McGill

COMP 303

Lecture 1

Encapsulation

Winter 2025

slides by Jonathan Campbell, adapted in part from those of Prof. Jin Guo

Announcements

- TAs/Mentors
- Ed discussions
- Office hours to start next week.
    - Google spreadsheet will be sent out.
- Midterm date.
- Feedback from Tuesday's class.
- https://www.codingfont.com/

Plan for today

- Installing Python + Java

- Encapsulation

Python

- Mac:
  - Install Homebrew: https://brew.sh/
    - brew install python
- Windows:
  - https://www.digitalocean.com/community/tutorials/install-python-windows-10
- Some Linux distributions:
  - apt install python3 python3-pip

Python IDE

- https://code.visualstudio.com/
- https://marketplace.visualstudio.com/items?itemName=ms-python.vscode-pylance
- https://mharty3.github.io/til/vs_code/pylance-type-checking/

Java

- JDK: https://www.oracle.com/ca-en/java/technologies/downloads/
- https://www.jetbrains.com/idea/download
  - Scroll down to Community Edition.

Cards

How can we design the representation of a playing card?

Cards

- What is a card?
- A rank from 0-12.
- One of four suits.

Cards

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

Pros:
Very simple - just a single integer.

Cons:
Recomputation of suit/rank.

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

Pros:
Suit and rank represented separately.

Cons:
??

Cards (in Python)

```python
card: int = 13                  # Ace of Hearts
suit: int = card / 13           # Hearts (1)
rank: int = card % 13           # Ace (0)
```

```python
card: list[int] = [1, 0] # Ace of Hearts
```

Types go after the variable name.
List defined using square brackets.

Cards

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

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

Cons:
Really easy to assign an invalid value.
(Only 52 ints are valid!)

```
int card = 9001;                // ???
```

Cards

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

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

Cons:

The representation (int) doesn't match the actual domain concept (playing card).
That is, an int is used to store any integer. It is too general.

```
int numCardsInHand = 7;
int card = numCardsInHand;              // ???
```

Cards

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

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

Cons:

If we decide to change the representation (from int to something else),
we have to make this change everywhere in the code where we use cards.

```
void printCard(int card)
```

Cards in Python

```
card: int = 13              # Ace of Hearts
suit: int = card / 13       # Hearts (1)
rank: int = card % 13       # Ace (0)
```

```
card: list[int] = [1, 0] # Ace of Hearts
```

Cons:

If we decide to change the representation (from int to something else),
we have to make this change everywhere in the code where we use cards.

```
def printCard(card: int) -> None
```

PRIMITIVE OBSESSION

- **PRIMITIVE OBSESSION** is an **anti-pattern** (a common problem that should be avoided).
- It is the practice of using primitive types (int, String, etc.) to represent domain concepts.
  - Primitive types do not contain any model-specific logic or behaviour.
  - Primitive types lose **type safety** (no compiler errors).

Primitive Obsession

- To fix this problem, we create a new `Card` class to represent a playing card.

- The `Card` class will contain the int (or whatever underlying representation), but will not expose it, that is, it will **hide** the underlying representation.

Cards

```
class Card {
    public int aCard; // 0-51 encodes the card
    public Card(int card) {
        this.aCard = card;
    }
}
```

Cards (in Python)

```
class Card:
    def __init__(self, card: int):
        self.aCard: int = card
```

Cards

- A special property about a playing card is that it can be uniquely defined by its suit and rank, and each of these has a fixed number of possible values.

- To represent these fixed values, we can use an **`enum`**.

Cards

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

enum Rank {
    ACE, TWO, ..., QUEEN, KING
}

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

Cards in Python

```
from enum import Enum
class Suit(Enum):
    CLUBS = 1
    DIAMONDS = 2
    SPADES = 3
    HEARTS = 4

class Card:
    def __init__(self, suit: Suit, ...):
        self.suit : Suit = suit
```

Deck

- How can we design the representation of a deck of cards?

```
List<Card> deck = new ArrayList<>();
deck : list[Card] = []
```

- Same problem as using an int for a card:
  - Could represent any list of cards, not just a deck.
  - If we want to change the representation, we'd have to change it everywhere in the source code.
  - Can easily be corrupted: could put in duplicates, etc.

Deck

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

- Now our type is only for Decks, and nothing else.

- Hides the decision of how cards are stored, so that it can be easily changed later.

- Lets us define specific methods to be used on Decks inside the same class, coupling data with computation.

Encapsulation

- Creating a type for our design abstraction is the first step of **encapsulation**:
  - the idea that data and computation should be bundled together,
  - external code should not need to worry about exactly how the data is represented, nor how the computation is done.

Without bundling

```
class Deck {
    public List<Card> aCards = new ArrayList<>();
}
class Card {
    public Rank aRank = null;
    public Suit aSuit = null;
}

// later
Deck deck = new Deck();
deck.aCards.add(new Card());
deck.aCards.add(new Card());
deck.aCards.get(1).aRank = deck.aCards.get(0).aRank;
System.out.println(deck.aCards.get(0).aSuit.toString());
```

Recall: access control modifiers

- For good encapsulation, we use the narrowest possible scope for class members.

Restricting scope

```
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 Suit getSuit() {
      return aSuit;
   }
}
```

Client code cannot interact with internal representation.
It can only use the public methods.

Restricting scope (Python)

```
class Card:
  def __init__(self, pRank: Rank, pSuit: Suit):
    self.__aRank = pRank # using __ instead of
    self.__aSuit = pSuit #                    private

def getRank(self) -> Rank:
  return self.__aRank

def getSuit(self) -> Suit:
  return self.__aSuit
```

Escaping references

```
public class Deck {
   private List<Card> aCards = new ArrayList<>();
   public Deck() {
      /* Add all 52 cards to the deck */
      /* Shuffle the cards */
   }
   public Card draw() {
      return aCards.remove(0);
   }
   public List<Card> getCards() {
       return aCards;
   }
}
```

Problem!

References

- Robillard ch. 2 (p.13-41)

Coming up

- Next lecture:
  - Encapsulation II