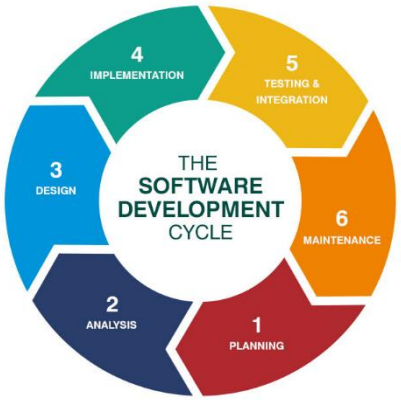
CS5132
DATA STRUCTURES AND ALGORITHMS

Name:_____ Date: _____

CHAPTER 0: INTRODUCTION TO DATA STRUCTURES

0.1 Software Development Life Cycle (SDLC) and Principles

In systems and software engineering, the Software / System Development Life Cycle (SDLC) is the process for planning, creating, testing, and deploying a software / system. There are **six stages** in the cycle, which are,



1) Planning
2) Analysis of Findings / Requirements Analysis
3) Design and Development
4) Implementation
5) Testing, Integration and Documentation
6) Maintenance

Planning and Requirements Analysis

Proper planning and building the required functionalities is the development of the software or system. In the requirements analysis phase, the objective here is to discover the ***objectives***, ***nature*** and ***scope*** of the software or system. A key aspect will be to establish the **problem statements and key facts and issues from the business / user point of view**. Such information are obtained through:

- Interviews with important stakeholders / company executives who hired you to build the software / system
- Surveys distributed to the target audience of the software or system
- Interviews done with selected members of the target audience, preferably representations of different categories of users (level and ability with tech, age, problems encountered, etc)

As a result of this phase, the following questions about your new / proposed upgraded software should be clearly answered:

- **Who** is the target audience / users of the software or system?
- **Why** does the software or system need to be developed in the first place?
- **What** are some problems the software or system is supposed to resolve? **What** are some objectives the new / upgraded software or system is supposed to fulfill?
- **When** should the software or system supposed to be complete?
- **Where** will the software or system be deployed and on what platform?
- **How** feasible it is to create the new / upgraded software or system?

Design and Development; Analysis of algorithms

When designing a software, the process involves user interface design, **selection of data structures**, and **formulating algorithms**. There are many design techniques available, whereby one good technique for designing an algorithm is the idea of decomposition, to break down the task at hand into multiple smaller subtasks, for each developer team to handle then after which integrating the solution together to form one big system.

The object-oriented approach allows for decomposition into smaller well-defined objects, which can then be developed by different developers simultaneously. Each developer develops a few classes, depending on the distinct functionality. The object-oriented approach also allows for abstraction, which is the separation of important details from the unimportant ones. The ***abstraction barrier*** allows us to implement class methods without needing to know the implementation details. To further support this principle, it is important to **add useful comments as a description of what each method does, or if it is due for an implementation update**. Given that software is usually developed as a team, this makes comments vital in allowing a developer, with minimal knowledge of the module, to pick up the pieces quickly and work on what is needed.

IntelliJ (and most other IDEs) have the **`TODO`** tag for comments, which signifies what further implementation updates or improvements the method is required, and is easily searchable too:

```
/**
 * TODO: Find out if there's an efficient alternative for
 * linear algebra and make this method a wrapper around it
**/
public static Matrix transpose(Matrix matrix){
    // contents of the method
}
```

There is also a standard convention in giving other developers a "summary" of what the given method is about, specifying the description, parameters (**`@param`**) and the return (**`@return`**):

```
/**
 * Translates text from English to alienspeak
 * @param textInEnglish the text in English I want to translate
 * @return the text after being translated into alienspeak
 */
public String translateToAlienspeak(String textInEnglish) {
    // contents of the function
}
```

Modularity enhances the understandability of software systems and change process. **Developers need not have to understand the entire system for changes to be made as details are localized into components**.

