Introduction to Systems Programming

Interfaces

2025 - Simón Gutiérrez Brida *(Based on material by Dr. Nazareno Aguirre)*

Revisiting types

A type defines a set of values, an what operations can be applied to those values.

The type itself doesn't define the operations, it defines which operations can be applied to values of its type.

Examples: natural numbers, integer numbers, real numbers, complex numbers, etc.

Revisiting types

A type in OOP defines a set of values, and also the operations that can be applied to those values.

The type defines both the values, and their representation, and the operations, and their implementations.

Examples: LinkedList, ArrayList, etc.

Revisiting types

An Abstract Data Type is a description of values and operations on those values, without defining the technical details on how those values are represented and how the operations work on those representations

- *Me*

Abstract Data Types

Lists

A List is a linearly organized collection of values, its main operations are:

- Creation
- Insertion/Deletion/Retrieval
- Properties about a list: empty, size, contains.

Abstract Data Types

Sets

A Set is an unordered collection of different elements, its main operations are:

- Creation
- Insertion/Deletion
- Union/Intersection
- Properties about a list: empty, size, contains, is a sub set.

Abstract Data Types

Stacks

A Stack is a linearly organized collection of values (similar to a list), it's a FILO collection (First In, Last Out), its main operations are:

- Creation
- Push/Pop
- Properties about a list: empty, size

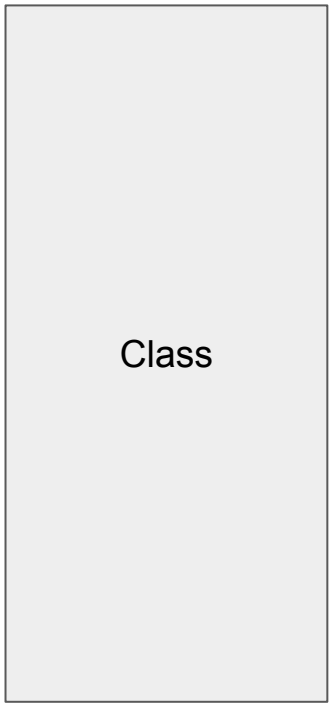
Abstract Data Types

Queues

A Queue is a linearly organized collection of values (similar to a list), it's a FIFO collection (First In, First Out), its main operations are:

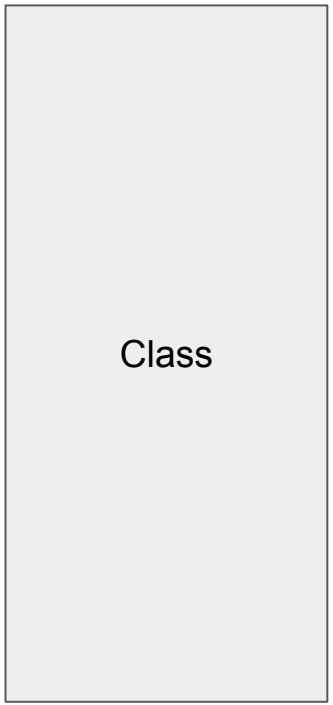
- Creation
- Enqueue/Dequeue
- Properties about a list: empty, size

Data Abstractions in Object Oriented Languages



- Defines a new type
- Operations (both declarations and definitions)
  - Create/Destroy Class values
  - Modify Class values
  - Check properties in Class values

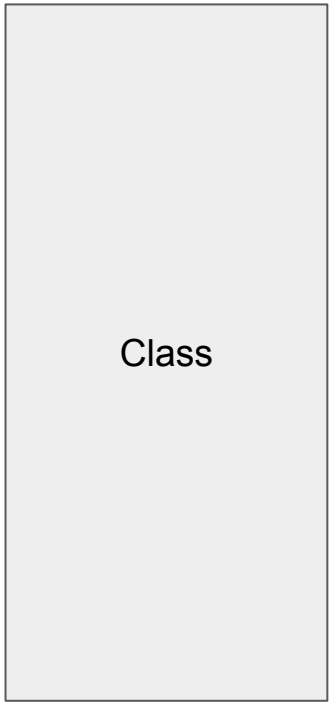
Data Abstractions in Object Oriented Languages



- Defines a new type
- Operations (both declarations and definitions)
  - Create/Destroy Class values
  - Modify Class values
  - Check properties in Class values

These are called, objects, they are instances of a class.

