Last Week

- Discussed and analysed the algorithm of:
  - Linear search
  - Binary search
- The use of Pseudocode in algorithm formulation
- Factors affecting the choice of algorithms
  - Time efficiency
  - Space efficiency
  - Development cost
  - Communication bandwidth and Communication methods
  - etc.
- Introduced Asymptotic notation
  - $O(n)$
  - $\Omega(n)$
  - $\Theta(n)$

  and their usage

CPT108 Data Structures and Algorithms

Lecture 5

Data Structures and Abstract Data Type

A Use Case: Phone Book

In the phone book, we have:

- A set of people' name and their phone numbers

Function required

- Search the phone number by person's name

Problem analysis (for the function)

- Input: What is given?
- Output: What is required?
- Constraints: Under what conditions?
- Abstraction: What information are essential?

Phone book

```
public class PhoneBook {

  public static void main(String[] arguments) {
    String[] names = new String[] { "Alan Turing", "Herbert Simon",
        "John von Neumann", "Edsger Dijkstra", "Linus Torvalds" };
    String[] contacts = new String[] { "+86 188 1234 5678", "+86 123 9876 5432",
        "+86 (51) 1357 2468", "+86 (51) 8642 7531" };

    String nameToSearch = "Alan Turing";
    int index = Search.linearSearch(nameToSearch, names);

    System.out.println("Name to search: " + nameToSearch);
    if (index < 0) System.out.println("Contact not found!");
    else System.out.println("Contact: " + contacts[index]);
    System.out.println("");

    nameToSearch = "Edsger Dijkstra";
    index = Search.linearSearch(nameToSearch, names);

    System.out.println("Name to search: " + nameToSearch);
    if (index < 0) System.out.println("Contact not found!");
    else System.out.println("Contact: " + contacts[index]);
    System.out.println("");

    nameToSearch = "Linus Torvalds";
    index = Search.linearSearch(nameToSearch, names);

    System.out.println("Name to search: " + nameToSearch);
    if (index < 0) System.out.println("Contact not found!");
    else System.out.println("Contact: " + contacts[index]);
    System.out.println("");
```

We can create two arrays, one for storing the names of the persons, and the other for storing their phone numbers.

We can then use linear search (or binary search) to search for the index of the person's name, and use the index on another array to retrieve the phone number.

Problem with the current approach

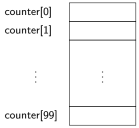
- The `name` and `contacts` are decoupled as if they have no relation at all!
- Difficult to modify or update
  - e.g., what will happen if we want to add some additional information to the person?

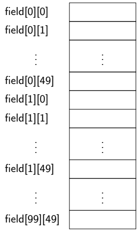
    ...say a second phone number?
- Error-prone

```
String[] names = { "Alan Turing", "Herbert Simon",
    "John von Neumann", "Edsger Dijkstra", "Linus Torvalds" };
String[] contacts = { "+86 188 1234 5678", "+86 123 9876 5432",
    "+86 (51) 1357 2468", "+86 (51) 8642 7531" };
```

Abstract Data Types

- Primitive data type
  - `int`, `long`, `double`, `String`, etc.
- Simple Data Structure like Array
  - stores elements of the same type in a contiguous memory





row-major ordering

- Other data type, such as:
  - `List`, `Set`, `Map`

  are known as Abstract data types (ADTs), i.e., a data type that consists of a *collection of values* together with a set of *basic operations* on these values

➡ *May not be suitable to represent the data as we need*

Abstract Data Types (cont.)

How can we represent a person in a phone book?

- We can create our own abstract data type (ADT) called `Person`, and keep everything together.
- In Java, it is just the same as a simple Java class!
  - And is also referred to as “*plain old java object* (POJO)”

```
public class Person {
  public String name;
  public String contact;
}
```

name of the object

attributes/properties of the object

- Like other data type
  - Use `new` to create a new object, e.g., `Person person = new Person();`
  - Use `.<attribute_name>` to refer to the object’s attribute, e.g., `person.name = "Bill"` will set the `name` of the object `person` to “Bill”

Abstract Data Types (cont.)

```
public class PhoneBook {

  public static void main(String... arguments) {

    Person[] persons = new Person[5];

    persons[0] = new Person();
    persons[0].name="Alan Turing";
    persons[0].contact="+86 188 1234 5678";

    persons[1] = new Person();
    persons[1].name="Herbert Simon";
    persons[1].contact="+86 123 9876 5432";

    .
    .
    .
    persons[4]=new Person();
    persons[4].name="Linus Torvalds";
    persons[4].contact="+86 188 3062 4700";

    String nameToSearch = "Alan Turing";
    int index = linearSearch(nameToSearch, persons);

    System.out.println("Name to search: " + nameToSearch);
    if (index < 0) System.out.println("Contact not found!");
    else System.out.println("Contact: " + persons[index].contact);
    System.out.println("");

    nameToSearch = "Edsger Dijkstra";
```

Create an array similar to the previous one but with `Person` data type

For each entry we have to create a `new` object

and set the property values

and the rest are similar

```
public class Person {
  public String name;
  public String contact;
}
```

this also needs to be changed

Abstract Data Types (cont.)