Any software engineering project in the industry, is done as a team of developers. Hence, coding collaboration platforms, such as GitHub (https://github.com/), are used where developers **create a repository (repo)** for their project and each developer contributes their own individual codes / components in the repo. An important concept is the notion of a **staging environment (or stage)** and **commits**. Commits are essentially **confirmed submissions of your contribution** to the project, and this can only be done if the commit is done on **the stage, which is a nearly exact replica of the production environment for testing**. It is also good practice to **include a message with your commits with clear explanations on any changes made**, for version tracking. Using collaborative development platforms also has concepts such as **branches** and **pull requests (PR)**, where these are for you to explore and learn in case you do end up as part of a software engineering team in the future: https://product.hubspot.com/blog/git-and-github-tutorial-for-beginners

Testing and Implementation of Software

Software testing methods are traditionally divided into white and black-box testing. These two approaches are used to describe the point of view that the tester takes when designing test cases.

White-box testing is primarily used to test the internal structures or workings of an application. White-box testing requires the testers to have **in-depth knowledge of the source code** being tested, such that the tests exercise every given path off the source codes to the point where the tester knows, for a given test case, knows which line of code is being executed and identify what the correct code should be.

Some testing techniques which fit the white-box testing paradigm include,

- **Control Flow Test**: Tracing how, for some test input, the output / value in memory changes in each line of code.
- **Data Flow Test**: Tracing if data from an external source is read correctly, tracing through how the data changes as the code progresses, and finally if the data writing is also correct.
- **Branches and Paths Test:** Tracing, for different test inputs, whether the input goes into the correct branch of conditional statements. This is usually combined with the idea of boundary test cases from black-box testing, but specific to the conditional statement.

Black-box testing is primarily used to **examine functionalities without any knowledge of the implementation details**. Black-box testing requires the testers to come up with test cases which tests for the behavior of each functionality in a robust manner, **without the need to perform an exhaustive testing from infinite number of test cases**. Hence, some testing cases which fit the black-box testing paradigm includes,

- **Correct use test case**: The test case utilized is within the correct domain for the functionality to work.
- **Incorrect use test case**: The test case utilized is **outside** the correct domain for the functionality to work. This is also used to determine if the program can correctly handle invalid inputs from potential users without crashing.
- **Boundary value test case:** The test case utilized is typically the boundary values of the correct domain for the functionality to work / not work.
- **Stress test case**: The test case usually involves a large volume of input to determine if the program can handle large volumes of input without crashing. This is only applicable if the functionality must process values from an Array / ArrayList data structure or equivalent.

Each type of test (white-box vs black-box) has situations where it is more appropriate to use one over the other:

	White-box Testing	**Black-box Testing**
Appropriate scenarios for testing	• Small fragments of code or modules • "Mission critical" software like managing bank transactions or dealing with national security, where no mistakes or loopholes can be afforded	• Large or integrated software or systems • Internal workings of software or system is not as critical as the end-user experience and correctness of outputs.

Implementation and Maintenance

In this phase, the software or system is deployed to a real-life environment where actual users begin to operate on the system. At certain points of time, the software or system is evaluated to ensure it does not become obsolete or adapts to everchanging regulations and policies. This phase may also involve evaluation of how well it fulfills the initial requirements and objectives set, or if it is ***reliable***, ***fault-tolerant***, and ***functions according to any approved functional requirements***.

To explain why the SDLC is a "cycle", during this phase, issues or areas for improvement or upgrading will be flagged out, and these are put into the planning pipeline for further development or improvements of the software / system, hence going back into the planning phase of the cycle.

Note that while the SDLC may seem like it pertains to software and systems specifically, the big ideas and thinking process can be applied to project or product development in general.

0.2 Data Structures and Algorithm Analysis

Data structures are constructs that can be defined within a programming language to store a collection of data. As one of the key goals when creating a program is to make it efficient, it is vital to create the program such that:

- Data (or a collection of objects) is organized such that it can be accessed efficiently, and the data is, if possible, also memory efficient
- Implemented operations and algorithms to access and manipulate the data are efficient.

Hence, when analysing the "efficiency" of a program, there various measures of efficiency we can utilise:

- **Time** analysis vs. **space** analysis.
- Worst-case, average-case and best-case analyses.

