McGill

COMP 303

Lecture 3

Types & polymorphism

Winter 2025

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

Announcements

- Office hour schedule on myCourses

- Team forming PDF

Plan for today

- Recap from last time
- Types & polymorphism
- TicTacToe example

Recap

Recap: terms from last time

- Object diagrams
- Escaping references
- Immutable objects
- Input validation
- Design by contract

Recap: Design by contract

- To remove this ambiguity, when writing a method, we will write a **contract** that specifies what should be true about the inputs (and outputs).

- This takes the form of **preconditions** and **postconditions**.

Recap: Input validation

```
/**
* ...
* @throws IllegalStateException if the deck
* is empty
*/
public Card draw() {
   if (isEmpty()) {
      throw new IllegalStateException();
   }
   return aCards.remove(aCards.size() - 1);
}
```

Types & polymorphism

From COMP 250

- Inheritance (extends)

- Interfaces (implements)

Recall: inheritance

```
class Animal {
  public void eat() {
    System.out.println("This animal eats food.");
  }
}
```

```
class Dog extends Animal {
  public void bark() {
    System.out.println("The dog barks.");
  }
}

Dog dog = new Dog();
dog.eat();
dog.bark();
```

Subclass inherits all methods and fields from parent.

Recall: inheritance

- A Dog **is a**(n) Animal.

- A HumanPlayer **is a** Player. An NPC **is a** Player.

- When we use "**is a**" to describe the relationship between two classes, it means that we want to one class to inherit from (**extend**) the other (the more specific inherits from the more general).

Recall: inheritance

- A subclass (or child class) inherits all fields and methods of the superclass (or parent class).

- A subclass can also override methods to customize behaviour.

- In Java, a class can only inherit from **one** other class (one parent).

Recall: interfaces

```
interface Flyable {
    void fly();
}
```

No implementation:
only specifies method headers

```
class Bird implements Flyable {
    public void fly() {
        System.out.println("Bird is flying.");
    }
}
```

Recall: interfaces

```
interface Printable {
    void print();
}

interface Scannable {
    void scan();
}

class MultiFunctionPrinter implements Printable, Scannable {
    public void print() {
        System.out.println("Printing...");
    }

    public void scan() {
        System.out.println("Scanning...");
    }
}
```

No implementation:
only specifies method headers

Recall: interfaces

- A Bird **can** Fly.

- A Printer **can** Print. A Multi-function printer **can** print and **can** scan.

- When we use "**can**" to describe the relationship between a class and interface, it means that we want the class to **implement** that interface.

Recall: interfaces

- An interface is only a **specification** of method headers. If a class implements the interface, it is expected to provide implementations for those methods.

- There is **no shared code** (methods or fields)* between two classes that implement the same interface (unlike two classes that inherit from the same parent).

- In Java, a class can implement any number of interfaces.

*Note: There are some simplifications here which we may revisit when we discuss abstract classes later in the term.

Inheritance vs. interfaces

- Inheritance:
  - for functionality that is shared between classes (is-a relationship).
  - E.g., a dog is a mammal.
- Interfaces:
  - for a shared behaviour that is implemented differently in each implementing class (**subtype**).
  - E.g.: a bird and a bat can both fly, but they do so differently, and a bird is not a mammal while a bat is.

Inheritance vs. interfaces

```
public class Dragon extends Enemy
         implements MonsterLike, FireBreather {

}
```

Dragon is a subtype of (at least) Enemy, MonsterLike, and FireBreather.
An instance of Dragon can be used whenever an object of those types is required.

Recall: Deck

```
class Deck {
    private List<Card> cards;

    public Deck() {
        // populate with 52 cards
        // shuffle
    }
}
```

Drawing from the Deck

```
class Deck {
    private List<Card> cards;

    public Deck() {
        // populate with 52 cards
        // shuffle
    }

    public Card draw() {
        return cards.remove(cards.size() - 1);
    }
}
```

Other kinds of Card lists

- CardSequence: a list of cards arranged in some kind of order (e.g., when playing Gin or Rummy).

- AggregatedDeck: combines multiple 52-card decks into one giant deck.

- Trick: a collection of cards played by all players in a single round of certain card games (e.g., when playing Hearts).

Other kinds of Card lists

- Now, consider the following method, which takes a Deck and draws a certain number of cards from it.

