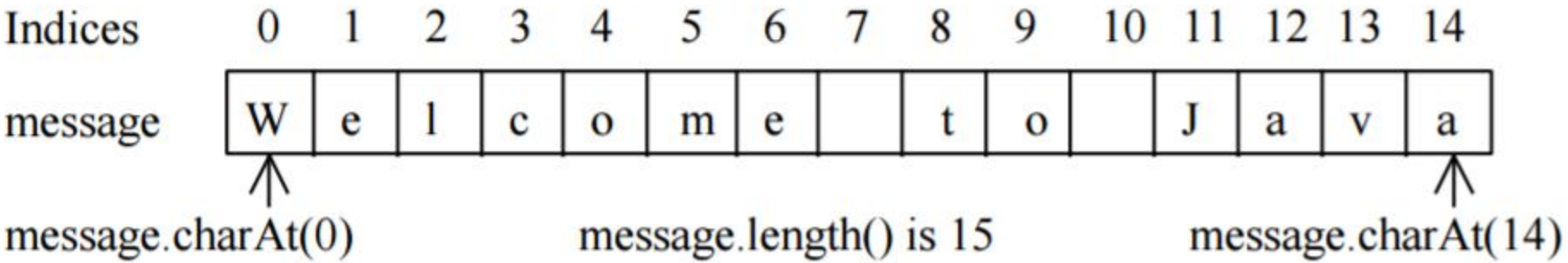
Finding String Length

- Finding string length using the length() method:

```
message = "Welcome";
message.length() // returns 7
```

Retrieving Individual Characters in a String

- Use message.charAt(index) char
- Index starts from 0



Extracting Substrings

java.lang.String
+subString(beginIndex: int): String +subString(beginIndex: int, endIndex: int): String

Returns this string's substring that begins with the character at the specified beginIndex and extends to the end of the string.

Returns this string's substring that begins at the specified beginIndex and extends to the character at index endIndex – 1. Note that the character at endIndex is not part of the substring.

StringBuilder and StringBuffer

- The StringBuilder/StringBuffer class is an alternative to the String class:
  - StringBuilder/StringBuffer can be used wherever a string is used
  - StringBuilder/StringBuffer is more flexible than String
  - You can add, insert, or append new contents into a string buffer, whereas the value of a String object is fixed once the string is created

StringBuilder Constructors

java.lang.StringBuilder
+StringBuilder() +StringBuilder(capacity: int) +StringBuilder(s: String)

Constructs an empty string builder with capacity 16.
Constructs a string builder with the specified capacity.
Constructs a string builder with the specified string.

Designing a Class

- Coherence: A class should describe a single entity
- Separating responsibilities: A single entity with too many responsibilities can be broken into several classes to separate responsibilities
- Classes are designed for reuse!
- Provide a public no-arg constructor and override the equals method and the toString method defined in the Object class whenever possible

Designing a Class

- Follow standard Java programming style and naming conventions:
  - Choose informative names for classes, data fields, and methods,
  - Place the data declaration before the constructor, and place constructors before methods,
  - Always provide a constructor and initialize variables to avoid programming errors.

Abstract method in abstract class

- An abstract method cannot be contained in a nonabstract class.
- In a nonabstract subclass extended from an abstract class, all the abstract methods must be implemented, even if they are not used in the subclass.
- If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined abstract.

Abstract classes

- An object cannot be created from abstract class
- An abstract class cannot be instantiated using the new operator.
- We can still define its constructors, which are invoked in the constructors of its subclasses.
  - For instance, the constructors of GeometricObject are invoked in the Circle class and the Rectangle class.
- A class that contains abstract methods must be abstract

Abstract classes

- An abstract class without abstract method:
  - It is possible to define an abstract class that contains no abstract methods.
  - We cannot create instances of the class using the new operator.
  - This class is used as a base class for defining new subclasses.

- A subclass can be abstract even if its superclass is concrete.
  - For example, the Object class is concrete, but a subclass, GeometricObject, is abstract

- A subclass can override a method from its superclass to define it abstract
  - rare, but useful when the implementation of the method in the superclass becomes invalid in the subclass.
  - the subclass must be defined abstract

Abstract classes as types

- You cannot create an instance from an abstract class using the new operator, but an abstract class can be used as a data type:

  GeometricObject c = new Circle(2);

  - The following statement, which creates an array whose elements are of GeometricObject type, is correct

    GeometricObject[] geo=new GeometricObject[10];

    - There are only null elements in the array!!!

