Welcome to the course

COMP610
Data Structure and Algorithms

AUT UNIVERSITY
TE WĀNANGA ARONUI O TAMAKI MAKAU RAU

Course Information

Course Title:	**Data Structure and Algorithms**
Course Code:	**COMP510**
Prerequisites:	**405704 (or 735320) and 715189**
Co-requisites:	**None**
Level:	**7**

Teaching Team

- **Dr Akbar Ghobakhlou**

  Lecturer

  akbar@aut.ac.nz

  Office: WT603A

  Phone: 921 9562

- **Teaching Assistants**

  Koz Ross (City Campus)

  koz.ross@aut.ac.nz

  Shane Birdsall (South Campus)

  gbv1537@aut.ac.nz

PLEASE TRY NOT TO CONTACT THEM OUTSIDE OF LAB HOURS! They are only paid to do labs, assist with marking!

AUT UNIVERSITY
TE WĀNANGA ARONUI O TAMAKI MAKAU RAU

Course Attendance!

1 x 2 hour lecture, Please do not fall asleep!

Tuesday	16:00 – 18:00 MC214 (South Campus)
Wednesday	14:00 – 16:00 WS102 (City Campus)

1 x 2 hour lab

Friday	14:00 – 16:00 WT204 (City Campus)
Friday	16:00 – 18:00 WT204(City Campus)
Thursday	16:00 – 8:00 MC209 (South Campus)

Please stick to your stream times!

Course also expects "*self directed learning*"

REALLY IMPORTANT YOU HAVE EXCELLENT ATTENDANCE

AUT UNIVERSITY
TE WĀNANGA ARONUI O TAMAKI MAKAU RAU

Assessments

Assessment type		Date
Interim Assessment (50%)	Lab (10%)	Weekly
	Assignment 1 (10%)	Week 5
	Mid-Semester Test (10%)	Week 6
	Assignment 2 (10%)	Week 7
	Assignment 3 (10%)	Week 11
Final Exam (50%)		To be confirmed
Overall requirement/s to pass the paper: To pass the paper, the student needs to gain: **• A minimum mark of 35% in overall coursework, AND/OR** **• A minimum mark of 35% in examination, AND** **• A C- (50%)overall grade**		

Resources

- **Java Structures, the Book**

Data Structures in Java, for the Principled Programmer

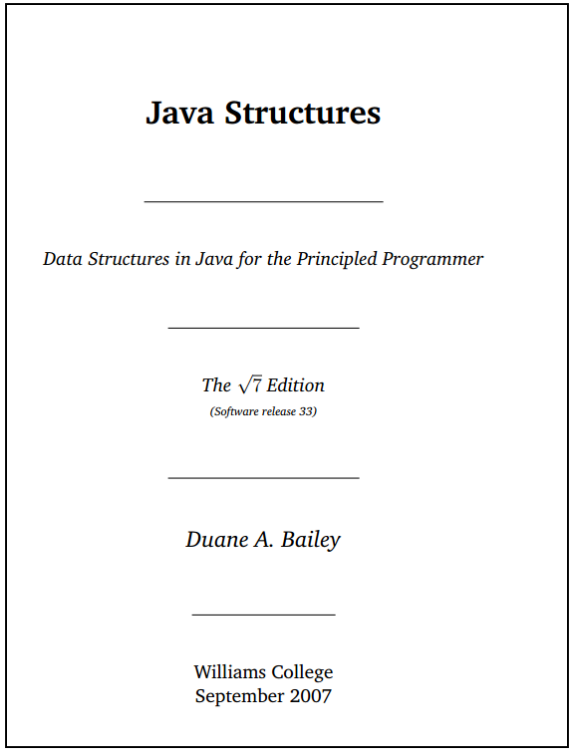
You are free to download *Java Structures*, the book, for educational use. You may read the book on-line,

http://dept.cs.williams.edu/~bailey/JavaStructures/Book.html

AUT Online

Check out AUT online for:

- Course materials
- Tutorial exercises
- Announcements


Java Structures

Data Structures in Java for the Principled Programmer

The $\sqrt{7}$ Edition

(Software release 33)

Duane A. Bailey

Williams College
September 2007

AUT UNIVERSITY
TE WĀNANGA ARONUI O TAMAKI MAKAU RAU

Course Policy

- ABSOLUTELY NO CHEATING!!!
- ASK FOR HELP!
  - Use *AUT Online* for questions on the assignment and to get sample code.
  - See me in person.
  - For personal matters email me.
- DO THE LABS!
  - You must do the labs during lab time. No Games, No Facebook, No cat videos, No Assignment work or other work.
- ATTEND LECTURES!
- No talking in class/lab when the TA's or I are talking.

AUT UNIVERSITY
TE WĀNANGA ARONUI O TAMAKI MAKAU RAU

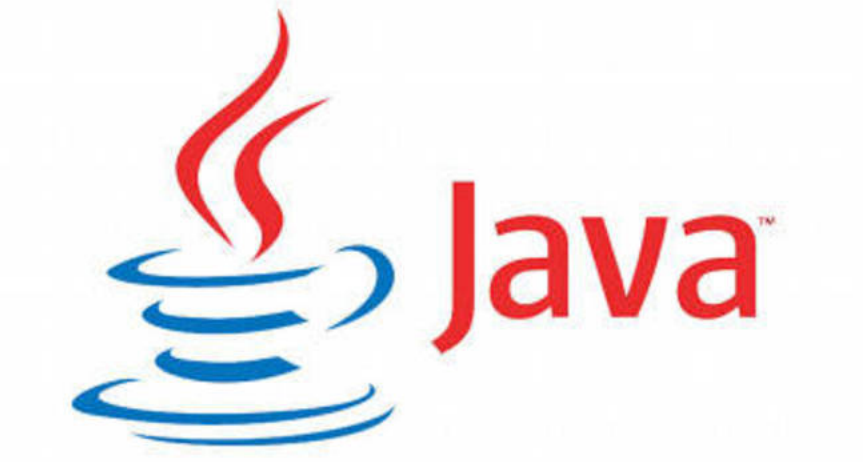
Course Content and Schedule

- Object-Orientation and Data Structures
- Recursion
- Sorting
- Control Structures
- Lists
- Stacks and Queues
- Ordered Structures
- Binary Trees Priority Queues
- Search Trees
- Maps Graphs

Java Skills

We assume:

- ❑Basic Types
- ❑Basic Control Flow
- ❑Basic GUI
- ❑Comments (JavaDoc)
- ❑Exception Handling
- ❑File I/O
- ❑Basic OO Concepts: polymorphism, inheritance, etc.
- ❑Reading Java API Specifications



What We Will Teach You!

