Chapter 1_1

Abstract data types (ADT) and their specification

Syllabus

- **Abstract data types (ADT) and their specification**: How to implement an ADT. Concrete state space, concrete invariant, abstraction function. Implementing operations, illustrated by the Text example.

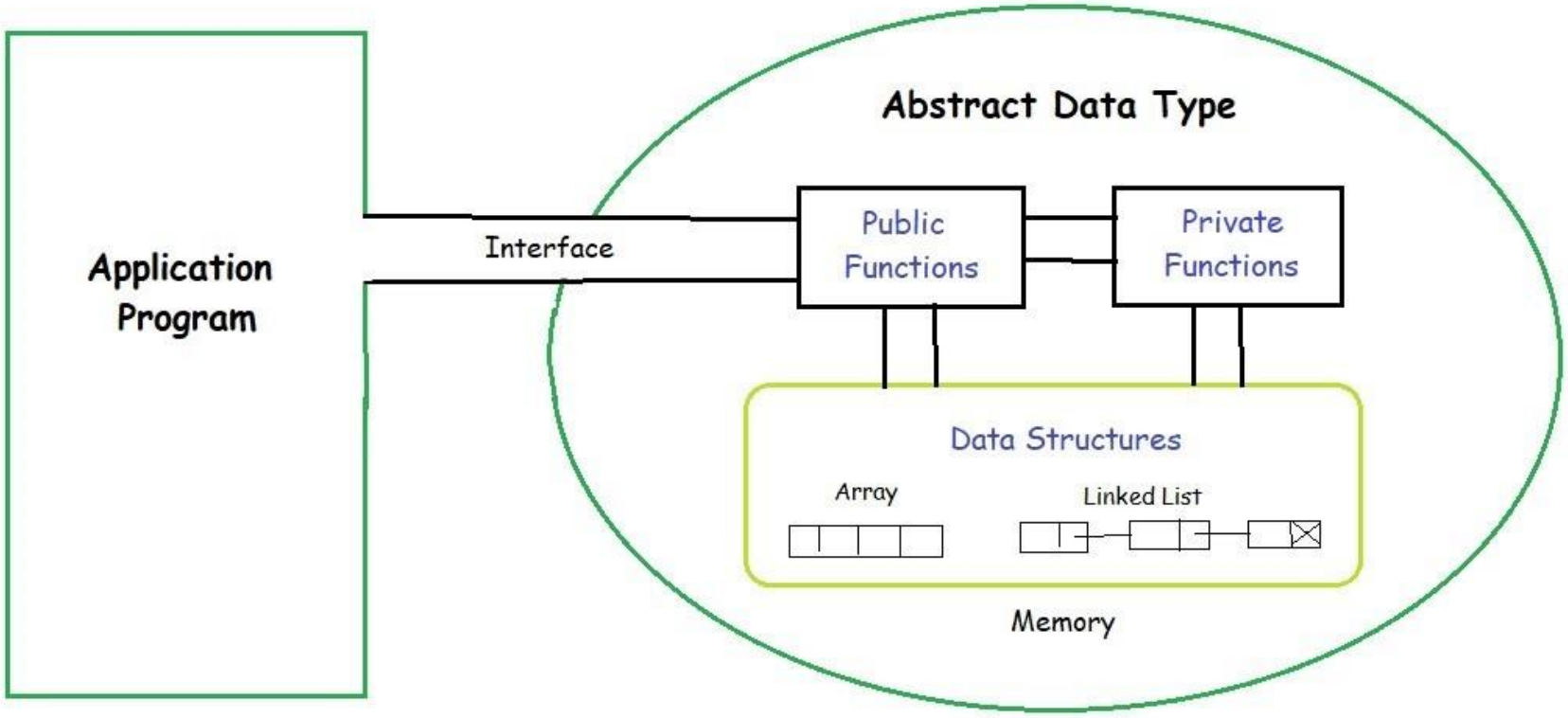
Abstract data types (ADT)

- Abstract Data types in Java are the most conceptual thing to learn in Java. It brings out the beauty of Java and its abstract implementation. Before we start to read about the topic in detail, let us take a real-life example of Abstract type.
- Put yourself inside the cockpit of a plane. You are Captain John Wick and you are in charge of flying the Boeing 777 from Delhi to London. Years of flight school and experience have sharpened your skills as a pilot and you are ready to take off. However, to fly a plane, do you need to know how the engines work? Do you know how the fuel discharge system works in an aircraft?
- No.
- You just need to know how to fly it.
- This is an abstraction. In Java, you can implement abstraction by using classes and interfaces.

