Current Focus

- Add methods and other fields
- Instance-controlled -- classes that export one instance for each enumeration constant via a public static final field

Back to our Card Class

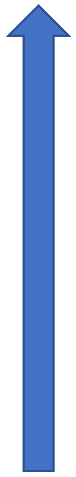
```java
public class Card
{
	public Rank aRank;
	public Suit aSuit;
}
```

```java
card.aRank = null;
System.out.println(card.aRank.toString());

   java.lang.NullPointerException
```

Access Modifiers for class members

- private: accessible from top-level class where it is declared
- package-private (default): from any class in the package
- protected: from subclass and any class in the package
- public: anywhere



Better Encapsulated Card Class

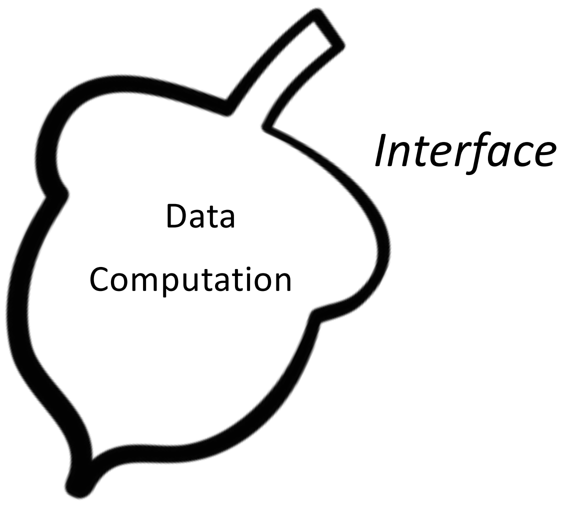
```
public class Card
{
    private Rank aRank;
    private Suit aSuit;

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

    public Rank getRank()
    {
        return aRank;
    }
    public void setRank(Rank pRank)
    {
        aRank = pRank;
    }

    ……
}
```