- To teach you about ***Data Structures***!

  Data Structures are ways of storing and organizing data in a computer system so that it can be used efficiently.

  Goal of this Course: Identify and develop abstract principles for structuring data in ways that make programs efficient.

- To teach you about basic ***Algorithms***, mainly for use with data structures.

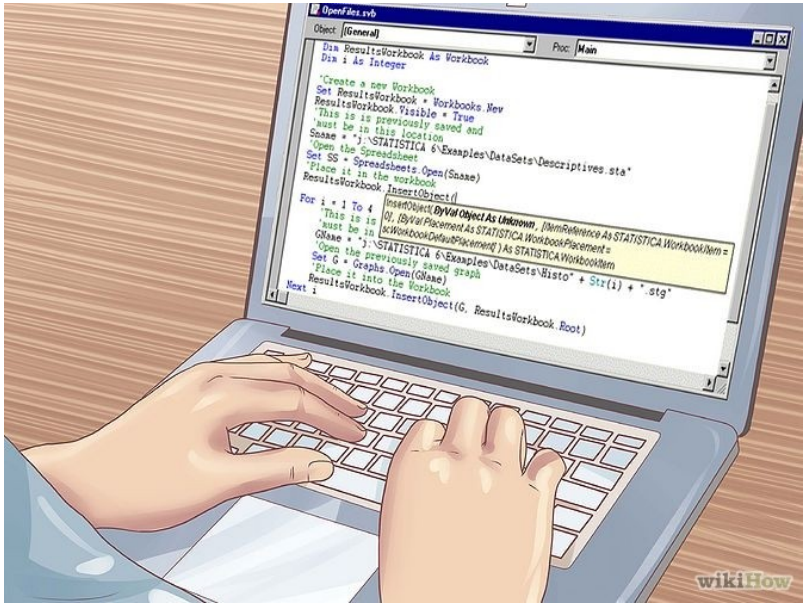
  Algorithm – a unambiguous, step-by-step procedure that terminates after a finite number of steps.

  A ***program*** is an implementation of an algorithm. An algorithm is the idea behind the program

- To improve your ***coding*** skills

AUT UNIVERSITY
TE WĀNANGA ARONUI O TAMAKI MAKAU RAU

The Coding Journey



How to learn programming:

- ❑ **Code**
- ❑ **Practice**
- ❑ **Focus**
- ❑ **Patience**

Week 1: Object Orientation and Data Structures

Part I: Data Abstraction

Part II: Container Classes

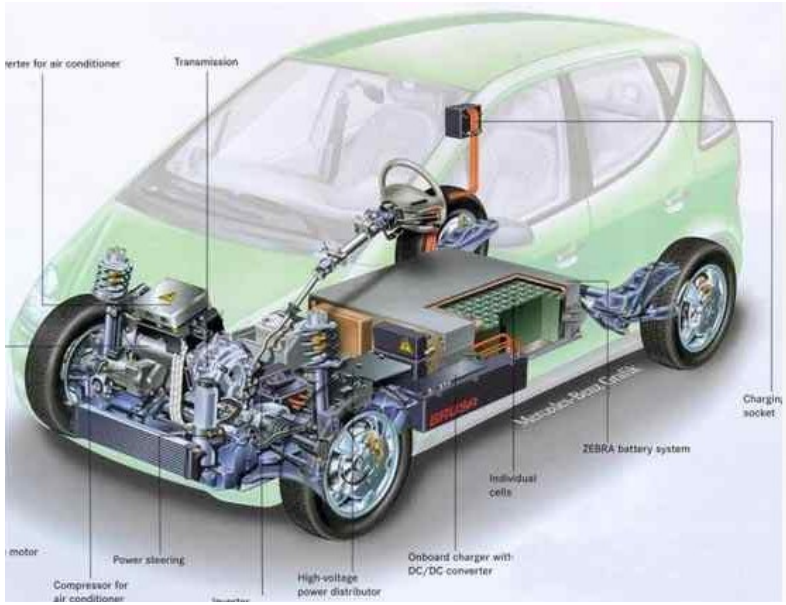
Part III: Java Collections

Part IV: Iterators

Data Abstraction

Data Abstraction

Data Abstraction



The notion of a car consists of a protocol or a contract, which is separate from the implementation of a car.

-Data Type: A protocol which specifies features and operations.
-Implementation: Data structures & algorithms that realizes the features and operations.

Encapsulation

An object-oriented language (Java)

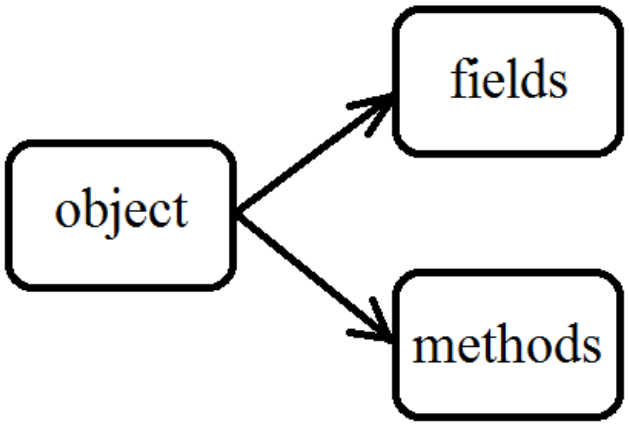
- abstracts data types into classes; and
- encapsulates data in objects – instances of classes.

Encapsulation

An object-oriented language (Java)

- abstracts data types into classes; and
- encapsulates data in objects – instances of classes.

