AUBURN

UNIVERSITY

SAMUEL GINN
COLLEGE OF ENGINEERING

Collections

COMP 2210 – Dr. Hendrix

Collections

Collections

*In computer science, a **collection** or container is a grouping of some variable number of data items (possibly zero) that have some shared significance to the problem being solved and need to be operated upon together in some controlled fashion. Generally, the data items will be of the same type or, in languages supporting inheritance, derived from some common ancestor type. A collection is a concept applicable to abstract data types, and does not prescribe a specific implementation as a concrete data structure, though often there is a conventional choice; see container (type theory) for type theory discussion.*

http://en.wikipedia.org/wiki/Collection_(computing)

Collections

*In computer science, a **collection** or container is a grouping of some variable number of data items (possibly zero) that have some shared significance to the problem being solved and need to be operated upon together in some controlled fashion. Generally, the data items will be of the same type or, in languages supporting inheritance, derived from some common ancestor type. A collection is a concept applicable to abstract data types, and does not prescribe a specific implementation as a concrete data structure, though often there is a conventional choice; see container (type theory) for type theory discussion.*

http://en.wikipedia.org/wiki/Collection_(computing)

Common collections: Bag, Set, List, Stack, Queue, Priority Queue, Map

General ways of organizing data and providing controlled access to that data.

Collections

*In computer science, a **collection** or container is a grouping of some variable number of data items (possibly zero) that have some shared significance to the problem being solved and need to be operated upon together in some controlled fashion. Generally, the data items will be of the same type or, in languages supporting inheritance, derived from some common ancestor type. A collection is a concept applicable to abstract data types, and does not prescribe a specific implementation as a concrete data structure, though often there is a conventional choice; see container (type theory) for type theory discussion.*

`http://en.wikipedia.org/wiki/Collection_(computing)`

Common collections: Bag, Set, List, Stack, Queue, Priority Queue, Map

General ways of organizing data and providing controlled access to that data.

Common data structures: Array, linked list, tree, heap, hash table

Specific ways of storing and connecting data in a program.

Collections

*In computer science, a **collection** or container is a grouping of some variable number of data items (possibly zero) that have some shared significance to the problem being solved and need to be operated upon together in some controlled fashion. Generally, the data items will be of the same type or, in languages supporting inheritance, derived from some common ancestor type. A collection is a concept applicable to abstract data types, and does not prescribe a specific implementation as a concrete data structure, though often there is a conventional choice; see container (type theory) for type theory discussion.*

http://en.wikipedia.org/wiki/Collection_(computing)

Common collections: Bag, Set, List, Stack, Queue, Priority Queue, Map

General ways of organizing data and providing controlled access to that data.

Common data structures: Array, linked list, tree, heap, hash table

Specific ways of storing and connecting data in a program.

Abstract Data Type: A specific way of providing a collection within a given programming language

```
public class ArrayList<T> implements List<T> { . . . }
```

Collections

*In computer science, a **collection** or container is a grouping of some variable number of data items (possibly zero) that have some shared significance to the problem being solved and need to be operated upon together in some controlled fashion. Generally, the data items will be of the same type or, in languages supporting inheritance, derived from some common ancestor type. A collection is a concept applicable to abstract data types, and does not prescribe a specific implementation as a concrete data structure, though often there is a conventional choice; see container (type theory) for type theory discussion.*

http://en.wikipedia.org/wiki/Collection_(computing)

Common collections: Bag, Set, List, Stack, Queue, Priority Queue, Map

General ways of organizing data and providing controlled access to that data.

Common data structures: Array, linked list, tree, heap, hash table

Specific ways of storing and connecting data in a program.

These terms are commonly used and misused interchangeably.

Abstract Data Type: A specific way of providing a collection within a given programming language

```
public class ArrayList<T> implements List<T> { . . . }
```

Java Collections Framework

https://docs.oracle.com/javase/8/docs/technotes/guides/collections/

The Java platform includes a collections framework. A collection is an object that represents a group of objects (such as the classic Vector class). A collections framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details.