Data Abstractions in Object Oriented Languages



- Defines a new type
- Operations (both declarations and definitions)
  - Create/Destroy Class values
  - Modify Class values
  - Check properties in Class values

These are called, objects, they are instances of a class.

*Follow formal conventions, usually **ClassName(arguments)***

Data and operations are not separated, each object encapsulates both data and operations.

Data Abstractions in Object Oriented Languages

Class

- Defines a new type
- Operations (both declarations and definitions)
  - Create/Destroy Class values
  - Modify Class values
  - Check properties in Class values

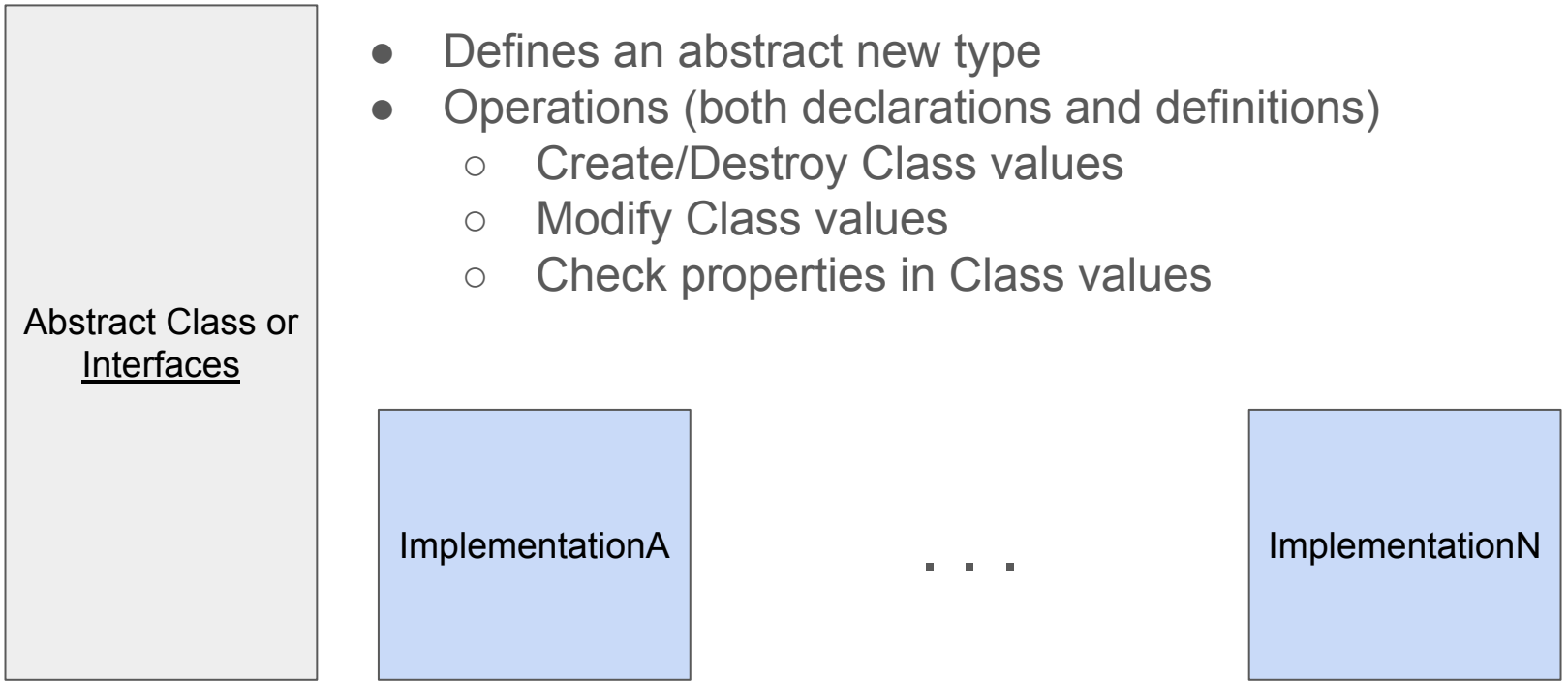
Usually memory management is automatic, we don't need to destroy values.