To compare the time efficiency of different algorithms, we utilize a **frequency count**:

```
x = x + y;
```

`x = x + y` run once

```
for i = 1 to n DO
    x = x + y;
```

`x = x + y` run n times

```
for i = 1 to n DO
    for j = 1 to n DO
        x = x + y;
```

`x = x + y` run n^2 times

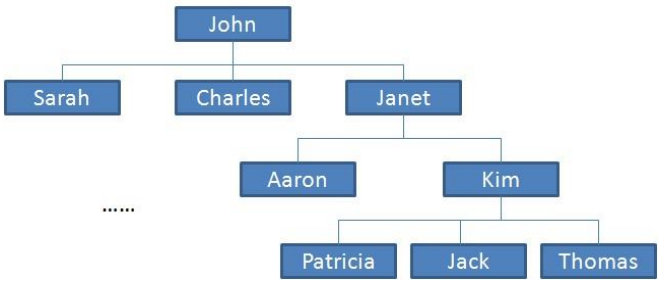
Data structures come in various forms and have their own specific uses and associated algorithms.

Example: Vectors

- Stores objects sequentially in memory
- Can access, change, insert or delete objects
- Algorithms for insert & delete will shift items as needed
- Space: O(n), Access/change = O(1), Insert/delete = O(n)

Example: A database of names with all company's management and employees which can be in the form of a list or make a tree.

name	position
Aaron	Manager
Charles	VP
George	Employee
Jack	Employee
Janet	VP
John	President
...	...



There are several common data structures: arrays, linked lists, queues, stacks, binary trees, hash tables, etc. These data structures can be classified as either ***linear*** or ***nonlinear*** data structures, based on how the data is conceptually organized or aggregated.

- **Linear structures**. The array, list, queue, and stack belong to this category. Each of them is a collection that stores its entries in a linear sequence, and in which entries may be added or removed at will. They differ in the restrictions they place on how these entries may be added, removed, or accessed (FIFO or LIFO for stack and queue respectively)

- **Non-linear structures**. Trees and graphs are classical non-linear structures. Data entries are not arranged in a sequence, but with different rules.

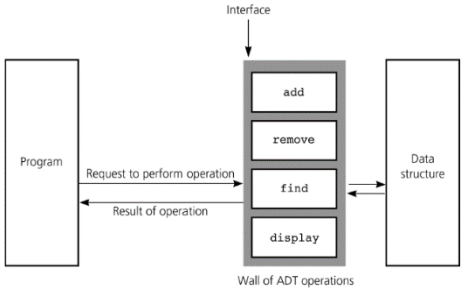
0.3 Abstract Data Types (ADT)

An Abstract Data Type (ADT) is a **collection of data** together with a specification of a set of operations on that collection of data (without the implementation details), whereby data structures are part of an ADT's implementation:

When a program needs data operations that are not directly supported by a language, you need to create your own ADT. You should first design the ADT by carefully **specifying the operations before implementation**.

In essence, we can classify the concept of classes which was covered in OOP as a self-defined ADT, where it contains a collection of primitive or object data types, and the associated methods (or functions), such that the user of the class **only needs to know what a data type can do, but not how it will be implemented**. This brings about the idea of **data abstraction**, where we think in terms of **what** we can do to a collection of data independently of its implementation details.

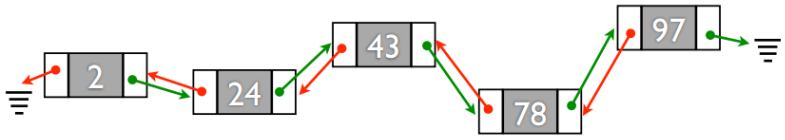
Applying the idea of abstraction to data structures, we have ADT for data structures, where there is a clean separation between interface and implementation details, and the user implements the ADT without tampering with the implementation. Abstraction makes the code robust and easy to maintain, and it may be used multiple times in various contexts. For example, the list ADT may be used directly in application code, or may be used to build another ADT, such as a stack.



In an ADT, for example a list, the “ordering” to store the data must be well-defined. The ordering principles for ADTs are typically one of the following:

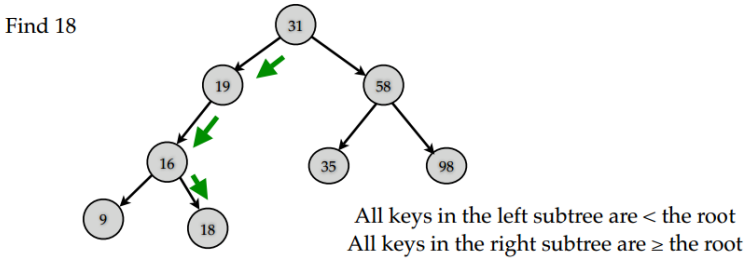
- **Natural Ordering:**
  - Keys are put in some order (numerical or alphabetical) so that we know something about where each key is relative to the other keys.