An object contains fields and methods.



Encapsulation

Example 1: Rational Numbers

Definition. A rational number is a pair (a, b) of integers where $b \neq 0$, and $\gcd(a, b) = 1$.

e.g. $(1,2)$ represents $\frac{1}{2}$ $(12,7)$,represents $\frac{12}{7}$

Encapsulation

Example 1: Rational Numbers

Definition. A rational number is a pair (a, b) of integers where $b \neq 0$, and $\gcd(a, b) = 1$.

e.g. $(1,2)$ represents $\frac{1}{2}$ $(12,7)$,represents $\frac{12}{7}$

Fields:

- int numerator
- int denominator

Encapsulation

Example 1: Rational Numbers

Definition. A rational number is a pair (a, b) of integers where $b \neq 0$, and $\gcd(a, b) = 1$.

e.g. $(1,2)$ represents $\frac{1}{2}$ $(12,7)$,represents $\frac{12}{7}$

Fields:

- int numerator
- int denominator

Methods:

- int getNumerator()
- int getDenominator()
- double getValue()
- Rational add(Rational other)
- void reduce()
- String toString()
- boolean equals(Object o)
- int hashCode()

Encapsulation

Example 1: Rational Numbers

```
public class Ratio
{
    protected int numerator;   // numerator of ratio
    protected int denominator; // denominator of ratio

    public Ratio(int top, int bottom)
    // pre: bottom != 0
    // post: constructs a ratio equivalent to top::bottom
    {
        numerator = top;
        denominator = bottom;
        reduce();
    }
    public int getNumerator()
    // post: return the numerator of the fraction
    {
        return numerator;
    }
    public int getDenominator()
    // post: return the denominator of the fraction
    {
        return denominator;
    }
```

Encapsulation

Example 1: Rational Numbers

```
public class Ratio
{
    protected int numerator;    // numerator of ratio
    protected int denominator; // denominator of ratio
```

Fields Declarations

```
    public Ratio(int top, int bottom)
    // pre: bottom != 0
    // post: constructs a ratio equivalent to top::bottom
    {
        numerator = top;
        denominator = bottom;
        reduce();
    }
```

Preconditions: specify condition in which the method could be called.

Postcondition: specify the state of the program after the method completion, provided the precondition was met.

Constructor

```
    public int getNumerator()
    // post: return the numerator of the fraction
    {
        return numerator;
    }
    public int getDenominator()
    // post: return the denominator of the fraction
    {
        return denominator;
    }
```

Accessors

Encapsulates data using protected fields, and only allow access through the get methods.

Encapsulation

Example 1: Rational Numbers

```
    protected void reduce()
    // post: numerator and denominator are set so that
    // the greatest common divisor of the numerator and denominator is 1
    {
        int divisor = gcd(numerator,denominator);
        if (denominator < 0) divisor = -divisor;
        numerator /= divisor;
        denominator /= divisor;
    }
    protected static int gcd(int a, int b)
    // post: computes the greatest integer value that divides a and b
    {
        if (a < 0) return gcd(-a,b);
        if (a == 0) {
            if (b == 0) return 1;
            else return b;
        }
        if (b < a) return gcd(b,a);
        return gcd(b%a,a);
    }
}
```

Encapsulation

Example 1: Rational Numbers

```
protected void reduce()
// post: numerator and denominator are set so that
// the greatest common divisor of the numerator and denominator is 1
{
    int divisor = gcd(numerator,denominator);
    if (denominator < 0) divisor = -divisor;
    numerator /= divisor;
    denominator /= divisor;
}
protected static int gcd(int a, int b)
// post: computes the greatest integer value that divides a and b
{
    if (a < 0) return gcd(-a,b);
    if (a == 0) {
        if (b == 0) return 1;
        else return b;
    }
    if (b < a) return gcd(b,a);
    return gcd(b%a,a);
}
}
```

Reduce the numerator and denominator to lowest terms by removing any common factors.

static method: its utility is independent of the object

Euclid's algorithm for finding greatest common divisor

utility methods: part of the implementation; not a necessary part of the data type

Encapsulation

Example 1: Rational Numbers

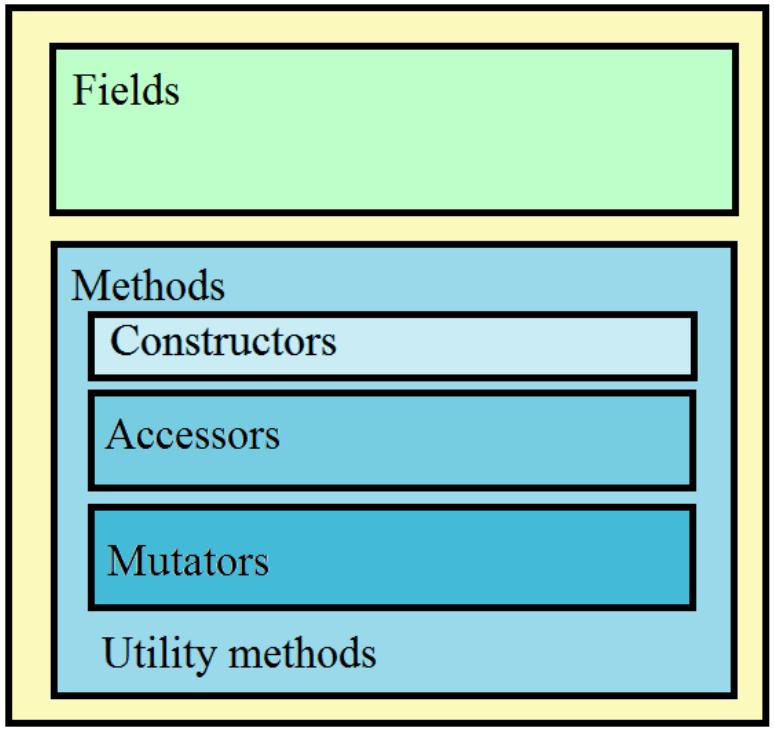
```
public static void main(String[] args)
{
    Ratio r = new Ratio(1,1);       // r == 1.0
    r = new Ratio(1,2);             // r == 0.5
    r.add(new Ratio(1,3));          // sum computed, but r still 0.5
    r = r.add(new Ratio(2,8));      // r == 0.75
    System.out.println(r.getValue()); // 0.75 printed

    System.out.println(r.toString());  // calls toString()
    System.out.println(r);  // calls toString()
}
```

Encapsulation

Java Classes

The general structure of a Java class



An accessor provides read-only access to an implementation's data, directly or indirectly. It does not modify the structure.