These are called, objects, they are instances of a class.

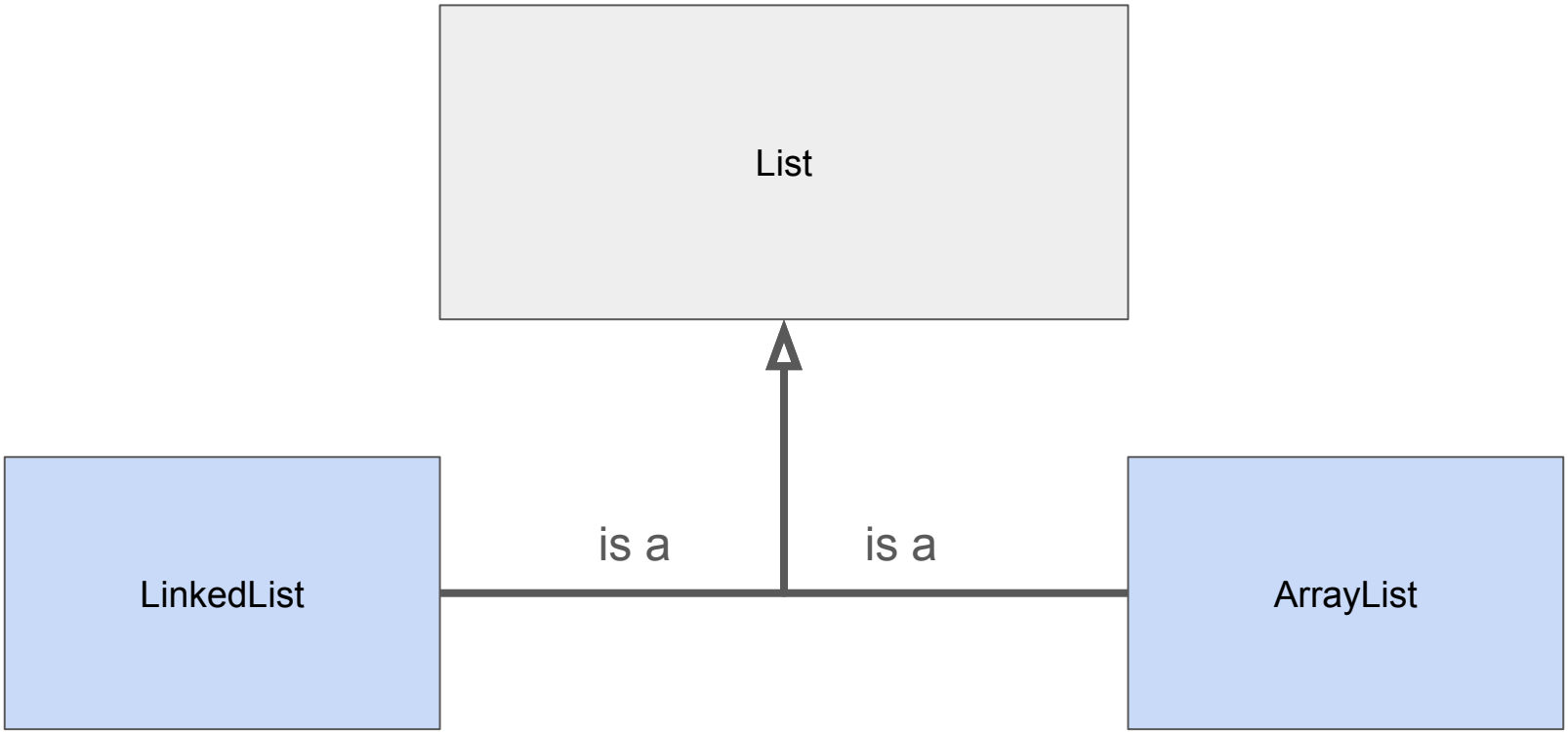
*Follow formal conventions, usually **ClassName(arguments)***