Abstract data types (ADT)

- Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of value and a set of operations.
- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.

Abstract data types (ADT)



Abstract data types (ADT)

- The user of data type does not need to know how that data type is implemented.
- We have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented. So a user only needs to know what a data type can do, but not how it will be implemented.

Abstract Data Types in Java

List ADT

Stack ADT

Queue ADT

Abstract Datatypes in Java

- There are many different types of abstract data types in Java.
- Simply speaking, if you define a datatype with the help of a programming language and then hide its implementation, then it is an abstract datatype.
- You just present the user with the datatype and the operations for that datatype. This enables the programmer to easily use complex data structures without thinking about its implementation.

Operations on Abstract Data Types in Java

Operations on Abstract Datatypes in Java

1. Creators – These operations primarily create objects of a certain type. However, note that some creators can take objects as parameters too. But a creator will never take an object of the same type as itself.

2. Producers – This may sound similar to the creator operations but producers produce a different object of the same type. It does this by performing some kind of operation on the objects and returning a new object. For example, when you merge two strings together, the producer method combines the two string objects and returns a new object.

3. Observers – These are essentially the operations that take an object of a single type and return an object of a different type. For example, if you want to check whether a stack is empty, the function takes a stack object and returns a boolean value.

4. Mutators – These operations change the object itself. This is evident when you add a new item to a list that changes the object itself.

List: Abstract Datatype in Java

- A list is a linear collection, like a stack and queue, but more flexible: adding and removing elements from a list does not have to happen at one end or the other.

Operation	Description
removeFirst	Removes the first element from the list
removeLast	Removes the last element from the list
remove	Removes a particular element from the list
first	Examines the element at the front of the list
last	Examines the element at the rear of the list
contains	Determines if a particular element is in the list
isEmpty	Determines whether the list is empty
size	Determines the number of elements in the list
iterator	Returns an iterator for the list's elements
toString	Returns a string representation of the list

List: Abstract Datatype in Java

- We use Java interfaces to formally define the operations on the lists

ListADT Interface

```
import java.util.Iterator;
public interface ListADT<T> {

  //  Removes and returns the first element from this list
  public T removeFirst ( );
  //  Removes and returns the last element from this list
  public T removeLast ( );
  //  Removes and returns the specified element from this list
  public T remove (T element);
  //  Returns a reference to the first element on this list
  public T first ( );
  //  Returns a reference to the last element on this list
  public T last ( );
  //  cont'd..
```