A mutator allows the user to modify the state of an object. It may also return a structure's data

A Special-Purpose Class

Example 2: A Bank Account

Goal: Create a BankAccount class with two fields:

- String acc // account id
- double bal // account balance

A Special-Purpose Class

Example 2: A Bank Account

```
public BankAccount(String acc, double bal)
// pre: account is a string identifying the bank account
// balance is the starting balance
// post: constructs a bank account with desired balance

public boolean equals(Object other)
// pre: other is a valid bank account
// post: returns true if this bank account is the same as other

public String getAccount()
// post: returns the bank account number of this account

public double getBalance()
// post: returns the balance of this bank account

public void deposit(double amount)
// post: deposit money in the bank account

public void withdraw(double amount)
// pre: there are sufficient funds in the account
// post: withdraw money from the bank account
```

A Special-Purpose Class

Example 2: A Bank Account

```
protected String account;  // the account number
protected double balance; // the balance associated with account

public BankAccount(String acc, double bal)
// pre: account is a string identifying the bank account
// balance is the starting balance
// post: constructs a bank account with desired balance
{
    account = acc;
    balance = bal;
}


public boolean equals(Object other)
// pre: other is a valid bank account
// post: returns true if this bank account is the same as other
{
    BankAccount that = (BankAccount)other;
    // two accounts are the same if account numbers are the same
    return this.account.equals(that.account);
}
```

A Special-Purpose Class

Example 2: A Bank Account

```
public String getAccount()
// post: returns the bank account number of this account
{
    return account;
}

public double getBalance()
// post: returns the balance of this bank account
{
    return balance;
}
```

Accessors

```
public void deposit(double amount)
// post: deposit money in the bank account
{
    balance = balance + amount;
}

public void withdraw(double amount)
// pre: there are sufficient funds in the account
// post: withdraw money from the bank account
{
    balance = balance - amount;
}
```

Mutators

A Special-Purpose Class

Example 2: A Bank Account

Question. Is it better to invest $100 over 10 years at 5% or to invest $100 over 20 years at 2.5% interest?

```
public static void main(String[] args)
{
    BankAccount jd = new BankAccount("Jain Dough",100.00);
    BankAccount js = new BankAccount("Jon Smythe",100.00);
    for (int years = 0; years < 10; years++)
    {
        jd.deposit(jd.getBalance() * 0.05);
    }
    for (int years = 0; years < 20; years++)
    {
        js.deposit(js.getBalance() * 0.025);
    }
    System.out.println("Jain invests $100 over 10 years at 5%.");
    System.out.println("After 10 years " + jd.getAccount() +
                       " has $" + jd.getBalance());
    System.out.println("Jon invests $100 over 20 years at 2.5%.");
    System.out.println("After 20 years " + js.getAccount() +
                       " has $" + js.getBalance());
}
```

A Special-Purpose Class

Example 2: A Bank Account

Take Home Question: Suppose we enrich the bank account by introducing a floating interest rate, a currency type, an account fee, and an overdraft cap. How should we modify the BankAccount class?

A General-Purpose Class

key	value
Bank Account	Balance (NZ$)
12-8313-9481921	281999.95
12-8471-1823913	1834.1
12-8131-1273895	183
12-4234-7452944	841183.38
12-8239-9487120	-5
12-8732-9327818	13841000

The BankAccount class can be viewed as a special case of a more general data structure: associations.

A General-Purpose Class

key	value
Bank Account	Balance (NZ$)
12-8313-9481921	281999.95
12-8471-1823913	1834.1
12-8131-1273895	183
12-4234-7452944	841183.38
12-8239-9487120	-5
12-8732-9327818	13841000

The BankAccount class can be viewed as a special case of a more general data structure: associations.

The Association Data Structure

Definition. An association is a key-value pair such that the key cannot be modified.

A General-Purpose Class

key	value
Bank Account	Balance (NZ$)
12-8313-9481921	281999.95
12-8471-1823913	1834.1
12-8131-1273895	183
12-4234-7452944	841183.38
12-8239-9487120	-5
12-8732-9327818	13841000

The BankAccount class can be viewed as a special case of a more general data structure: associations.

The Association Data Structure

Definition. An association is a key-value pair such that the key cannot be modified.

Associations appear everywhere: e.g.

- Bank accounts
- Student databases
- Dictionaries
- Emails
- OpenStreetMap

A General-Purpose Class

The Association Data Structure

```
public Association(Object key, Object value)
// pre: key is non-null
// post: constructs a key-value pair

public Association(Object key)
// pre: key is non-null
// post: constructs a key-value pair; value is null

public boolean equals(Object other)
// pre: other is non-null Association
// post: returns true iff the keys are equal

public Object getValue()
// post: returns value from association

public Object getKey()
// post: returns key from association

public Object setValue(Object value)
// post: sets association's value to value
```

A General-Purpose Class

The Association Data Structure

```
public Association(Object key, Object value)
// pre: key is non-null
// post: constructs a key-value pair

public Association(Object key)
// pre: key is non-null
// post: constructs a key-value pair; value is null

public boolean equals(Object other)
// pre: other is non-null Association
// post: returns true iff the keys are equal

public Object getValue()
// post: returns value from association

public Object getKey()
// post: returns key from association

public Object setValue(Object value)
// post: sets association's value to value
```

Object is the most general data type in Java

A General-Purpose Class

The Association Data Structure

```
public class Association
{
      protected Object theKey; // the key
      protected Object theValue; // the value

      public Association(Object key, Object value)
      // pre: key is non-null
      // post: constructs a key-value pair
      {
          if(key==null) {
              System.out.println("Key must not be null.");
              return;
          }
          theKey = key;
          theValue = value;
      }
      public Association(Object key)
      // pre: key is non-null
      // post: constructs a key-value pair; value is null
      {
          this(key,null);
      }
```