Data and operations are not separated, each object encapsulates both data and operations.

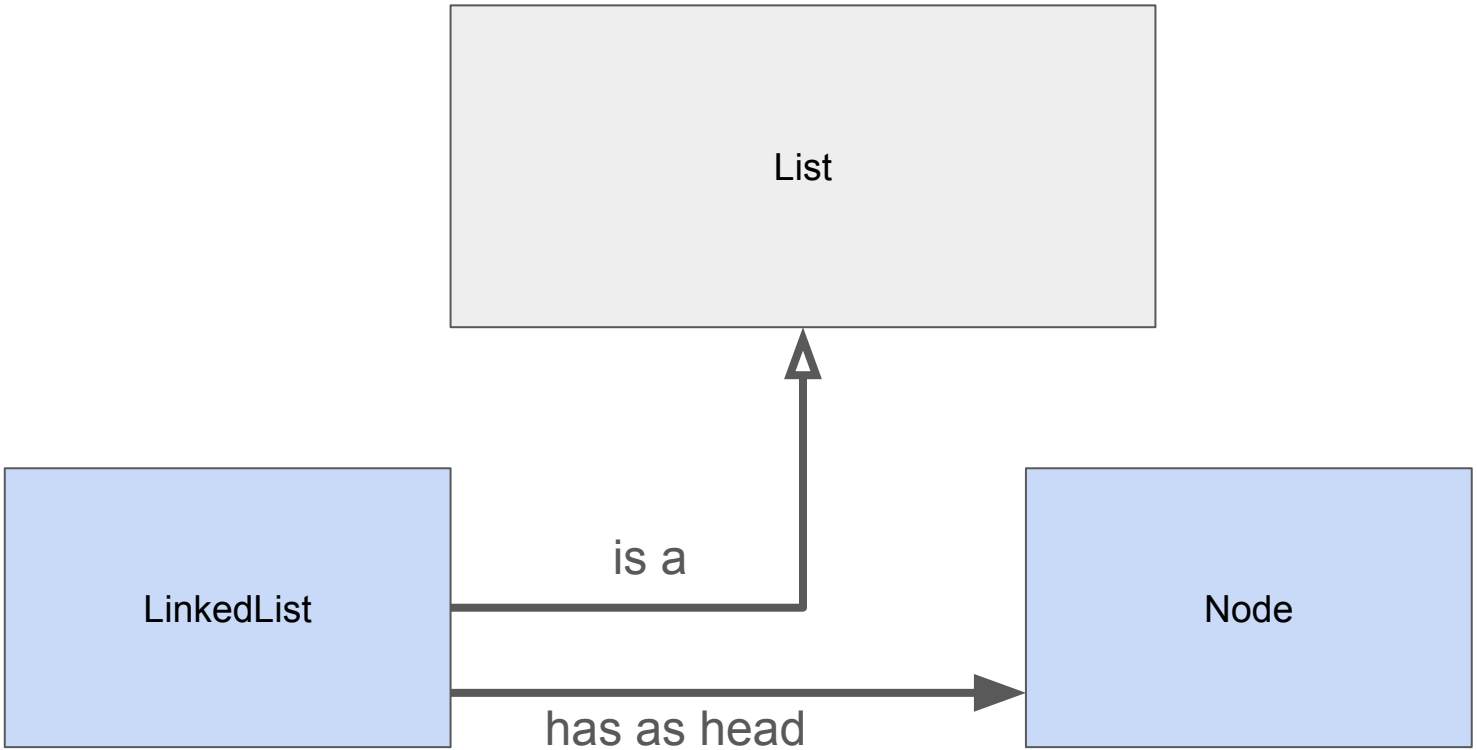
Data Abstractions in Object Oriented Languages