The primary advantages of a collections framework are that it:

- ***Reduces programming effort** by providing data structures and algorithms so you don't have to write them yourself.*
- ***Increases performance** by providing high-performance implementations of data structures and algorithms. Because the various implementations of each interface are interchangeable**, programs can be tuned** by switching implementations.*
- ***Provides interoperability** between unrelated APIs by establishing a common language to pass collections back and forth.*
- ***Reduces the effort required to learn APIs** by requiring you to learn multiple ad hoc collection APIs.*
- ***Reduces the effort required to design and implement APIs** by not requiring you to produce ad hoc collections APIs.*
- ***Fosters software reuse** by providing a standard interface for collections and algorithms with which to manipulate them.*

http://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html

Java Collections Framework

- The JCF consists of
  - **Collection interfaces** - interfaces that define different types of collections such as lists and sets.
    - These interfaces form a hierarchy with `java.util.Collection` as the root.
    - There is a related interface hierarchy rooted at `java.util.Map`.
  - **General-purpose implementations** - classes that provide the primary implementations of the collection interfaces.
  - And lots of other things …

Java Collections Framework

- The JCF consists of
  - **Collection interfaces** - interfaces that define different types of collections such as lists and sets.
    - These interfaces form a hierarchy with `java.util.Collection` as the root.
    - There is a related interface hierarchy rooted at `java.util.Map`.
  - **General-purpose implementations** - classes that provide the primary implementations of the collection interfaces.
  - And lots of other things …

The basis of the JCF are the interfaces.

Sample JCF Interfaces

- **Collection** - A group of objects. No assumptions are made about the order of the collection (if any) or whether it can contain duplicate elements.
- **Set** - The familiar set abstraction. No duplicate elements permitted. May or may not be ordered. Extends the Collection interface.
- **List** - Ordered collection, also known as a sequence. Duplicates are generally permitted. Allows positional access. Extends the Collection interface.
- **Queue** - A collection designed for holding elements before processing. Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations.
- **Deque** - A double ended queue, supporting element insertion and removal at both ends. Extends the Queue interface.
- **Map** - A mapping from keys to values. Each key can map to one value.
- **SortedSet** - A set whose elements are automatically sorted, either in their natural ordering (see the Comparable interface) or by a Comparator object provided when a SortedSet instance is created. Extends the Set interface.
- **SortedMap** - A map whose mappings are automatically sorted by key, either using the natural ordering of the keys or by a comparator provided when a SortedMap instance is created. Extends the Map interface.
- **NavigableSet** - A SortedSet extended with navigation methods reporting closest matches for given search targets. A NavigableSet may be accessed and traversed in either ascending or descending order.

Why interfaces?

Why specify an interface and then a class that implements it? Why not just write the class?

Why interfaces?

Why specify an interface and then a class that implements it? Why not just write the class?

Interfaces provide the best mechanism in Java for decoupling a specification from its (various) implementations.

Why interfaces?

Why specify an interface and then a class that implements it? Why not just write the class?

Interfaces provide the best mechanism in Java for decoupling a specification from its (various) implementations.

```
public class Client {

   CollectionInterface<MyType> c = new ImplementingClass<MyType>();

}
```

Why interfaces?

Why specify an interface and then a class that implements it? Why not just write the class?

Interfaces provide the best mechanism in Java for decoupling a specification from its (various) implementations.

```
public class Client {

   CollectionInterface<MyType> c = new ImplementingClass<MyType>();

}
```

The only spot in the client that depends in any way on the collection implementation.

Why interfaces?

Why specify an interface and then a class that implements it? Why not just write the class?

Interfaces provide the best mechanism in Java for decoupling a specification from its (various) implementations.

```
public class Client {

   CollectionInterface<MyType> c = new ImplementingClass<MyType>();

}
```

The only spot in the client that depends in any way on the collection implementation.