The abstract Calendar class and its GregorianCalendar subclass

- An instance of java.util.Date represents a specific instant in time with millisecond precision.

- java.util.Calendar is an abstract base class for extracting detailed information such as year, month, date, hour, minute and second from a Date object for a specific calendar.
  - Subclasses of Calendar can implement specific calendar systems such as Gregorian calendar, Lunar Calendar and Jewish calendar.
    - java.util.GregorianCalendar is for the Gregorian calendar

The GregorianCalendar Class

- Java API for the GregorianCalendar class:

  http://docs.oracle.com/javase/8/docs/api/java/ util/GregorianCalendar.html

- New GregorianCalendar() constructs a default GregorianCalendar with the current time.

- New GregorianCalendar(year, month, date) constructs a GregorianCalendar with the specified year, month, and date.
  - The month parameter is 0-based, i.e., 0 is for January.

The abstract Calendar class and its GregorianCalendar subclass

java.util.Calendar
#Calendar() +get(field: int): int +set(field: int, value: int): void +set(year: int, month: int, dayOfMonth: int): void +getActualMaximum(field: int): int +add(field: int, amount: int): void +getTime(): java.util.Date +setTime(date: java.util.Date): void

Constructs a default calendar.
Returns the value of the given calendar field.
Sets the given calendar to the specified value.
Sets the calendar with the specified year, month, and date. The month parameter is 0-based, that is, 0 is for January.
Returns the maximum value that the specified calendar field could have.
Adds or subtracts the specified amount of time to the given calendar field.
Returns a Date object representing this calendar's time value (million second offset from the Unix epoch).
Sets this calendar's time with the given Date object.

The abstract Calendar class and its GregorianCalendar subclass

java.util.GregorianCalendar
+GregorianCalendar() +GregorianCalendar(year: int, month: int, dayOfMonth: int) +GregorianCalendar(year: int, month: int, dayOfMonth: int, hour:int, minute: int, second: int)

Constructs a GregorianCalendar for the current time.
Constructs a GregorianCalendar for the specified year, month, and day Of month.

Constructs a GregorianCalendar for the specified year, month, day of month, hour, minute, and second. The month parameter is 0-based, That is, 0 is for January.

The get Method in Calendar Class

java.util.GregorianCalendar
+GregorianCalendar() +GregorianCalendar(year: int, month: int, dayOfMonth: int) +GregorianCalendar(year: int, month: int, dayOfMonth: int, hour:int, minute: int, second: int)

Constructs a GregorianCalendar for the current time.
Constructs a GregorianCalendar for the specified year, month, and day Of month.

Constructs a GregorianCalendar for the specified year, month, day of month, hour, minute, and second. The month parameter is 0-based, That is, 0 is for January.

Collections Framework

The collections framework was designed to meet several goals, such as –

- The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hashtables) were to be highly efficient.
- The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
- The framework had to extend and/or adapt a collection easily.

Collections Framework 'Cont'

The entire collections framework is designed around a set of standard interfaces.

Several standard implementations such as **LinkedList, HashSet,** and **TreeSet**, of these interfaces are provided that you may use as-is and you may also implement your own collection, if you choose.

Collections Framework

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following.

- **Interfaces**
- **Implementations**
- **Algorithms**

Interfaces

These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.

Implementation ‘Classes’

These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.

Algorithms

These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.

Types of Collection Framwork Interfaces

The Collection Interface- This enables you to work with groups of objects; it is at the top of the collections hierarchy.

The List Interface- This extends **Collection** and an instance of List stores an ordered collection of elements.

The Set- This extends Collection to handle sets, which must contain unique elements.

Types of Collection Framwork Interfaces

The SortedSet - This extends Set to handle sorted sets.

The Map - This maps unique keys to values.

he Map.Entry - This describes an element (a key/value pair) in a map. This is an inner class of Map.

The SortedMap - This extends Map so that the keys are maintained in an ascending order

The Enumeration- This is legacy interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. This legacy interface has been superceded by Iterator.

Collection Interface

It is the foundation upon which the collections framework is built. It declares the core methods that all collections will have. These methods are summarized in the following table.

Because all collections implement Collection, familiarity with its methods is necessary for a clear understanding of the framework. Several of these methods can throw an **UnsupportedOperationException**.