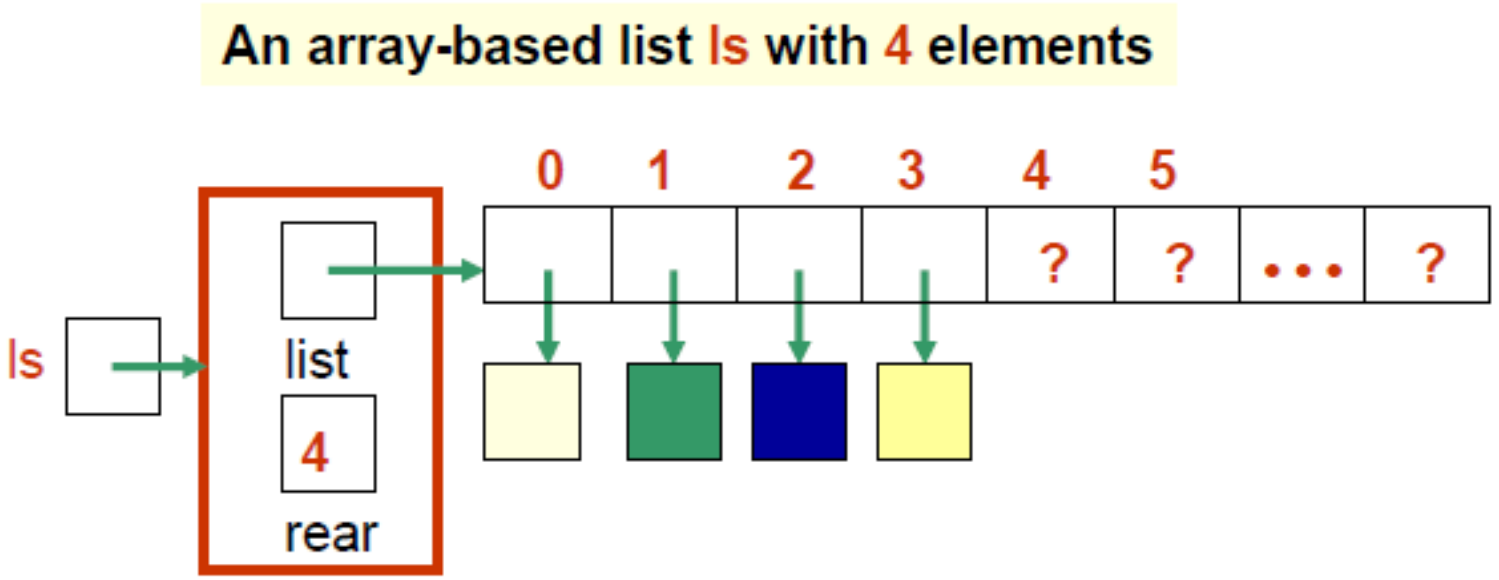
List: Abstract Datatype in Java

```
  // ..cont'd
  // Returns true if this list contains the specified target element
  public boolean contains (T target);
  // Returns true if this list contains no elements
  public boolean isEmpty( );
  // Returns the number of elements in this list
  public int size( );
  // Returns an iterator for the elements in this list
  public Iterator<T> iterator( );
  // Returns a string representation of this list
  public String toString( );
}
```

List Implementation using Arrays

- Container is an array
- Fix one end of the list at index 0 and shift ***as needed*** when an element is added or removed
- Is a shift needed when an element is added
  - at the front?
  - somewhere in the middle?
  - at the end?
- Is a shift needed when an element is removed
  - from the front?
  - from somewhere in the middle?
  - from the end?

List Implementation using Arrays



List Implementation using Arrays

```
//-----------------------------------------------------------------
//  Removes and returns the specified element.
//-----------------------------------------------------------------
public T remove (T element) throws ElementNotFoundException
{
   T result;
   int index = find (element);    // uses helper method find
   if (index == NOT_FOUND)
      throw new ElementNotFoundException("list");
   result = list[index];
   rear--;
   // shift the appropriate elements
   for (int scan=index; scan < rear; scan++)
      list[scan] = list[scan+1];
   list[rear] = null;
   return result;
}
```

The remove( ) operation

9-26

List Implementation using Arrays

```
//-----------------------------------------------------------------
//  Returns the array index of the specified element,
//  or the constant NOT_FOUND if it is not found.
//-----------------------------------------------------------------
private int find (T target)
{
   int scan = 0, result = NOT_FOUND;
   boolean found = false;
   if (! isEmpty( ))
      while (! found && scan < rear)
         if (target.equals(list[scan])
            found = true;
         else
            scan++;
   if (found)
      result = scan;
   return result;
}
```

The find( ) helper method

9-27

List Implementation using Arrays

```
//-----------------------------------------------------------
//  Returns true if this list contains the specified element.
//-----------------------------------------------------------
public boolean contains (T target)
{
   return (find(target) != NOT_FOUND);
                              //uses helper method find
}
```

The contains( ) operation

Poly, an immutable datatype: overview

```
/**
 * A Poly is an immutable polynomial with
 * integer coefficients.  A typical Poly is
 *          c0 + c1x + c2x^2 + ...
 **/
class Poly {
```

Abstract state (specification fields)

Poly: creators

```
// effects: makes a new Poly = 0
public Poly()

// effects: makes a new Poly = cx^n
// throws: NegExponent if n < 0
public Poly(int c, int n)
```

Poly: observers

```
// returns: the degree of this,
//   i.e., the largest exponent with a
//   non-zero coefficient.
//   Returns 0 if this = 0.
public int degree()

// returns: the coefficient of the term
//   of this whose exponent is d
// throws: NegExponent if d < 0
public int coeff(int d)
```

Poly: observers

Observers

- Used to obtain information about objects of the type
- Return values of other types
- Never modify the abstract value
- Specification uses the abstraction from the overview