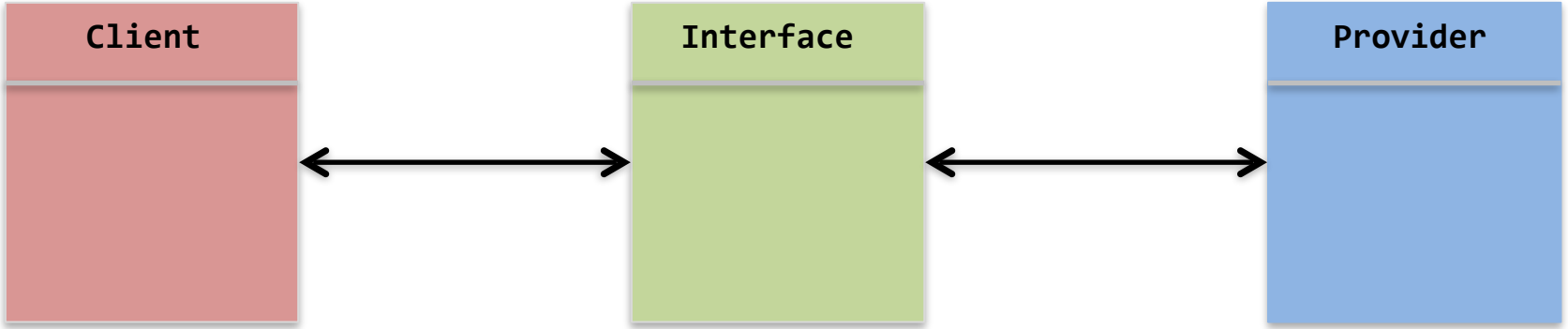
The client is only dependent on the abstract behavior described in the interface. Any class that provides this behavior will work.

Why interfaces?

An interface defines a contract between a client and a provider.

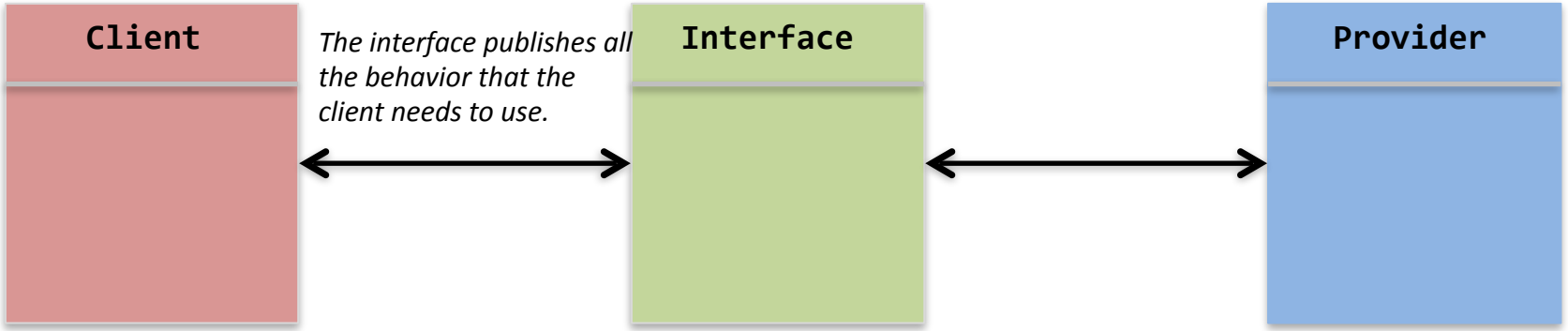
Why interfaces?

An interface defines a contract between a client and a provider.



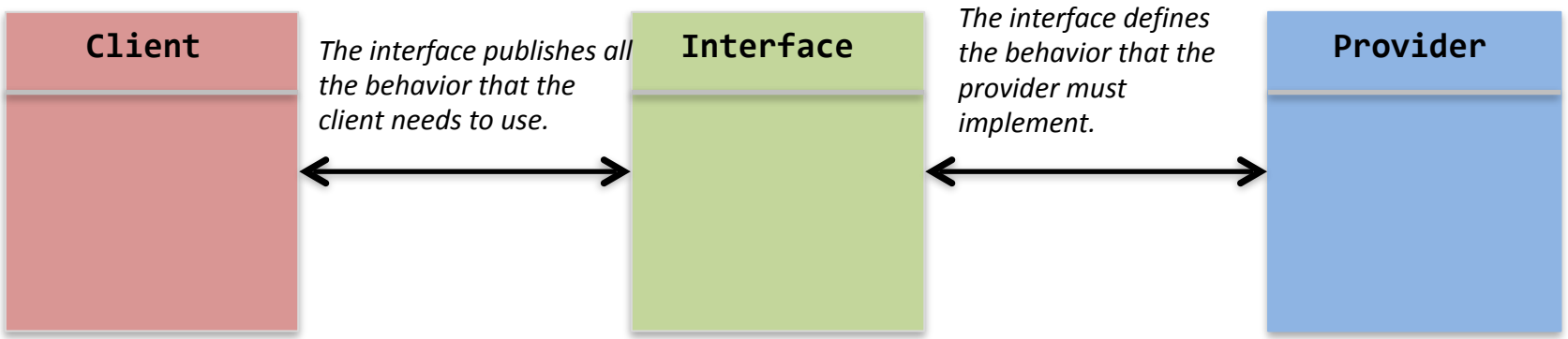
Why interfaces?