Methods of Collection Interface 'Cont'

boolean add(Object obj)- Adds obj to the invoking collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection, or if the collection does not allow duplicates.

boolean addAll(Collection c)- Adds all the elements of c to the invoking collection. Returns true if the operation succeeds (i.e., the elements were added). Otherwise, returns false

void clear( )- Removes all elements from the invoking collection.

Methods of Collection Interface

boolean contains(Object obj)- Returns true if obj is an element of the invoking collection. Otherwise, returns false.

oolean containsAll(Collection c)- Returns true if the invoking collection contains all elements of **c**. Otherwise, returns false.

boolean equals(Object obj)- Returns true if the invoking collection and obj are equal. Otherwise, returns false.

Collection Classes

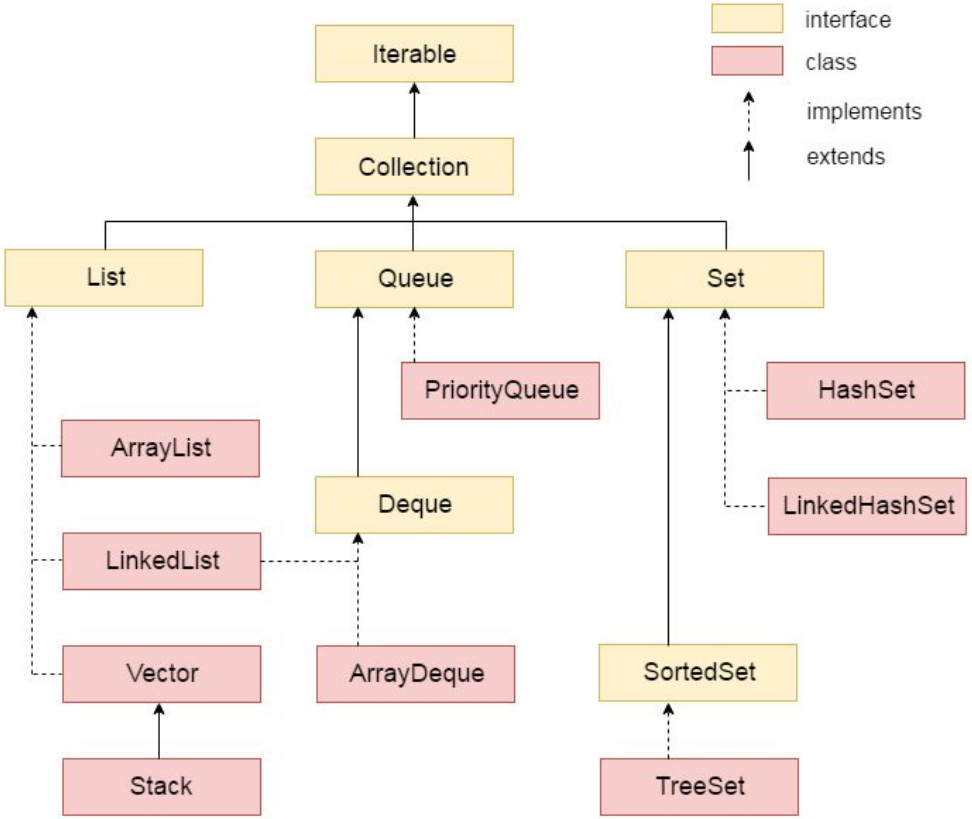
Java provides a set of standard collection classes that implement Collection interfaces.

Some of the classes provide full implementations that can be used as-is and others are abstract class, providing skeletal implementations that are used as starting points for creating concrete collections.

claim.Academy
(st. louis)

Collection Hierarchy

Let us see the hierarchy of collection framework.The **java.util** package contains all the classes and interfaces for Collection framework.



Types of Collection Classes

LinkedList - Implements a linked list by extending AbstractSequentialList.

ArrayList - Implements a dynamic array by extending AbstractList.

HashMap - Extends AbstractMap to use a hash table.

TreeMap - Extends AbstractMap to use a tree.

Types of Collection Classes

Stack - A subclass of Vector that implements a standard last-in, first-out stack.

Dictionary - An abstract class that represents a key/value storage repository and operates much like Map

Hashtable - Was part of the original java.util and is a concrete implementation of a Dictionary.

Inheritance

The process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order.

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).