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

Other kinds of Card lists

- The drawCards method only works with Deck objects, and calls its draw method.
- But we may want to use drawCards for our other classes (AggregatedDeck, CardSequence, Trick).
- How can we do so?
  - We can make all these classes **implement** the same interface (which defines a draw method).

CardSource

```
public interface CardSource {
   /**
   * Returns a card from the source.
   *
   * @return The next available card.
   * @pre !isEmpty()
   */
   Card draw();
   /**
   * @return True if there is no card in the
   * source.
   */
   boolean isEmpty();
}
```

CardSource

```
public class Deck implements CardSource {
    ...
}

// CardSource source = new Deck();
```

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

Data

Data

Computation

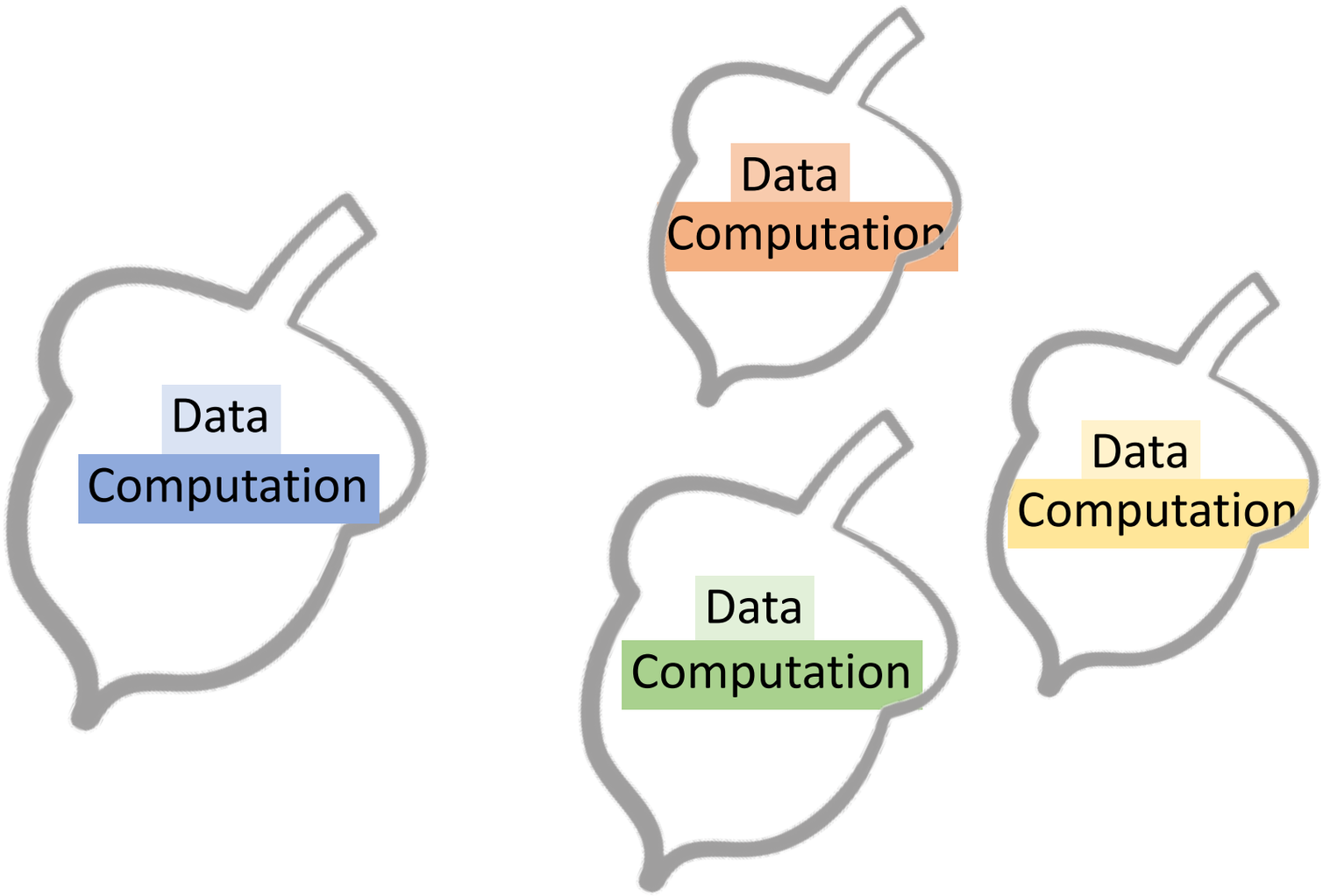
Computation

Data

Computation

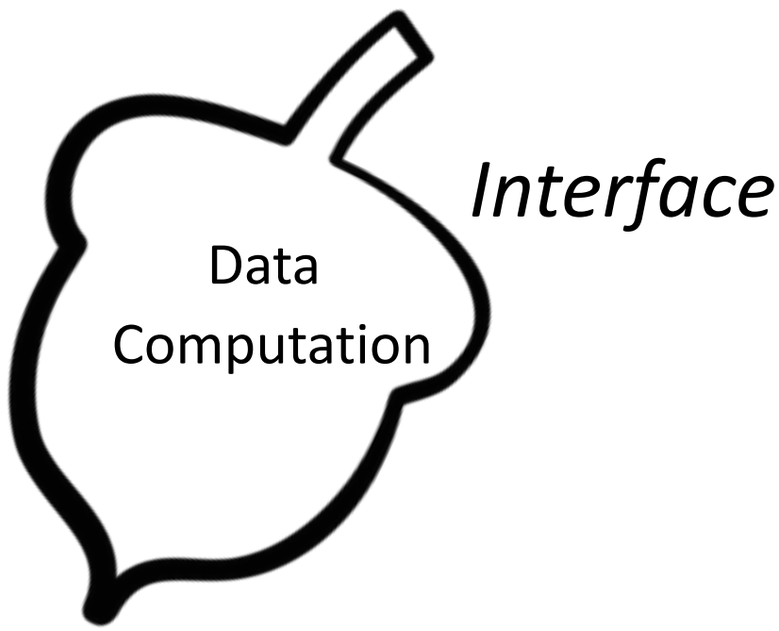
Data

Computation



Functions are achieved through Object Interaction

Object Interaction



Supply the service through public interface

Polymorphism

- **Polymorphism**: "having many forms"
- The idea that an object can look like many different things. E.g., if a class inherits from another class, or implements an interface, it can look like that class/ interface!

Polymorphism

- There are different types of polymorphism:

  - **subclass** polymorphism: when a class inherits from a parent class, then it can look like the parent class. (We will talk about this later.)

  - **subtype** polymorphism: more general than the above, and includes the case when a class implements an interface (so it can look like that interface) -- meaning that it implements all the operations in it.

