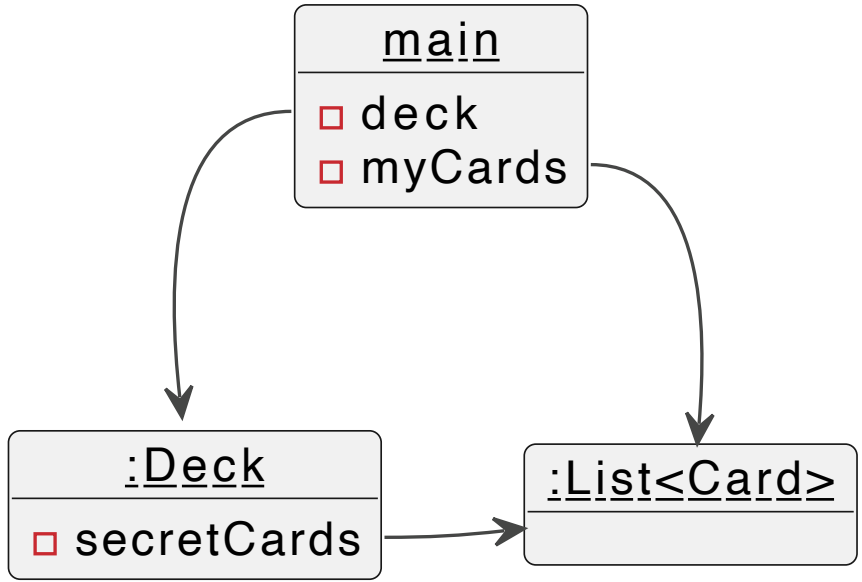
Reference escaping

- Cosa è? Violazione encapsulation



1. getter ritorna un riferimento a un *segreto*

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

2. setter assegna al *segreto* un qualche riferimento che gli viene passato

```
public void setCards(List<Card> cards) {
  secretCards = cards;
}
```

3. costruttore assegna al *segreto* un qualche riferimento che gli viene passato

```
public Deck(List<Card> cards) {
  secretCards = cards;
}
```

Encapsulation and information hiding

Parnas [L8]
Solo ciò che è nascosto può essere cambiato liberamente e senza pericoli

I due scopi fondamentali sono:

- facilitare la comprensione del codice
  - vengono definite le responsabilità
- rendere più facile modificarne una parte senza fare danni

Immutabilità

- Cosa è una classe immutabile?
- Non c'è modo di cambiare lo **stato** dell'oggetto dopo la sua inizializzazione
  - non fornisce metodi che modificano lo stato
  - ha tutti attributi *privati* (non obbligatorio)
  - ha tutti gli attributi *final* (non obbligatorio dichiararlo)
  - assicura accesso esclusivo a tutte le parti non immutabili

Code smell

https://en.wikipedia.org/wiki/Code_smell
https://refactoring.guru/refactoring/smells
https://luzkan.github.io/smells/

- Codice duplicato
- Metodo troppo lungo
- Troppi livelli di indentazione
- Troppi attributi/responsabilità per classe
- Lunghe sequenze di *if-else* o *switch*
- Classe troppo grande
- Lista di parametri troppo lunga
- Numeri *magici*
- Commenti
- Nomi oscuri o inconsistenti
- Codice morto
- *getter* e *setter*

Un mazzo di 52 carte

- Vogliamo *ideare* una **struttura dati** capace di rappresentare un mazzo o una mano (sequenza) di carte tratte dal mazzo
- Il mazzo è quello con:
  - 4 semi (cuori, quadri, fiori e picche)
  - 13 valori (asso, due, …, regina, re)

Prima idea

- Mappiamo le nostre 52 carte su 52 interi: [0-51] (0-12 Cuori, 13-25 Quadri…)
- Usiamo il contenitore di interi per la sequenza di carte

```
1  int[] deck = {15,10,1,8}; // mano con quattro carte
2
3  int card = 15;          // 15 = ??
4  int suit = card / 13; // 1 => Quadri
5  int rank = card % 13; // 2 => Tre (si comincia da 0)
```

Problemi:

- scomodo dover fare operazioni per capire seme e valore
- non inibisco operazioni non lecite (per il compilatore è un intero)

Encapsulation e astrazioni

- Diamo un nome ai concetti (type abstraction)

```java
class Deck {
  private Card[] cards;
}

class Card {
  private int value; // 0-51
}
```

```java
class Card {
  private int[] card = {1, 3};
  // 1 = Quadri, 3 = Tre (cominciando da 1)
}
```

Rappresentazione interna

```java
enum Suit { CLUBS, DIAMONDS, SPADES, HEARTS };
enum Rank { ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
            NINE, TEN, JACK, QUEEN, KING };

class Card {
  private Suit suit;
  private Rank rank;
}

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

Getter e setter ?

```java
class Card {
  private Suit suit;
  private Rank rank;


  public Rank getRank() { return rank; }
  public Suit getSuit() { return suit; }
  public void setRank(Rank r) { rank = r; }
  public void setSuit(Suit s) { suit = s; }
}

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

```java
// Volendo potevo mantenere un int per
// struttura interna
class Card {
  private int suit;
  public Suit getSuit() {
    return switch (suit) {
      case 0 -> Suit.CLUBS;
      case 1 -> Suit.DIAMONDS;
      case 2 -> Suit.SPADES;
      case 3 -> Suit.HEARTS;
      default -> null;
    };
  }

  public void setSuit(Suit s) {
    suit = s.ordinal();
  }
}
```

Problemi

```java
class Main {
  public static void main(String[] args) {
    Deck deck = new Deck();
    Card card = new Card();

    card.setSuit(Suit.DIAMONDS);
    card.setRank(Rank.THREE);
    deck.getCards().add(card);
    deck.getCards().add( new Card() );

    System.out.println("There are " + deck.getCards().size() + " cards:");

    for (Card currentCard : deck.getCards())
      System.out.println(currentCard.getRank() + " of " + currentCard.getSuit());
  }
}
```

https://martinfowler.com/bliki/TellDontAsk.html

Tell-Don't-Ask Principle

Non chiedere i dati, ma dì cosa vuoi che faccia sui dati

- Cercare di minimizzare i getter studiando cosa ci facciamo con il valore ritornato e definendo funzioni opportune

```
class Card {
  private Suit suit;
  private Rank rank;

  public Card(Suit s, Rank r) {
    suit = s;
    rank = r;
  }
  @Override
  public String toString() {
    return rank + " of " + suit;
  }
}
```

```
class Deck {
  private ArrayList<Card> cards = new ArrayList<>();
  @Override
  public String toString() {
    String ans = "There are " + cards.size() + " cards
    for (Card currentCard : cards)
      ans += currentCard.toString() + '\n';
    return ans;
  }
  public void add(Card card) {
    cards.add(card);
  }
}
```

Estraiamo le interfacce

```java
public static List<Card> drawCards(Deck deck, int number) {
  List<Card> result = new ArrayList<>();
  for (int i = 0; i < number && !deck.isEmpty(); i++) {
    result.add(deck.draw());
  }
  return result;
}
```

Questo metodo funziona solo per gli oggetti di tipo Deck

In realtà cosa gli interessa?

solo che l'oggetto abbia i due metodi

- `isEmpty()` che controlla se ci sono ancora carte
- `draw()` che restituisce la prossima carta (e la toglie dal mazzo)