An interface defines a contract between a client and a provider.



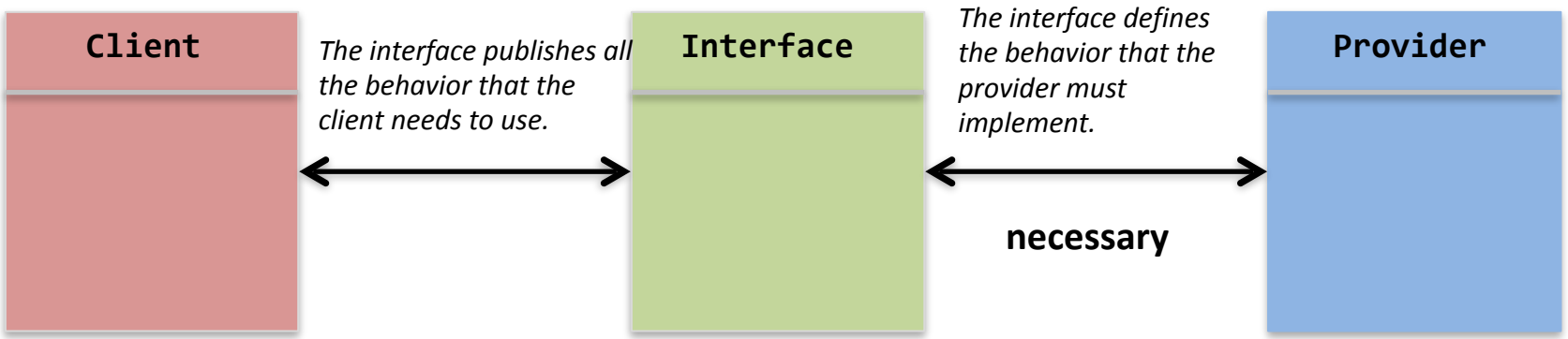
Why interfaces?

An interface defines a contract between a client and a provider.



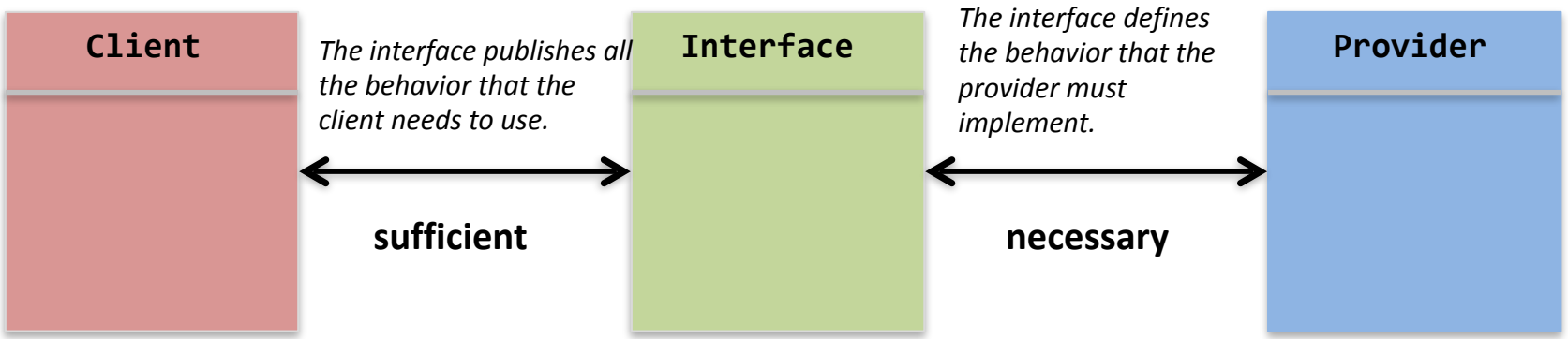
Why interfaces?