- **Linking:**
  - Pointers added to each record to find related records quickly, and express relationships between pieces of information.



  - No restriction on how records are located in physical memory
  - Insertion & deletion requires a **traversal** and depending on implementation, will be easier if a pointer is maintained somewhere in the middle.
  - Size of data does not need to be known and such a structure allows for a dynamically sized structure.

- **Partitioning:**
  - Divide the records into 2 or more groups, each group sharing a particular property.
  - Ordering implicitly gives a partitioning based on the “less than” relation.
  - Partitioning usually combined with linking to link records to their supposed parent.
  - Trees follow such a partitioning principle.



In summary:

- Data storage & operations are encapsulated by an ADT.
- ADT specifies permitted operations as well as time and space guarantees.
- User does not need to know the implementation details (but we are concerned with mplementation in this module).
- The implementation of the ADT requires a well-defined ordering describing the relationship between records.

How do I choose the right data structures?

When writing a program, one of the first steps is determining or choosing the data structures. What are the "right" data structures for the program? The **interface of operations** supported by a data structure is one factor to consider when choosing between several available data structures. Another important factor is the **efficiency** of the data structure: how much *space* does the data structure occupy, and what are the *running times* of the operations in its interface?

0.4 Useful Concepts for Module: Java Generics and Collection Framework

Java Generics

Suppose you want to create a new class that encapsulates a queue of circles. You wrote:

```
1  public class CircleQueue {
2      private Circle[] circles;
3      :
4      public CircleQueue(int size) {...}
5      public boolean isFull() {...}
6      public boolean isEmpty() {...}
7      public void enqueue(Circle c) {...}
8      public Circle dequeue() {...}
9  }
```

Later, you found that you need a new class that encapsulates a queue of points. You wrote:

```
1  public class PointQueue {
2     private Point[] points;
3     :
4     public PointQueue(int size) {...}
5     public boolean isFull() {...}
6     public boolean isEmpty() {...}
7     public void enqueue(Point p) {...}
8     public Point dequeue() {...}
9  }
```

You realize that there are actually a lot of similar code. Invoking the *abstraction principle*, which states that *"Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts*", you decided to create a queue of Objects to replace the two classes above.

```
1  public class ObjectQueue {
2      private Object[] objects;
3      :
4      public ObjectQueue(int size) {...}
5      public boolean isFull() {...}
6      public boolean isEmpty() {...}
7      public void enqueue(Object o) {...}
8      public Object dequeue() {...}
9  }
```

Now you have a generic class, that you can use to store objects of any kind, including a queue of Strings, a queue of Colors, etc. The early Java collection library contains many such generic data structures that stores elements of type `Object`. To create a queue of 10 circles, you need:

```
1  ObjectQueue cq = new ObjectQueue(10);
2  cq.enqueue(new Circle(new Point(0, 0), 10));
3  cq.enqueue(new Circle(new Point(1, 1), 5));
4   :
```

Getting a circle out of the queue is a bit more troublesome: **`Circle c = cq.dequeue();`**

The code would generate a compilation error, since we are trying to perform a narrowing reference conversion; we cannot assign a variable of type `Object` to type `Circle` without type casting: **`Circle c = (Circle)cq.dequeue();`**

The code above might generate a runtime `ClassCastException` if there is an object in the queue that is not `Circle` or its subclass. To avoid runtime error, we should check the type first:

```
1  Object o = cq.dequeue();
2  if (o instanceof Circle) { Circle c = (Circle)o; }
```

To avoid the need to check the object type all the time, we require **generics**, which allows a *generic class* or a *generic interface* of some type T to be written:

```
1  public class Queue<T> {
2     private T[] objects;
3     :
4     public Queue(int size) {...}
5     public boolean isFull() {...}
6     public boolean isEmpty() {...}
7     public void enqueue(T o) {...}
8     public T dequeue() {...}
9  }
```

T is known as *type parameter*. The same code as before can be written as:

```
1  Queue<Circle> cq = new Queue<Circle>(10);
2  cq.enqueue(new Circle(new Point(0, 0), 10));
3  cq.enqueue(new Circle(new Point(1, 1), 5));
4  Circle c = cq.dequeue();
```