`this`

- The particular `Poly` object being accessed
- *Target* of the invocation
- Also known as the *receiver*

```
Poly x = new Poly(4, 3);
int c = x.coeff(3);
System.out.println(c);    // prints 4
```

Poly: producers

```
// returns: this + q (as a Poly)
public Poly add(Poly q)

// returns: the Poly equal to this * q
public Poly mul(Poly q)

// returns: -this
public Poly negate()
```

Poly: producers

Operations on a type that create other objects of the type

Common in immutable types like `java.lang.String`

- `String substring(int offset, int len)`

No side effects

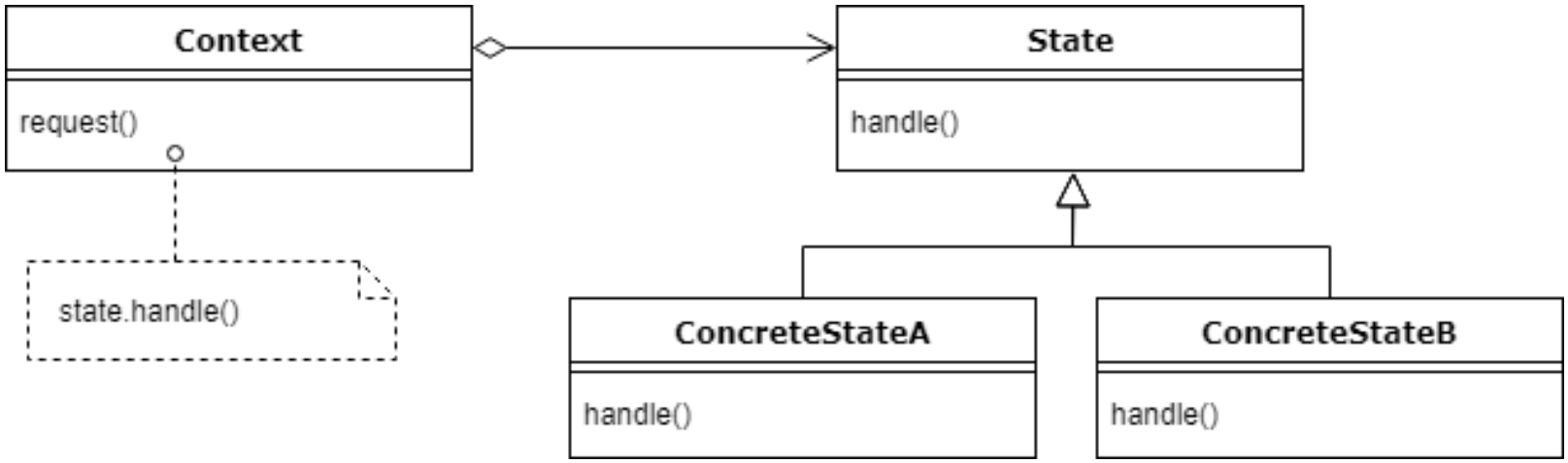
- Cannot change the abstract value of existing objects

State space

- A state space is the set of possible states, in this case of an object.
- To make things simple, let's visualize a **TV box** operated with remote controller. We can change the state of TV by pressing buttons on remote. But the state of TV will change or not, it depends on the current state of the TV. If TV is ON, we can switch it OFF, mute or change aspects and source. But if TV is OFF, nothing will happen when we press remote buttons. For a switched OFF TV. only possible next state can be switch ON.
- Another example can be of **Java thread** states. A thread can be one of its five states during it's life cycle. It's next state can be determined only after getting it's current state. e.g. we can not start a stopped thread or we cannot a make a thread wait, until it has started running.

State Design Pattern Implementation

- Define separate (state) objects that encapsulate state-specific behavior for each state. That is, define an interface (state) for performing state-specific behavior, and define classes that implement the interface for each state.



Design participants

- **State** – The interface define operations which each state must handle.
- **Concrete States** – The classes which contain the state specific behavior.
- **Context** – Defines an interface to client to interact. It maintains references to concrete state object which may be used to define current state of object. It delegates state-specific behavior to different State objects.

ADT

- We study a more formal mathematical idea of what it means for a class to implement an ADT, via the notions of *abstraction functions* and *rep invariants*.
- These mathematical notions are eminently practical in software design.
- The abstraction function will give us a way to cleanly define the equality operation on an abstract data type (which we'll discuss in more depth in a future).
- The rep invariant will make it easier to catch bugs caused by a corrupted data structure.