A General-Purpose Class

The Association Data Structure

```
public Object getValue()
// post: returns value from association
{
    return theValue;
}

public Object getKey()
// post: returns key from association
{
    return theKey;
}

public Object setValue(Object value)
// post: sets association's value to value
{
    Object oldValue = theValue;
    theValue = value;
    return oldValue;
}
}
```

Week 1: Object Orientation and Data Structures

Part II: Container Classes

Containers: Java Generics

Example 3: A Box class

We can use a box container to store a single data **item.**

Containers: Java Generics

Example 3: Box class: Usual Implementation

```
public class Box {
private Object object;
public void set(Object object) this.object=object; public Object get()
return object;
}
```

Containers: Java Generics

Example 3: Box class: Usual Implementation

```
public class Box {
private Object object;
public void set(Object object) this.object=object; public Object get()
return object;
}
```

To create a Box object with String Box stringBox = new Box();
stringBox.add(''YES'');
String s = (String)integerBox.get();

Containers: Java Generics

Example 3: Box class: Usual Implementation

```
public class Box {
private Object object;
public void set(Object object) this.object=object; public Object get()
return object;
}
```

To create a Box object with String Box stringBox = new Box();
stringBox.add(''YES'');
String s = (String)integerBox.get();

Disadvantages:
-Multiple type casts involved.
-A ClassCastException may be thrown at run-time if the wrong type of box is used.
-Need to implement a different box class for each type.

Containers: Java Generics

Generics Types

Generic types are parameters that can appear in the place of data types.

Example 3: Box class: Generic Implementation

```
public class Box<E> {
        private E eObj;   // E stands for "Element"
        public void set(E eObj) { this.eObj = eObj;}
        public E get() { return eObj; }
    }
```

Containers: Java Generics

Generics Types

Generic types are parameters that can appear in the place of data types.

Example 3: Box class: Generic Implementation

```
public class Box<E> {
        private E eObj;    // E stands for "Element" public
        void set(E eObj) { this.eObj = eObj;} public E
        get() { return eObj; }
}
```

To create a Box class with String:

```
Box<String> stringBox = new Box<String>();
stringBox.add("YES");
String s = stringBox.get();
```

Containers: Java Generics

In defining class Box<E>

1.Only local variables and non-static methods can use type E.

Containers: Java Generics

In defining class Box<E>

1.Only local variables and non-static methods can use type E.

```
static  E  obj;
```

Containers: Java Generics

In defining class Box<E>

1.Only local variables and non-static methods can use type E.
static E obj; Not allowed!

Containers: Java Generics

In defining class Box<E>

1.Only local variables and non-static methods can use type E.
static E obj; Not allowed!

2.DO NOT invoke E's constructor or methods.

Containers: Java Generics

In defining class Box<E>

1.Only local variables and non-static methods can use type E.
static E obj; Not allowed!

Containers: Java Generics

In defining class Box<E>

1.Only local variables and non-static methods can use type E.
static E obj; Not allowed!

2.DO NOT invoke E's constructor or methods.
E obj = new E();
E[] obj = new E[10];
E obj = (E)(new Object());

Containers: Java Generics

In defining class Box<E>

1.Only local variables and non-static methods can use type E.
static E obj; Not allowed!

2.DO NOT invoke E's constructor or methods.
E obj = new E(); Not allowed!
E[] obj = new E[10];
E obj = (E)(new Object());

Containers: Java Generics

In defining class Box<E>

1.Only local variables and non-static methods can use type E.
static E obj; Not allowed!

2.DO NOT invoke E's constructor or methods.
E obj = new E(); Not allowed!
E[] obj = new E[10]; Not allowed!
E obj = (E)(new Object());

Containers: Java Generics

In defining class Box<E>

1.Only local variables and non-static methods can use type E.
static E obj; Not allowed!

2.DO NOT invoke E's constructor or methods.
E obj = new E(); Not allowed!
E[] obj = new E[10]; Not allowed!
E obj = (E)(new Object()); Allowed

Containers: Java Generics

In defining class Box<E>

1.Only local variables and non-static methods can use type E.
static E obj; Not allowed!

2.DO NOT invoke E's constructor or methods.
E obj = new E(); Not allowed!
E[] obj = new E[10]; Not allowed!
E obj = (E)(new Object()); Allowed

3. Inheritance is NOT preserved.
String extends Object, but
Box<String> DOES NOT extend Box<Object>

Containers: Java Generics

In defining class Box<E>

1.Only local variables and non-static methods can use type E.
static E obj; Not allowed!

2.DO NOT invoke E's constructor or methods.
E obj = new E(); Not allowed!
E[] obj = new E[10]; Not allowed!
E obj = (E)(new Object()); Allowed

3. Inheritance is NOT preserved.
String extends Object, but
Box<String> DOES NOT extend Box<Object>

Box<String> box1 = new Box<String>(); Box<Object> box2 = box1;

Containers: Java Generics

In defining class Box<E>

1.Only local variables and non-static methods can use type E.
static E obj; Not allowed!

2.DO NOT invoke E's constructor or methods.
E obj = new E(); Not allowed!
E[] obj = new E[10]; Not allowed!
E obj = (E)(new Object()); Allowed

3. Inheritance is NOT preserved.
String extends Object, but
Box<String> DOES NOT extend Box<Object>

Box<String> box1 = new Box<String>(); Box<Object> box2 = box1; Not allowed!

Containers: Java Generics

Generic Association

Recall: An association binds a key object with a value object.