Here, we pass `Circle` as the *type argument* to T, creating a *parameterized type* `Queue<Circle>`. In Line 4, we no longer need to cast, and there is no danger of runtime error due to an object of the wrong class being added to the queue, for doing the following will generate a compile-time error.

```
1  Queue<Circle> cq = new Queue<Circle>(10);
2  cq.enqueue(new Point(1, 3));
```

Generic typing is a type of polymorphism as well and is also known as *parametric polymorphism*. Recall that you have used generics in OOP1, for example, the **`ArrayList`**

```
ArrayList<Student> list = new ArrayList<Student>();
```

Java Collection Framework

One of the basic interfaces in Java Collection Framework is `Collection<E>`, it looks like:

```
1  public interface Collection<E> extends Iterable<E> {
2      boolean add(E e);
3      boolean contains(Object o);
4      boolean remove(Object o);
5      void clear();
6      boolean isEmpty();
7      int size();
8
9      Object[] toArray();
10     <T> T[] toArray(T[] a);
11
12     boolean addAll(Collection<? extends E> c);
13     boolean containsAll(Collection<?> c);
14     boolean removeAll(Collection<?> c);
15     boolean retainAll(Collection<?> c);
16     :
17 }
```

Like a generic class that you have seen, `Collection` is a *generic interface* parameterized with a type parameter E. It extends a generic `Iterable<E>` interface.

The first six methods of `Collection<E>` should be self-explanatory:

- add adds an element into the collection;
- contains checks if a given object is in the collection;
- remove removes a single instance of the given object from the collection;
- clear removes all objects from the collection;
- isEmpty()checks if the collection has no elements or not; and finally,
- size returns the number of elements.

Note that `contains()` relies on the implementation of `equals()` to check if the object exists in the collection or not. Similarly, `remove()` relies on `equals()` to find the matching objects.

You might notice that, instead of `contains(E e)` and `remove(E e)`, the `Collection` interface uses `contains(Object o)` and `remove(Object o)`. This little inconsistency, however, is harmless. For instance, if you have a collection intended for circles only, adding a non-Circle could be disastrous. Trying to remove a non-Circle or checking for a non-Circle, would just return false.

The method `toArray()` on Line 9 returns an array containing all the elements inside this collection. The second overloaded `toArray` method takes in an array of generic type `T`. If the collections fit in `a`, `a` is filled and returned. Else, it allocates a new array of type `T` and returned. The second `toArray` method is an example of a generic method. It is the caller responsibility to pass in the right type, otherwise, an `ArrayStoreException` will be thrown.

`addAll` adds all the elements of collection `c` into the current collection;
`containsAll` checks if all the elements of collection `c` are contained in the current collection;
`removeAll` removes all elements from collection `c`;
`retainsAll` remove all elements not in `c`.

We note that in the above four methods mentioned, the collection `c` has the type **`Collection<?>`**. **`Collection<?>`** is a supertype of `Collection<T>`, whatever `T` is. So, we can pass in a Collection of any reference type to check for equality. In this case, we are not adding anything to the collection, so the type of `c` is set to be as general as possible.

In `addAll`, however, `c` is declared as **`Collection<? extends E>`**. Since we are adding to the collection, we can only add elements that either has the type `E` or a type that is a subtype `E`.

0.5 Useful Concepts for Module: Comparable and Comparator Interface

Comparable Interface

The **`Comparable`** interface defines the **`compareTo`** method for comparing objects. Suppose you want to design a generic method to find the larger of two objects of the same type. To accomplish this, the two objects must be comparable, so the common behaviour for the objects must be comparable. Java provides the **`Comparable`** interface for this purpose.

```
1 public interface Comparable<E> {
2       public int compareTo(E o);
3 }
```

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's **natural ordering**, and the class's **`compareTo`** method is referred to as its ***natural comparison*** *method*.