Invariants

- Resuming our discussion of what makes a good abstract data type, the final, and perhaps most important, property of a good abstract data type is that it **preserves its own invariants**.
- An *invariant* is a property of a program that is always true, for every possible runtime state of the program.
- Saying that the ADT *preserves its own invariants* means that the ADT is responsible for ensuring that its own invariants hold. It doesn't depend on good behavior from its clients.
- Immutability is one crucial invariant that we've already encountered: once created, an immutable object should always represent the same value, for its entire lifetime.

Immutability

```
public class Tweet
{
        public String author;
        public String text;
        public Date timestamp;
        public Tweet(String author, String text, Date timestamp)
        {
                this.author = author;
                this.text = text;
                this.timestamp = timestamp;
        }
}
```

Immutability

- How do we guarantee that these Tweet objects are immutable – that, once a tweet is created, its author, message, and date can never be changed?
- The first threat to immutability comes from the fact that clients can — in fact must — directly access its fields. So nothing's stopping us from writing code like this:

```
Tweet t = new Tweet("RAM", "Thanks to all those men out there inspiring me every day", new Date());
t.author = "SAM";
```

Immutability

- This is a trivial example of **representation (Rep) exposure**, meaning that code outside the class can modify the representation directly.
- Rep exposure like this threatens not only invariants, but also representation independence.
- Fortunately, Java gives us language mechanisms to deal with this kind of rep exposure.

Immutability

```
public class Tweet
{
        private final String author;
        private final String text;
        private final Date timestamp;
        public Tweet(String author, String text, Date timestamp)
        {
                this.author = author;
                this.text = text;
                this.timestamp = timestamp;
         }
```

Immutability

```
    public String getAuthor()
     {
            return author;
    }
    public String getText()
    {
            return text;
    }
    public Date getTimestamp()
     {
            return timestamp;
     }
}
```

Immutability

- The private and public keywords indicate which fields and methods are accessible only within the class and which can be accessed from outside the class.
- The final keyword also helps by guaranteeing that the fields of this immutable type won't be reassigned after the object is constructed.
- But that's not the end of the story: the rep is still exposed! Consider this perfectly reasonable client code that uses Tweet:

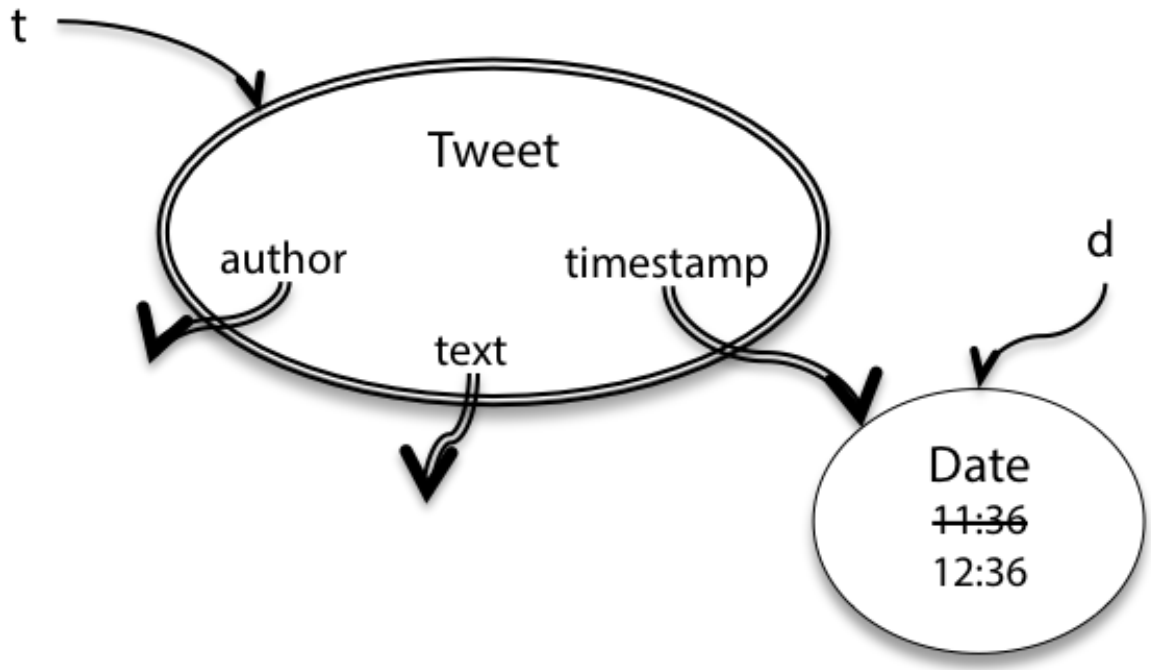
Immutability

```
public static Tweet retweetLater(Tweet t)
{
        Date d = t.getTimestamp();
        d.setHours(d.getHours()+1);
        return new Tweet("SAM", t.getText(), d);
}
```