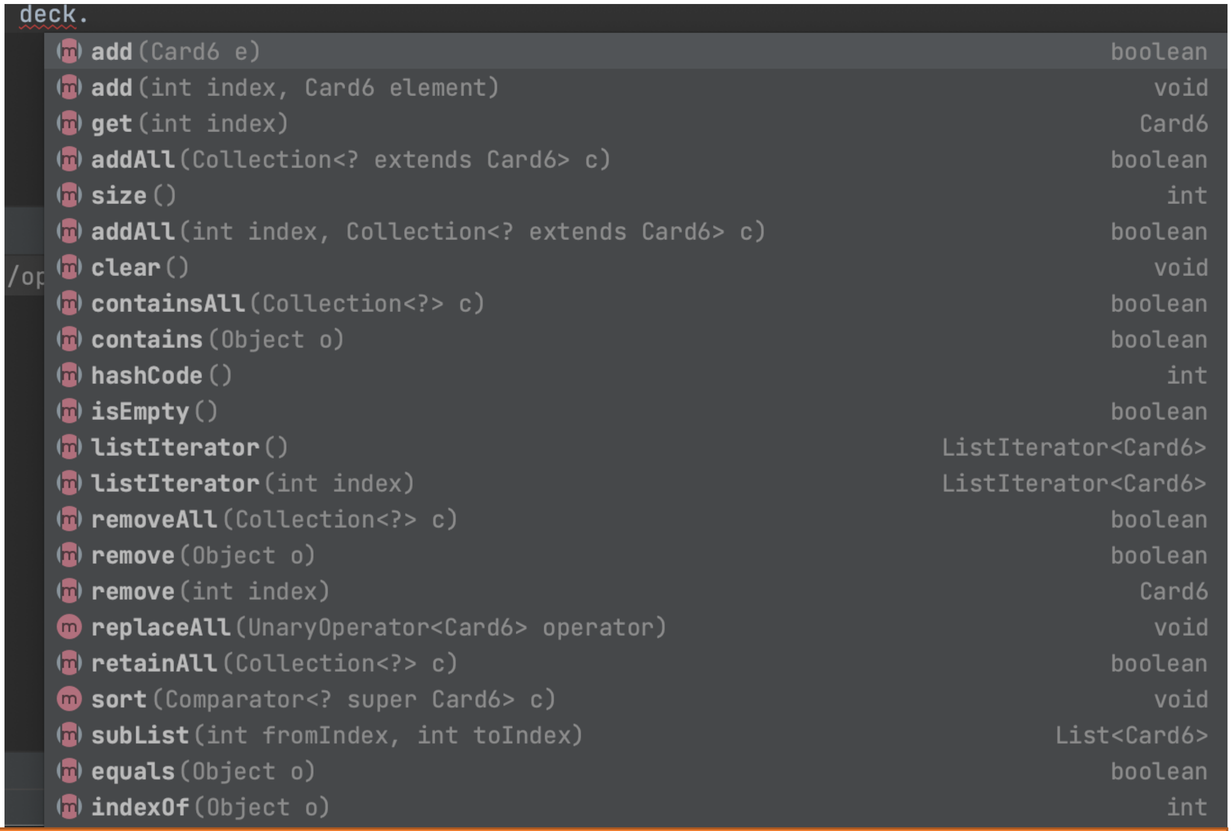
Encapsulation



Goal: to minimize the contact points

Representation of Deck?

```java
List<Card> deck = new ArrayList<>();
```



```
deck.
add(Card6 e)                                          boolean
add(int index, Card6 element)                         void
get(int index)                                        Card6
addAll(Collection<? extends Card6> c)                 boolean
size()                                                int
addAll(int index, Collection<? extends Card6> c)      boolean
clear()                                               void
containsAll(Collection<?> c)                          boolean
contains(Object o)                                    boolean
hashCode()                                            int
isEmpty()                                             boolean
listIterator()                                        ListIterator<Card6>
listIterator(int index)                               ListIterator<Card6>
removeAll(Collection<?> c)                            boolean
remove(Object o)                                      boolean
remove(int index)                                     Card6
replaceAll(UnaryOperator<Card6> operator)             void
retainAll(Collection<?> c)                            boolean
sort(Comparator<? super Card6> c)                     void
subList(int fromIndex, int toIndex)                   List<Card6>
equals(Object o)                                      boolean
indexOf(Object o)                                     int
```

https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/List.html

Representation of Deck?

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

	public void addCard(Card pCard)
	{
		aCards.add(pCard);
	}
}
```

```java
Deck deck = new Deck();
Card card1 = new Card(Rank.ACE, Suit.CLUBS);
Card card2 = new Card(Rank.TWO, Suit.CLUBS);
deck.addCard(card1);
deck.addCard(card2);
```

Information Hiding

- *On the criteria to be used in decomposing systems into modules*

David Parnas - Communications of the ACM, 1972 - dl.acm.org

We have tried to demonstrate by these examples that it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others. Since, in most cases, design decisions transcend time of execution, modules will not correspond to steps in the processing. To achieve an efficient implementation we must abandon the assumption that a module is one or more subroutines, and instead allow subroutines and programs to be assembled collections of code from various modules.

Information Hiding

- A principle to divide any piece of equipment, software or hardware, into modules of functionality.

- Modularization can improve the flexibility and comprehensibility of a system while allowing the shortening of its development time.

Representation of Deck

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

        public void addCard(Card pCard)
        {
                aCards.add(pCard);
        }
}
```

```java
Deck deck = new Deck();
Card card1 = new Card(Rank.ACE, Suit.CLUBS);
Card card2 = new Card(Rank.TWO, Suit.CLUBS);
deck.addCard(card1);
deck.addCard(card2);
```

What can be further improved?

Information Leaking:

- Escaping References: Why this is bad?

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

        public void addCard(Card pCard)
        {
                aCards.add(pCard);
        }
}
```

```
Deck deck = new Deck();
Card card1 = new Card(Rank.ACE, Suit.CLUBS);
Card card2 = new Card(Rank.TWO, Suit.CLUBS);
deck.addCard(card1);
deck.addCard(card2);
```

Information Leaking:

- Escaping References: Why this is bad?

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

        public void addCard(Card pCard)
        {
                aCards.add(pCard);
        }
}
```

```
Deck deck = new Deck();
Card card1 = new Card(Rank.ACE, Suit.CLUBS);
Card card2 = new Card(Rank.TWO, Suit.CLUBS);
deck.addCard(card1);
deck.addCard(card2);
```

```
card1.setRank(Rank.THREE);
```

What is the status of the Deck now?

Escaping References

- It should NOT be possible to change the state of an object without going through its own methods.

- Red flag:

Storing an external reference internally!

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

        public void addCard(Card pCard)
        {
                aCards.add(pCard);
        }
}
```

Escaping References

- It should NOT be possible to change the state of an object without going through its own methods.

- Red flag:

Returning a reference to an internal object!

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

        public List<Card> getCards()
        {
                return aCards;
        }
}
```

Escaping References

- It should NOT be possible to change the state of an object without going through its own methods.

- Red flag:

Leaking references through Shared structures!

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

        public void collect(List<Card> pAllCard)
        {
                pAllCard.addAll(aCards);
        }
}
```

Change Card to Immutable

- Immutable: the internal state of the object cannot be changed after initialization.
- How to change the Card Class?

Change Card to Immutable

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

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

    public Rank getRank()
    {
        return aRank;
    }
    ~~public void setRank(Rank pRank)~~
    ~~{~~
        ~~aRank = pRank;~~
    ~~}~~

    ……
}
```

Change Card to Immutable

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

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

    public Rank getRank()
    {
        return aRank;
    }

    ……
}
```

What about Deck?

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

        public void addCard(Card pCard)
        {
                aCards.add(pCard);
        }
}
```

```
Deck deck = new Deck();
Card card1 = new Card(Rank.ACE, Suit.CLUBS);
Card card2 = new Card(Rank.TWO, Suit.CLUBS);
deck.addCard(card1);
deck.addCard(card2);
```

Objectives of this Module

- Programming mechanisms:
  - Scope and Visibility
- Concepts and Principles:
  - Information Hiding, Encapsulation, Escaping Reference, Immutability
- Design Techniques:
  - Object Diagrams
- Patterns and Antipatterns:
  - Primitive Obsession 👎

Next Lecture

M1 (b) – **Encapsulation**