An interface defines a contract between a client and a provider.



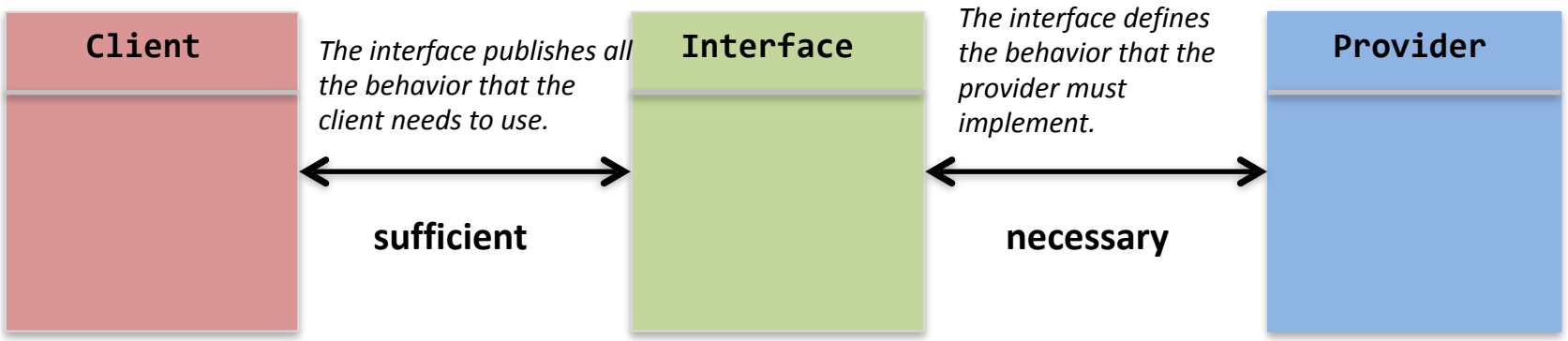
Why interfaces?

An interface defines a contract between a client and a provider.



Why interfaces?

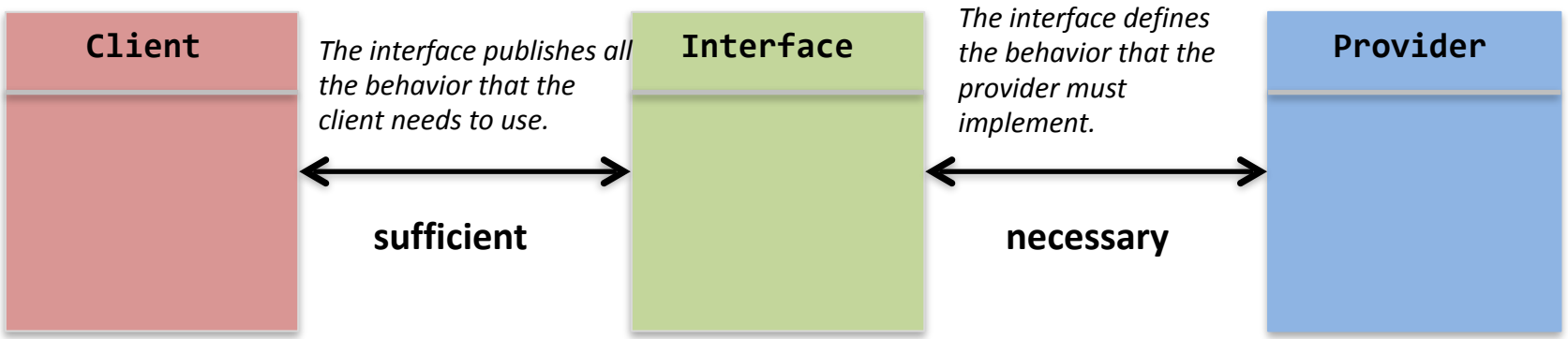
An interface defines a contract between a client and a provider.