- retweetLater takes a tweet and should return another tweet with the same message (called a *retweet*) but sent an hour later.
- The retweetLater method might be part of a system that automatically echoes funny things that Twitter celebrities say.

Immutability

- What's the problem here? The getTimestamp call returns a reference to the same date object referenced by tweet t. t.timestamp and d are aliases to the same mutable object. So when that date object is mutated by d.setHours(), this affects the date in t as well, as shown in the diagram below.



Immutability

- Tweet's immutability invariant has been broken. The problem is that Tweet leaked out a reference to a mutable object that its immutability depended on. We exposed the rep, in such a way that Tweet can no longer guarantee that its objects are immutable. Perfectly reasonable client code created a subtle bug.
- We can patch this kind of rep exposure by using defensive copying: making a copy of a mutable object to avoid leaking out references to the rep. Here's the code:

```
public Date getTimestamp()
{
        return new Date(Date.getTime());
}
```

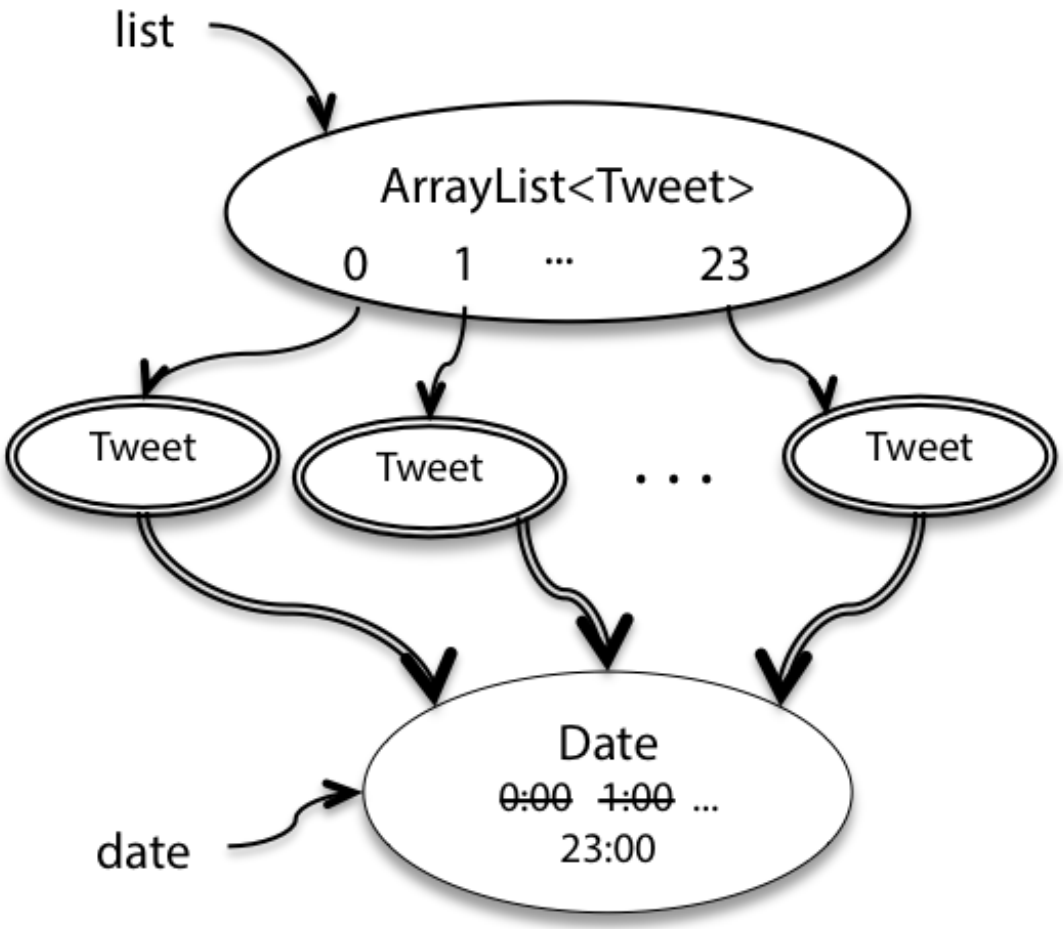
Immutability

- Mutable types often have a copy constructor that allows you to make a new instance that duplicates the value of an existing instance. In this case, Date's copy constructor uses the timestamp value, measured in milliseconds since January 1, 1970. As another example, StringBuilder's copy constructor takes a String. Another way to copy a mutable object is clone(), which is supported by some types but not all. There are unfortunate problems with the way clone() works in Java.
- So we've done some defensive copying in the return value of getTimestamp. But we're not done yet! There's still rep exposure. Consider this (again perfectly reasonable) client code:

Immutability

```
public static List<Tweet> tweetEveryHourToday ()
{
        List<Tweet> list = new ArrayList<Tweet>();
        Date date = new Date();
        for (int i=0; i < 24; i++)
        {
                date.setHours(i);
                list.add(new Tweet(“SAM", "keep it up! you can do
                        it", date));
        }
        return list;
}
```

Immutability



Immutability

- This code intends to advance a single Date object through the 24 hours of a day, creating a tweet for every hour. But notice that the constructor of Tweet saves the reference that was passed in, so all 24 Tweet objects end up with the same time, as shown in this previous diagram.
- Again, the immutability of Tweet has been violated. We can fix this problem too by using judicious defensive copying, this time in the constructor:

Immutability

```
public Tweet(String author, String text, Date timestamp)