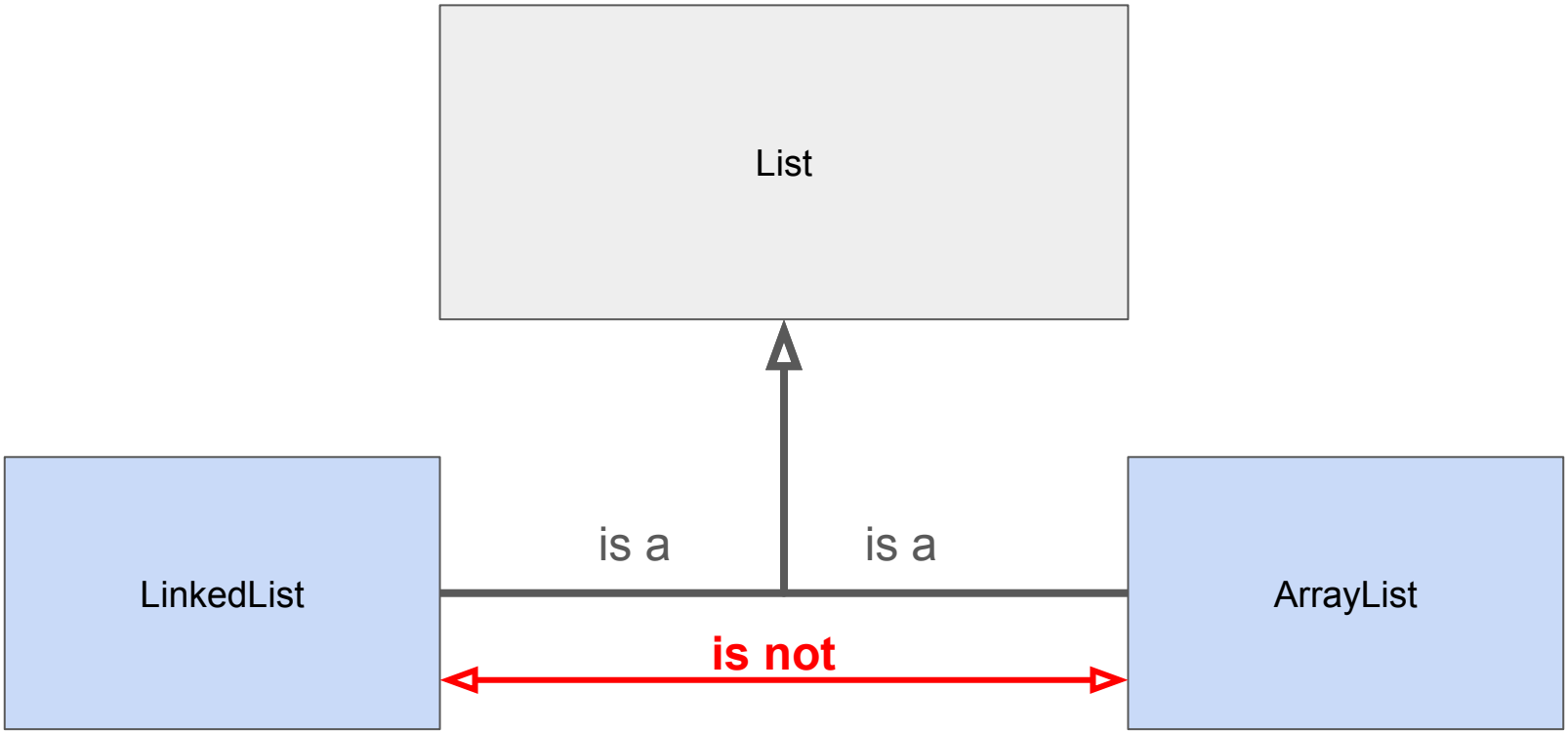
Abstract Data Types in OOP



Abstract Data Types in OOP



An interface and an implementation are both types



Interfaces as Types

An interface in Java allows to define a new Type, while only providing a description but without providing a specific representation; and the operations applied to values of that Type, without providing any details on implementation.

- Implementing classes are subtypes of the interface type.
- So, polymorphism is available with interfaces as well as classes.

Interfaces as Specifications

- Strong separation of functionality from implementation.
  - Though parameter and return types are mandated.

- Clients interact independently of the implementation.
  - But clients can choose from alternative implementations.

Interfaces in Java

An interface in Java allows to define a new Type, while only providing a description but without providing a specific representation; and the operations applied to values of that Type, without providing any details on implementation.

- An interface cannot have instances.
- An interface cannot define/declare constructors.
- An interface cannot define/declare fields (*).
- An interface cannot define methods (*).