Interfaccia e Polimorfismo

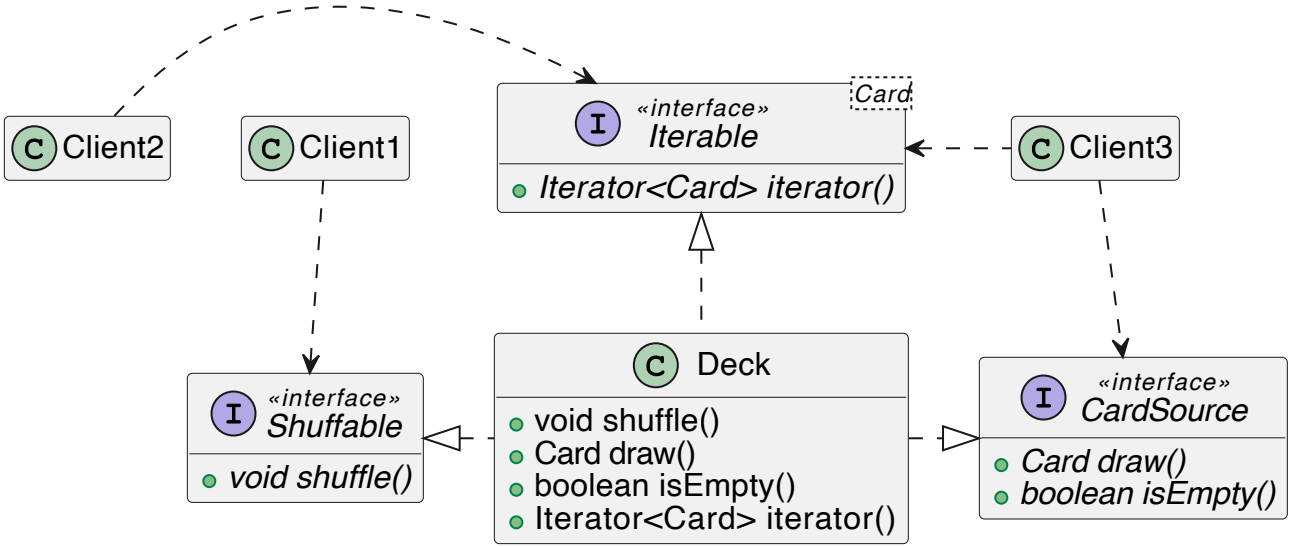
```java
public interface CardSource {
  /**
   * @return The next available card.
   * @pre !isEmpty()
   */
  Card draw();

  /**
   * @return True if there is no card in the source.
   */
  boolean isEmpty();
}

public class Deck implements CardSource { . . . }

public static List<Card> drawCards(CardSource deck, int number) {
  List<Card> result = new ArrayList<>();
  for (int i = 0; i < number && !deck.isEmpty(); i++) {
    result.add(deck.draw());
  }
  return result;
}
```

Interface segregation



Polimorfismo e *loose coupling*

- La capacità di un identificatore di variabile/parametro di accettare oggetti in realtà di forme diverse, a patto che siano dei suoi sottotipi

```
Deck deck = new Deck();

CardSource source = deck;

List<Card> cards;
cards = drawCards(deck, 5 );
```

Collegamento dinamico e *extensibility*

```
public static List<Card> drawCards(CardSource cardSource, int number) {
  List<Card> result = new ArrayList<>();
  for (int i = 0; i < number && !cardSource.isEmpty(); i++) {
   result.add(cardSource.draw());
  }
  return result;
}
```

- Il compilatore non sa a tempo di compilazione quale metodo `draw()` dovrà chiamare
- Rimanda la decisione al momento effettivo della esecuzione

Permette di chiamare codice *non ancora scritto*

Collegato quindi a estendibilità, a *Open Close Principle*

Esempio nella libreria standard di Java

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

  public void shuffle() { Collections.shuffle(cards); }
}
```

- `cards` è un oggetto di tipo `ArrayList<Card>`
- che implementa interfaccia `List<Card>`
- che è passabile come parametro a `shuffle` di Collections

```
public static void shuffle(List<?> list)
```

- `shuffle` accetta List di qualunque elemento

Infatti per *disordinare* una lista di elementi non dobbiamo mica guardarli…

DOMANDA: per riordinare un mazzo di carte, ci sarà una funzione equivalente da sfruttare?

```
public void sort() { Collections.sort(cards); }
```

Comparable interface

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

- Definisce un singolo metodo che restituisce un valore minore, uguale o maggiore a 0, a seconda del risultato del confronto.
- Questo metodo è quello che userà la `sort` di Collections.

Quindi la interfaccia di `Collections.sort` è:

```
public static <T extends Comparable<? super T>> void sort(List<T> list);
```

Cioè `sort` ha bisogno che gli elementi della lista da ordinare implementino `Comparable`, e quindi siano confrontabili tra di loro.