{
        this.author = author;
        this.text = text;
        this.timestamp = new Date(timestamp.getTime());
}
```

- In general, you should carefully inspect the argument types and return types of all your ADT operations. If any of the types are mutable, make sure your implementation doesn't return direct references to its representation. Doing that creates rep exposure.

Immutability

- An even better solution is to prefer immutable types. If we had used an immutable date object, like java.time.ZonedDateTime, instead of the mutable java.util.Date, then we would have ended this section after talking about public and private. No further rep exposure would have been possible.
- **Immutable Wrappers Around Mutable Data Types**
  - The Java collections classes offer an interesting compromise: immutable wrappers.
  - Collections.unmodifiableList() takes a (mutable) List and wraps it with an object that looks like a List, but whose mutators are disabled – set(), add(), remove() throw exceptions. So you can construct a list using mutators, then seal it up in an unmodifiable wrapper (and throw away your reference to the original mutable list), and get an immutable list.

Rep Invariant and Abstraction Function

- We now take a deeper look at the theory underlying abstract data types. If you understand the theory deeply, you'll be able to build better abstract types, and will be less likely to fall into subtle traps.
- In thinking about an abstract type, it helps to consider the relationship between two spaces of values.
- The space of representation values (or rep values for short) consists of the values of the actual implementation entities. In simple cases, an abstract type will be implemented as a single object, but more commonly a small network of objects is needed, so this value is actually often something rather complicated. For now, though, it will suffice to view it simply as a mathematical value.

Rep Invariant and Abstraction Function

- The space of abstract values consists of the values that the type is designed to support. These are a figment of our imaginations. They're platonic entities that don't exist as described, but they are the way we want to view the elements of the abstract type, as clients of the type. For example, an abstract type for unbounded integers might have the mathematical integers as its abstract value space; the fact that it might be implemented as an array of primitive (bounded) integers, say, is not relevant to the user of the type.
- Now of course the implementor of the abstract type must be interested in the representation values, since it is the implementor's job to achieve the illusion of the abstract value space using the rep value space.