- An interface can declare public methods (*).

(*) Several things have changed since Java 9, I recommend staying away from new features until the basic ones are fully understood.

Seeing a class through an Interface (1)

```java
package laboratory;

import java.util.ArrayList;
import laboratory.data.Student;
import laboratory.data.Teacher;
//could use laboratory.data.* to import all classes

/**
* <Class documentation>
*/
public class Laboratory {

 private ArrayList<Student> students;
 private Teacher teacher;
 private String name;

 /**
 * <Constructor documentation>
 */
 public Laboratory(String name, Teacher teacher) {
  //<implementation>
 }

 //More methods

 /**
 * <Method documentation>
 */
 public int numberOfStudents() {
  return students.size();
 }
}
```

The interface of the class

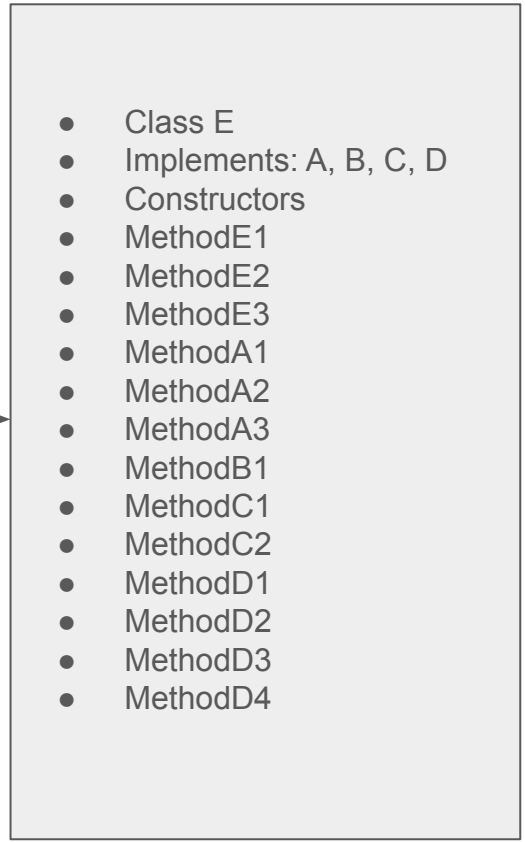
```java
package laboratory;

import java.util.ArrayList;
import laboratory.data.Student;
import laboratory.data.Teacher;

/**
* <Class documentation>
*/
public class Laboratory {

 /**
 * <Constructor documentation>
 */
 public Laboratory(String name, Teacher teacher)

 //More methods (documentation and signatures)

 /**
 * <Method documentation>
 */
 public int numberOfStudents()
}
```