Containers: Java Generics

Generic Association

Recall: An association binds a key object with a value object.

```
public class Association<K,V>
{
    protected K theKey; // the key
    protected V theValue; // the value
    public Association(K key, V value)
    // pre: key is non-null
    // post: constructs a key-value pair
    public V getValue()
    // post: returns value from association
    public K getKey()
    // post: returns key from association
    public V setValue(V value)
    // post: sets association's value to value
}
```

Containers: Java Generics

Generic Association

Recall: An association binds a key object with a value object.

```
public class Association<K,V>
{
    protected K theKey; // the key
    protected V theValue; // the value
    public Association(K key, V value)
    // pre: key is non-null
    // post: constructs a key-value pair
    public V getValue()
    // post: returns value from association
    public K getKey()
    // post: returns key from association
    public V setValue(V value)
    // post: sets association's value to value
}
```

```
Association<String,Integer> personAttribute =
   new Assocation<String,Integer>("Age",34);
```

Containers: Java Generics

Generic Numerical Association

A numerical association is an association where the key is a number.

Containers: Java Generics

Generic Numerical Association

A numerical association is an association where the key is a number.

```
public class NumericalAssociation<K extends Number,V>
       extends Association<K,V>
{
    public NumericalAssociation(K key, V value)
    // pre: key is a non-null value of type K
    // post: constructs a key-value pair
    {
        super(key,value);
    }
    public int keyValue()
    // post: returns the integer that best approximates the key
    {
        return getKey().intValue();
    }
}
```

```
NumericalAssociation<Integer,String> i;
NumericalAssociation<Double,Integer> d;
NumericalAssociation<Number,Object> n;
NumericalAssociation<BigInteger,Association<String,Object>> I;
```

Week 1: Object Orientation and Data Structures

Part III: Java Collections

Collections

A collection is a container class that stores an unspecified number of data items. A collection allows the user to:

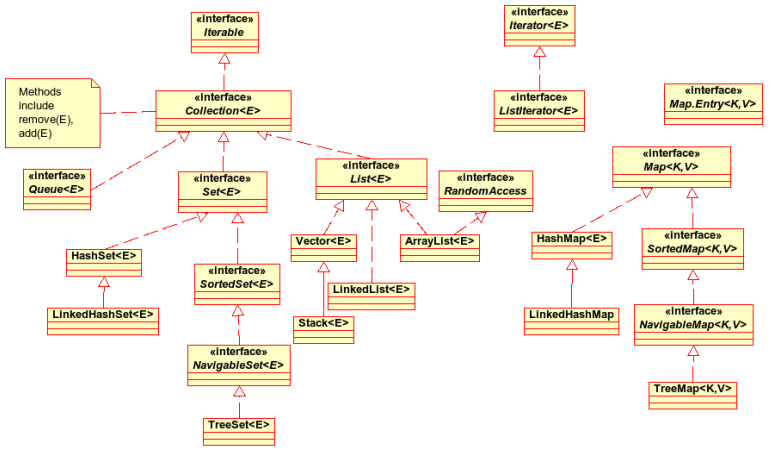
- accessor: query membership of an element in the container
- mutators:
  - add an element to the container
  - remove an element from the container

Collections

A collection is a container class that stores an unspecified number of data items. A collection allows the user to:

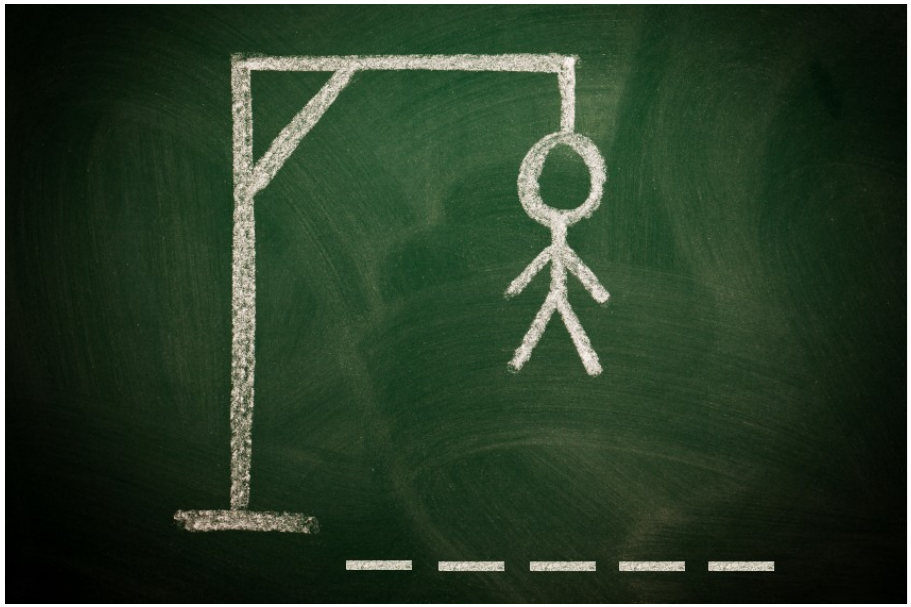
- accessor: query membership of an element in the container
- mutators:
  - add an element to the container
  - remove an element from the container

The Java Collection Framework provides a range of collection classes for use, i.e., ArrayList, LinkedList, Stack, HashSet, HashMap, etc.