Rep Invariant and Abstraction Function

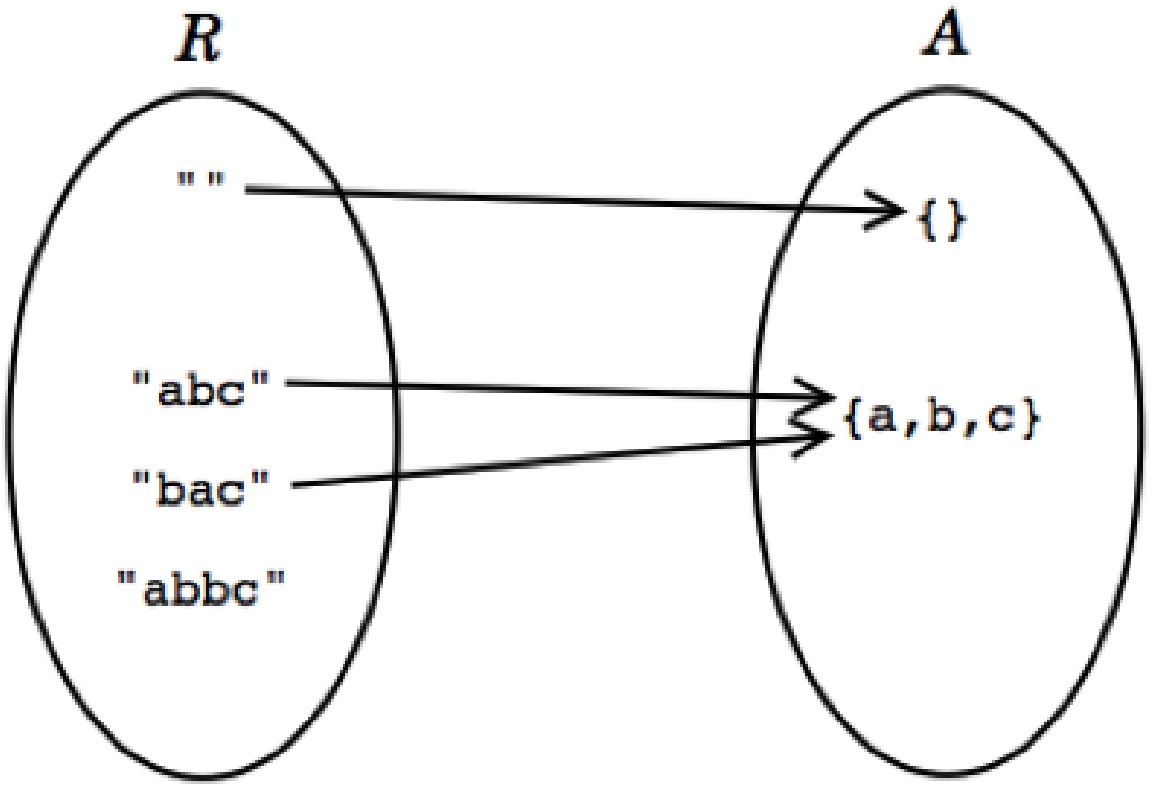
- Suppose, for example, that we choose to use a string to represent a set of characters:

```
public class CharSet
{
        private String s;
         ...
}
```

- Then the rep space R contains Strings, and the abstract space A is mathematical sets of characters.

Rep Invariant and Abstraction Function

- We can show the two value spaces graphically, with an arc from a rep value to the abstract value it represents.



Rep Invariant and Abstraction Function

- **Every abstract value is mapped to by some rep value**. The purpose of implementing the abstract type is to support operations on abstract values. Presumably, then, we will need to be able to create and manipulate all possible abstract values, and they must therefore be representable.
- **Some abstract values are mapped to by more than one rep value**. This happens because the representation isn't a tight encoding. There's more than one way to represent an unordered set of characters as a string.
- **Not all rep values are mapped**. In this case, the string "abbc" is not mapped. In this case, we have decided that the string should not contain duplicates.

Rep Invariant and Abstraction Function

- An *abstraction function* that maps rep values to the abstract values they represent:

  AF : R → A

  The arcs in the diagram show the abstraction function.

- A *rep invariant* that maps rep values to booleans:

  RI : R → boolean

  For a rep value r, RI(r) is true if and only if r is mapped by AF.

Rep Invariant and Abstraction Function

- Both the rep invariant and the abstraction function should be documented in the code.

```
public class CharSet
{
        private String s;
        // Rep invariant:
        // s contains no repeated characters
        // Abstraction Function:
        // represents the set of characters found in s ...
}
```

End of Chapter 1_1

Questions?