This contract relationship between the client and provider is the basis of making client code dependent only on the “what” and not the “how.”

Why interfaces?

An interface defines a contract between a client and a provider.

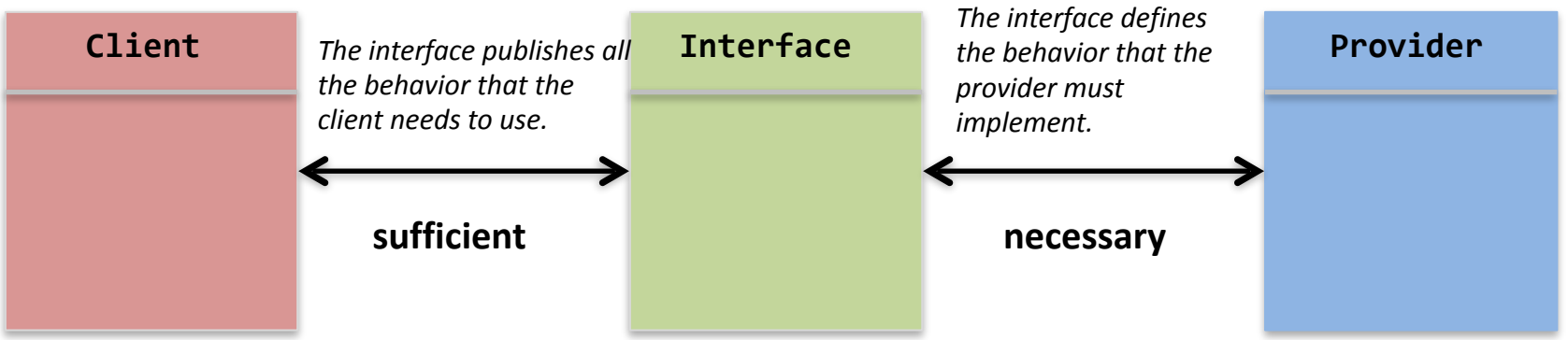


This contract relationship between the client and provider is the basis of making client code dependent only on the “what” and not the “how.”

```
public class Client {

   CollectionInterface<MyType> c = new ImplementingClass<MyType>();

}
```

Why interfaces?

An interface defines a contract between a client and a provider.



This contract relationship between the client and provider is the basis of making client code dependent only on the “what” and not the “how.”

```
public class Client {

   CollectionInterface<MyType> c = new ImplementingClass<MyType>();

}
```

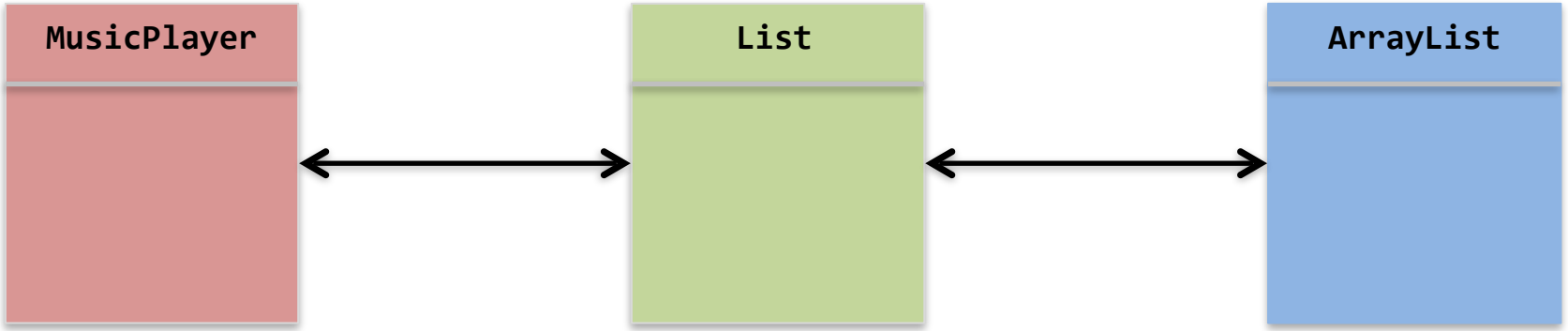
There are, of course, exceptions to this rule.