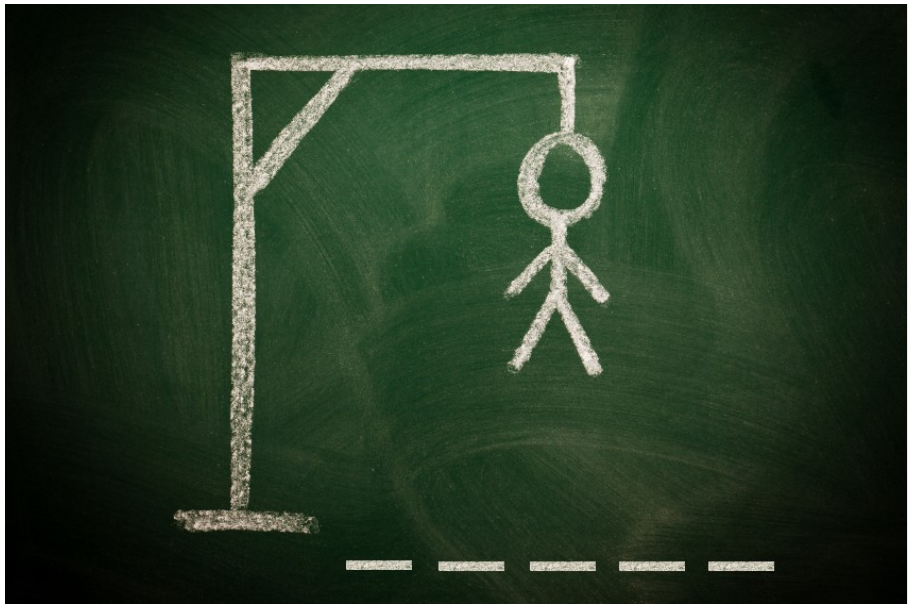
Collections

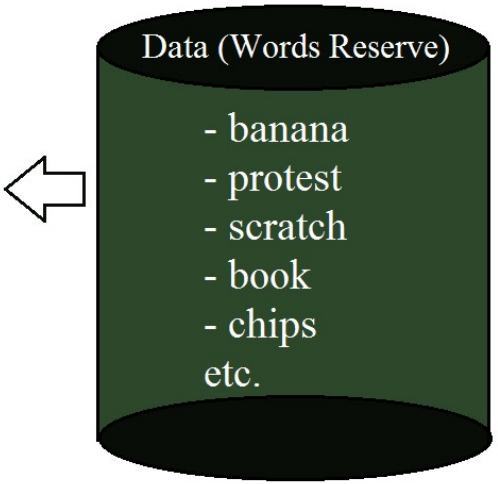
Example 4: Developing a Hangman game



Collections

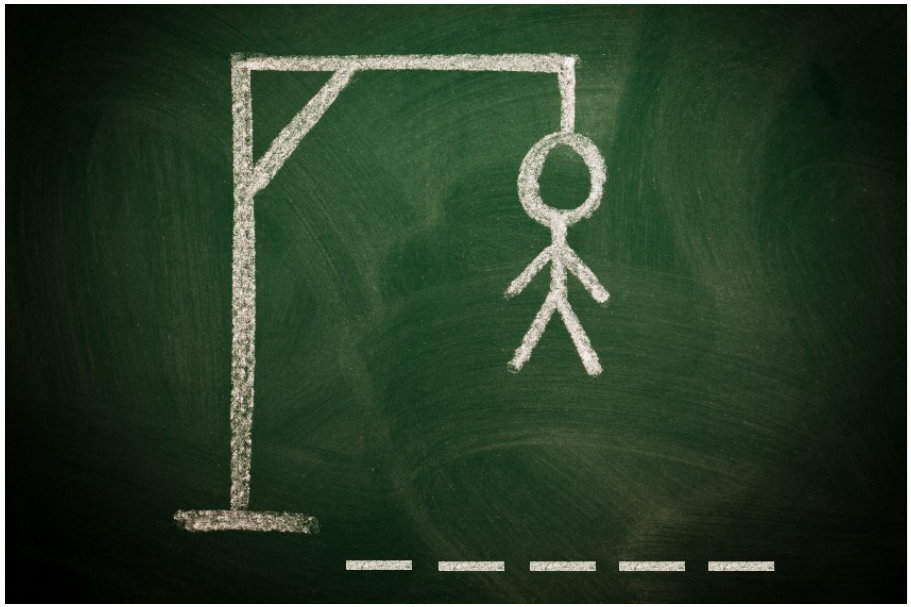
Example 4: Developing a Hangman game

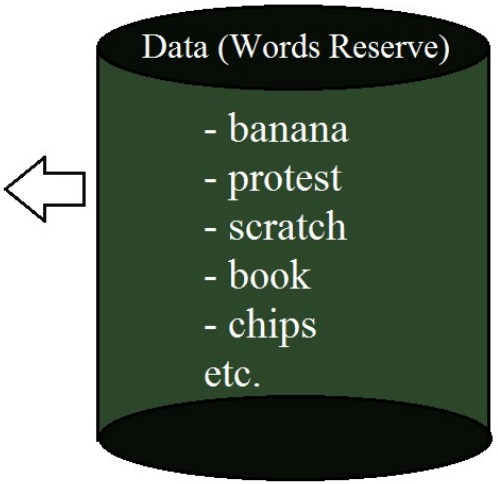




Collections

Example 4: Developing a Hangman game

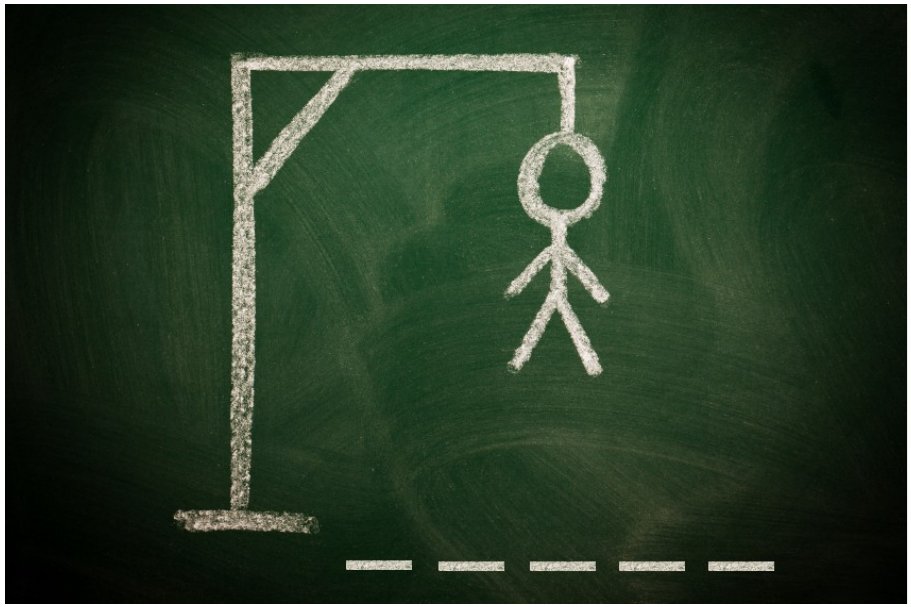


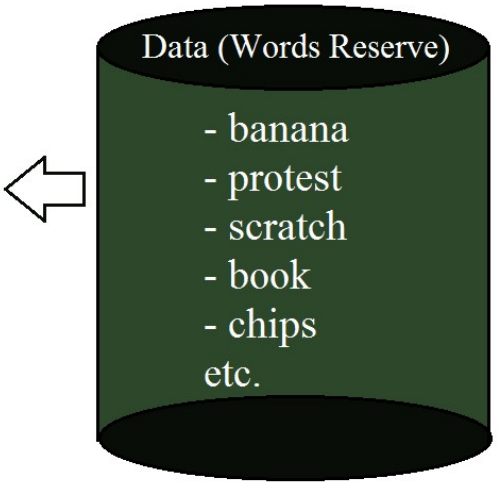


Task: Develop a data structure for the words reserve

Collections

Example 4: Developing a Hangman game





Task: Develop a data structure for the words reserve

Operations:

-Add a word

-Remove a word

-Select a random word

-Query if the words reserve is empty

Collections

Example 4: Developing a Hangman game

Task: Develop a data structure for the words reserve

```
public class WordList
{
    public WordList(int size)
    // pre: size >= 0
    // post: construct a word list capable of holding "size" words

    public boolean isEmpty()
    // post: return true iff the word list contains no words

    public void add(String s)
    // post: add a word to the word list, if it is not already there

    public String selectAny()
    // pre: the word list is not empty
    // post: return a random word from the list

    public void remove(String word)
    // pre: word is not null
    // post: remove the word from the word list
}
```

Collections

Example 4: Developing a Hangman game

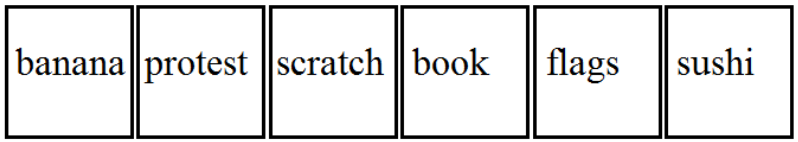
To implement the WordList class, we may use the ArrayList class provided by Java. An array list is a data structure which stores elements in an array, while allowing adding and deleting elements.

Collections

Example 4: Developing a Hangman game

To implement the WordList class, we may use the ArrayList class provided by Java. An array list is a data structure which stores elements in an array, while allowing adding and deleting elements.

- Every element has a position index (between 0 and $n - 1$)
- Accessing an element by get(i)
- If adding an element at position i, shift elements $i + 1, \ldots, n - 1$ to the right by one position
- If deleting an element at position i, shift all elements
- $i + 1, \ldots, n - 1$ to the left by one position



Collections

Example 4: Developing a Hangman game

```
import java.util.*;
public class WordList{
    protected ArrayList<String> theList;
    protected Random generator;
    public WordList(int size){
        theList = new ArrayList<String>(size);
        generator = new Random();
    }
    public boolean isEmpty(){
        return theList.isEmpty();
    }
    public void add(String s){
        theList.add(s);
    }
    public String selectAny(){
        int i=Math.abs(generator.nextInt())%theList.size();
        return (String)theList.get(i);
    }
    public void remove(String word){
        theList.remove(word);
    }
}
```

Collections

Example 4: Developing a Hangman game

Once the word list is created, we may use it in the game:

```
public class HangMan{
    public static void main (String[] args){
        WordList list;
        String targetWord;
        list=new WordList(10);
        list.add("banana");
        list.add("protest");