  - **parametric** polymorphism: when a function is written that can operate on multiple types, e.g., in Python, or using generics in Java or templates in C++.

Polymorphism

- What are the benefits of polymorphism?
  - **Extensibility**: new types can be introduced which, if implementing an interface or inheriting from a parent, can automatically be used in existing code.
  - **Loose coupling**: code that calls methods on an object which were specified in an interface does not need to worry about the particular implementation of those methods.

Parametric polymorphism

```
public interface ListOfCard {
    boolean add(Card pElement);
    Card get(int index);
}
```

```
public interface ListOfNumbers {
    boolean add(Number pElement);
    Number get(int index);
}
```

```
public interface ListOfIntegers {
    boolean add(Integer pElement);
    Integer get(int index);
}
```

Generics

```
public interface List<E> {
    boolean add(E pElement);
    E get(int index);
}

// List<Card> cards
```

Generics

```
public interface List<E> { // E: type parameter
    boolean add(E pElement);
    E get(int index);
}

// List<Card> cards = new ...
//       Card:  type argument
```

Generics

```
public interface List<E> { // E: type parameter
    boolean add(E pElement);
    E get(int index);
}

// List<Card> cards
//       Card:  type argum
```

Convention:
E for Element
K for Key
V for Value
T for Type
? for Type not used in method.

Generics: Pair

```
public class Pair<T>
{
    final private T aFirst;
    final private T aSecond;

    public Pair(T pFirst, T pSecond)
    {
        aFirst = pFirst;
        aSecond = pSecond;
    }

    public T getFirst() { return aFirst; }
    public T getSecond() { return aSecond; }
}
```

TicTacToe

- Implementation

- Integrating into project

References

- Robillard ch. 3 (p.43-46)

Coming up

- Next lecture:
  - More about types and polymorphism