Seeing a class through an Interface (2)

- Class E
- Implements: A, B, C, D
- Constructors
- MethodE1
- MethodE2
- MethodE3
- MethodA1
- MethodA2
- MethodA3
- MethodB1
- MethodC1
- MethodC2
- MethodD1
- MethodD2
- MethodD3
- MethodD4

The interface of the class

As E



Seeing a class through an Interface (3)

- Class E
- Implements: A, B, C, D
- Constructors
- MethodE1
- MethodE2
- MethodE3
- MethodA1
- MethodA2
- MethodA3
- MethodB1
- MethodC1
- MethodC2
- MethodD1
- MethodD2
- MethodD3
- MethodD4

- Class E
- MethodA1
- MethodA2
- MethodA3

Seeing a class through an Interface (4)

- Class E
- Implements: A, B, C, D
- Constructors
- MethodE1
- MethodE2
- MethodE3
- MethodA1
- MethodA2
- MethodA3
- MethodB1
- MethodC1
- MethodC2
- MethodD1
- MethodD2
- MethodD3
- MethodD4

The interface of the class

As B

- Class E
- MethodB1

Seeing a class through an Interface (5)

Seeing a class through an Interface (6)

- Class E
- Implements: A, B, C, D
- Constructors
- MethodE1
- MethodE2
- MethodE3
- MethodA1
- MethodA2
- MethodA3
- MethodB1
- MethodC1
- MethodC2
- MethodD1
- MethodD2
- MethodD3
- MethodD4

The interface of the class

As D

- Class E
- MethodD1
- MethodD2
- MethodD3
- MethodD4

Demo

Looking at LinkedList with different interfaces

It is possible for one class to implement several interfaces. We will start by looking at Java's own LinkedList, and several interfaces implemented by it.

Demo

Making our own ADT and implementation

The implementation of an ADT might not be different than the implementation of another. A LinkedList can be used to implement Lists, Sets, Collections (*), Queues, Stacks, Deques, etc.

** Consider methods map, all, and any.*

Introduction to Systems Programming

Function overloading

Several function names with different arguments

In a lot of cases, we have the same function/operation for different inputs (quantity or types).

For example, a max function: max(a, b) will return a if a is greater than b, according to some criteria.

Let's explore some of these cases...

Several function names with different arguments

- max(char, char), what should this do?
- max(int, int), what should this do?
- max(String, String), what should this do?
- max(Student, Student), what should this do?
- Could we abstract all of these using interfaces and generics?

We have seen things like this before

```
1 + 3
"Hello" + " World!"
35 + " seconds"
3.5f + 4
```

This is an example of an operator overloading. But we could write this as a function/method.

We have seen things like this before

```
1 + 3
"Hello" + " World!"
35 + " seconds"
3.5f + 4
```

```
public Object add(int a, int b) {
  return new Integer(a + b);
}

public Object add(String a, String b) {
  return a.concact(b);
}

public Object add(int a, String b) {
  return String.valueOf(a).concact(b);
}

public Object add(float a, int b) {
  return new Float(a + ((float) b));
}
```

We have seen things like this before

```
1 + 3
"Hello" + " World!"
35 + " seconds"
3.5f + 4
```

This is in fact a very artificial example, we are abusing returning Object to be able to return anything as an Object.

```
public Object add(int a, int b) {
  return new Integer(a + b);
}
```

```
public Object add(String a, String b) {
  return a.concact(b);
}
```

```
public Object add(int a, String b) {
  return String.valueOf(a).concact(b);
}
```

```
public Object add(float a, int b) {
  return new Float(a + ((float) b));
}
```

Demo

Making a demo for the previous add example

- max(char, char), what should this do?
- max(int, int), what should this do?
- max(String, String), what should this do?
- max(Student, Student), what should this do?
- Could we abstract all of these using interfaces and generics?