Object Construction and Destruction

- Constructor and Destructor
  - *Constructor*
    - used to create an instance of the object and allocate “enough” memory to it
  - *Destructor*
    - invoke automatically when the object is out of scope, and
    - *release*/*free* the memory back to the system
  - Every object *must* have a constructor; while the destructor is an *optional*
    - If *no* constructor is provided, then the system will create a constructor, known as *default constructor*, automatically
  - A *default constructor* (or *default value constructor*) is a constructor with *no* argument, e.g., `Person person = new Person();`
  - Similar to other functions, data, known as *arguments*, can be passed to the constructor to *initialize* the object, e.g.,
    `Person person = new Person(name, contact);`
  - Destructor, on the other hand, does *not* require any argument

It should be noted that there is *no* concept of destructor in Java. Instead, the Java garbage collector (GC) (inside the Java Virtual Machine (JVM)) will dispose any Java objects that are out-of-scope automatically.

Abstract Data Types (cont.)

```
public class Person {

    public String name;
    public String contact;

    public Person(String name, String contact) {
        this.name = name;
        this.contact = contact;
    }

    public Person() {
        this("", "");
    }

}
```

Constructor has to be the same name as the class

We can put the name and contact as arguments to the constructor

And initialize the object's values here!

set name = "" and contact = ""

Abstract Data Types (cont.)

In practice

```
public class Person {

  public String name;
  public String contact;

  public Person() {
    this("", "");
  }

  public Person(String name,
    String contact) {
    this.name = name;
    this.contact = contact;
  }

}
```

1. Change all attributes' modifiers to `private`

```
public class Person {

  private String name;
  private String contact;

  public Person() {
    this("", "");
  }

  public Person(String name,
      String contact) {
    this.name = name;
    this.contact = contact;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getContact() {
    return contact;
```

2. Provide access to attributes through different methods

These methods also known as "*getters*" and "*setters*" of the attributes

Data-directed design

- Design directed by the choice and representation of data structures
- Data requirements:
  - In addition to the getters and setters methods, what functions to be performed on the data
  - What's the proper scope
    - Ownership?
      – who owns the data
    - How is it ***shared***?

An example: Counter

Counter: A device which stores the number of times a particular event or process has occurred.

- Data required
    - counter value (`int`)
- Functions provide
    - initialize/reset counter
    - increment counter
    - decrement counter
    - return counter value

```
public class Counter {

  private int value;

  /**
   * Initialize/reset the
         counter
   */
  public void reset() {}

  /**
   * Increment counter
   */
  public void increment() {}

  /**
   * Decrement counter
   */
  public void decrement() {}

  /**
   * Get counter value
   *
   * @return value of the
         counter
   */
  public int getValue() {}

}
```

An example: Counter (cont.)

```
public class Counter {

  private int value;

  /**
   * Initialize/reset the counter
   */
  public void reset() {
    value = 0;
  }

  /**
   * Get counter value
   *
   * @return value of the counter
   */
  public int getValue() {
    return value;
  }

  /**
   * Increment counter
   */
  public void increment() {
    if (value < Integer.MAX_VALUE) {
      value++;
    } else {
      System.out.println("Counter
           overflow: Increment ignored!"
           );
    }
  }

  /**
   * Decrement counter
   */
  public void decrement() {
    if (value > Integer.MIN_VALUE) {
      value--;
    } else {
      System.out.println("Counter
           underflow: Decrement ignored!
           ");
    }
  }

}
```

Can we use a constructor here to initialize the counter and remove the `reset` method?

```
public Counter() {
  value = 0;
}
```

or

```
public Counter() {
  initialize();
}
```

Another example: Complex number

Complex number: a number system that extends the real numbers with an imaginary unit

- Data required
  - real (`real`)
  - imaginary (`real`)
- Functions provide
  - addition
  - multiplication

```
public class IndexedSet {

  private static final int DEFAULT_MAX_SIZE =
        20;
  private static final int INVALID_INDEX =
       -1;

  private int maxSize;
  private int size;

  private double[] items;

  public IndexedSet() {}

  public IndexedSet(int maxSize) {}

  public int addItem(double item) {}

  public double retrieve(int index) {}

  public int getSize() {}

}
```

An example: Complex number (cont.)

```
public class Complex {

  private double real;
  private double imag;

  public Complex(double real, double imag) {
    this.real = real;
    this.imag = image;
  }

  public Complex() {
    this(0.0, 0.0);
  }

  public void add(Complex complex) {
    this.real += complex.real;
    this.imag += complex.imag;
    return this;
  }

  public void multiply(Complex complex) {
    this.real = this.real * complex.real - this.imag * complex.imag;
    this.imag = this.real * complex.imag + this.imag * complex.real;
    return this;
  }

  // + all getters and setters

}
```

How to resolve the problem?

What is the output of the following code segment?

```
Complex complex1 = new Complex(1, 2) ;
Complex complex2 = new Complex(1, 2) ;
System.out.println("complex1.real=" + complex1.real);
System.out.println("complex1.imag=" + complex1.imag);
System.out.println( (complex1 == complex2) );
```

Reading

- Chapter 3, Cormen (2022)