Why interfaces?

An interface defines a contract between a client and a provider.

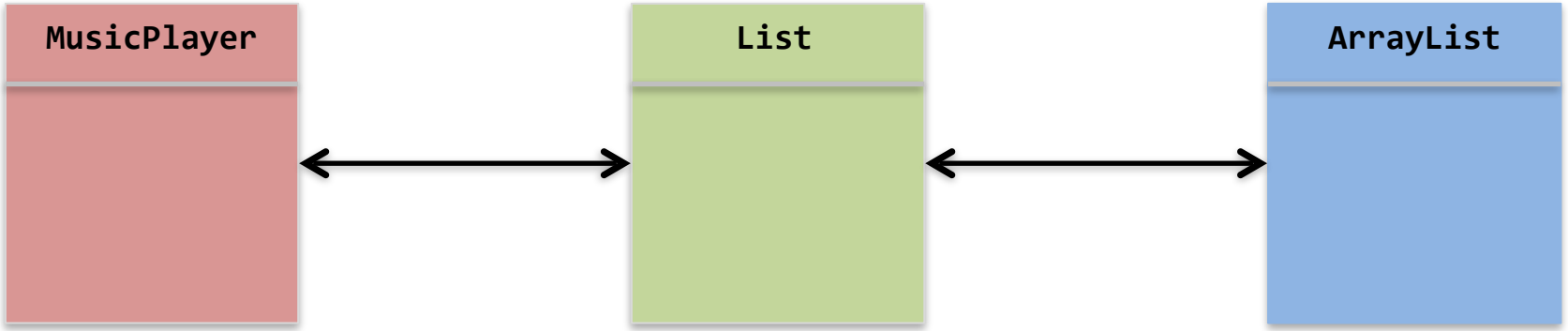
Why interfaces?

An interface defines a contract between a client and a provider.



Why interfaces?

An interface defines a contract between a client and a provider.



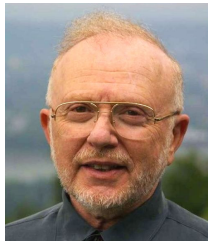

```
public class MusicPlayer {

    List<Song> playlist = new ArrayList<Song>();

}
```

Why interfaces?

Big Ideas *Information hiding, encapsulation, abstract data types*

Why interfaces?

Big Ideas *Information hiding, encapsulation, abstract data types*



David Parnas

"… one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others."

On the Criteria To Be Used in Decomposing Systems into Modules. *CACM* 15(12), 1972.

"The connections between modules are the assumptions which the modules make about each other."

Information Distribution Aspects of Design Methodology. *Information Processing* (71), 1972.

Barbara Liskov

"When a programmer makes use of an abstract data object, he is concerned only with the behavior which that object exhibits but not with any details of how that behavior is achieved by means of an implementation. The behavior of an object is captured by the set of characterizing operations. Implementation information, such as how the object is represented in storage, is only needed when defining how the characterizing operations are to be implemented. The user of the object is not required to know or supply this information."

Programming with abstract data types. SIGPLAN Not. 9, 4 (March 1974)

Sample JCF Classes

Each JCF interface has one or more implementing classes.

Set	TreeSet	HashSet	LinkedHashSet
List	ArrayList	LinkedList	Vector
SortedSet	TreeSet	ConcurrentSkipListSet	
Map	TreeMap	HashMap	LinkedHashMap

Sample JCF Classes

Each JCF interface has one or more implementing classes.

Set	TreeSet	HashSet	LinkedHashSet
List	ArrayList	LinkedList	Vector
SortedSet	TreeSet	ConcurrentSkipListSet	
Map	TreeMap	HashMap	LinkedHashMap

Note the naming convention for the implementing classes:

- The first part of the name gives a clue to the data structure being used.
- The second part of the name lists the interface being implemented.

There are exceptions to this naming convention (e.g., Vector).

Building our own collections

It’s possible that you might need to build a customized collection one day.

It’s guaranteed that you will need to build and manage your own data structures.

A solid understanding of how data structures work is required, fundamental knowledge.

Building our own collections and data structures is an excellent way to understand:

- Algorithm efficiency
- Software design and implementation issues
- Engineering tradeoffs