You may refer to this link:
https://docs.oracle.com/javase/tutorial/collections/interfaces/order.html

The **`compareTo`** method determines the order of this object with the specified object **`o`** and returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

The **`Comparable`** interface is a generic interface. The generic type **`E`** is replaced by a concrete type when implementing this interface. Many classes in the Java library implement **`Comparable`** to define a natural order for objects. For example:

```
1  public class Integer extends Number implements Comparable<Integer> {
2      // class body omitted
3      @Override
4      public int compareTo(Integer o) { // Implementation omitted }
5  }
6
7  public class String extends Object implements Comparable<String> {
8      // class body omitted
9      @Override
10     public int compareTo(String o) {  // Implementation omitted }
11 }
```

Array of objects that implement this interface can be sorted automatically by **`Arrays.sort()`**:

```
1  import java.util.*;
2  public class SortString {
3      public static void main(String[] args){
4          String[] names={"Jaren", "Maria", "Star", "Dot","Com"};
5          System.out.println("Before sorting\n");
6
7          for(String s: names) System.out.println(s);
8
9          System.out.println("After sorting\n");
10         Arrays.sort(names);
11         for(String s: names) System.out.println(s);
12     }
13 }
```

Arraylist of objects that implement this interface can be sorted by **`Collections.sort()`**.
An example is shown in the next page.

```
1   import java.util.*;
2   import java.text.*;
3   public class SortDate {
4       public static void main(String[] args) {
5           DateFormat formatter = new SimpleDateFormat("dd-MMM-yy");
6
7           ArrayList<Date> dateList = new ArrayList<Date>();
8           try{
9               dateList.add((Date)formatter.parse("29-AUG-2008"));
10              dateList.add((Date)formatter.parse("13-FEB-2007"));
11              dateList.add((Date)formatter.parse("29-JAN-2010"));
12              dateList.add((Date)formatter.parse("29-MAR-2001"));
13          } catch(ParseException ex){}
14
15          System.out.println(dateList);
16
17          Collections.sort(dateList); //sorted in chronological order
18          System.out.println(dateList);
19
20      }
21  }
```

Consider sorting a list of **`Employee`** objects using Comparable interface.

```
1   public class Employee {
2       private int empId;
3       private String name;
4       private int age;
5
6       public Employee(int empId, String name, int age) {
7           // set values on attributes
8       }
9       // getters & setters
10  }
```

`Employee`'s natural ordering would be done according to the employee Id. For that, the above **`Employee`** class must be altered to **add the comparing ability** as follows.

```
1   public class Employee implements Comparable<Employee> {
2       private int empId;
3       private String name;
4       private int age;
5
6       /**
7       * Compare a given Employee with this object.
8       * If employee id of this object is
9       * greater than the received object,
10      * then this object is greater than the other.
11      */
12      public int compareTo(Employee o) {
13          return this.empId - o.empId ;
14      }
15      // ….
16  }
```

The new **`compareTo()`** method does the trick of implementing the natural ordering of the instances. If a collection of **`Employee`** objects is sorted using **`Collections.sort(ArrayList)`** method; sorting happens according to the ordering done inside this method.

Comparator Interface

If we need to sort using other fields of the **Employee**, we'll have to change the Employee class's **compareTo()** method to use those fields. But then we'll lose this **empId** based sorting mechanism. This is not a good alternative if we need to sort using different fields at different occasions. But no need to worry, **Comparator** is there to save us.

The Comparator interface is defined as follows:

```
1  public interface Comparator<T> {
2      int compare(T o1, T o2);
3  }
```

By writing a class that implements the **java.util.Comparator** interface, you can sort **Employee** objects using any field as you wish even without touching the **Employee** class itself. **Employee** class does not need to implement **java.lang.Comparable** or **java.util.Comparator** interface.

Let's now define a **EmpSortByName** class which will be used to sort **Employee** instances according to the **name** field. In this class, the **compare()** method sorting mechanism is implemented. In the **compare()** method we get two **Employee** instances and we have to return which object is greater.

```
1  public class EmpSortByName implements Comparator<Employee>{
2
3      public int compare(Employee o1, Employee o2) {
4          return o1.getName().compareTo(o2.getName());
5      }
6  }
```

To test this sorting mechanism, you must use the **Collections.sort(List, Comparator)** method instead of **Collections.sort(List)** method:

```
1  import java.util.*;
2
3  public class TestEmployeeSort {
4      public static void main(String[] args) {
5
6          List coll = Util.getEmployees();
7          //use Comparator implementation
8          Collections.sort(coll, new EmpSortByName());
9          printList(coll);
10     }
11
12     private static void printList(List<Employee> list) {
13         System.out.println("EmpId\tName\tAge");
14         for (Employee e: list) {
15             System.out.println(e.getEmpId() + "\t" + e.getName() + "\t" +
16                                e.getAge());
17         }
18     }
19 }
```