        list.add("scratch");
        list.add("apple");
        list.add("blanket");
        while(!list.isEmpty()){
            targetWord=list.selectAny();
            play(targetWord);
            list.remove(targetWord);
        }
    public static void play(String target){
        Game logic goes here...
    }
}
```

Interfaces

A Note on Usability

It may be useful to describe the interface for a number of different classes, without committing to an implementation.

Example. We always need to manage collections of data with the following operations:

- Add a data item
- Remove a data item
- Query about the size of the collection
- Check for emptiness

Hence we create an interface for a collection with these methods.

Interfaces

Structure

```
public interface Structure
{
    public int size();
    public boolean isEmpty();
    public void clear();
    public boolean contains(Object value);
    public void add(Object value);
    public Object remove(Object value);
    public java.util.Enumeration elements();
    public Iterator iterator();
    public Iterator iterator();
    public Collection values();
}
```

Interfaces

Structure

```
public interface Structure
{
    public int size();
    public boolean isEmpty();
    public void clear();
    public boolean contains(Object value);
    public void add(Object value);
    public Object remove(Object value);
    public java.util.Enumeration elements();
    public Iterator iterator();
    public Iterator iterator();
    public Collection values();
}
```

When creating the class WordList:

```
public class WordList implements
Structure
```

Interfaces

Set

```
public interface Set extends Structure
{
    public void addAll(Structure other);
    // pre: other is non-null
    // post: values from other are added into this set
    public boolean containsAll(Structure other);
    // pre: other is non-null
    // post: returns true if every value in set is in other

    public void removeAll(Structure other);
    // pre: other is non-null
    // post: values of this set contained in other are removed

    public void retainAll(Structure other);
    // pre: other is non-null
    // post: values not appearing in the other structure are removed
}
```

Week 1: Object Orientation and Data Structures

Part IV: Iterators

Iterators

Goal: Going through all elements of a collection.

Iterators

Goal: Going through all elements of a collection.

Example: Iterating an array

```
for(int i=1; i<a.length;i++)
```

Some operation on `a[i]`

Iterators

Goal: Going through all elements of a collection.

Example: Iterating an array

```
for(int  i=1;  i<a.length;i++)
Some operation on a[i]
```

Note:

- An array assumes a pre-defined ordering of its elements (indices of elements)
- For an arbitrary collection (data structure), such an ordering is often not explicitly defined
- To go through all elements, one would need to traverse the data structure – assign an ordering

Iterators

Iterators

An iterator is a control structure that defines an order of traversal in a data structure.

An iterator must be defined on a pre-defined data structure. An iterator hides the complexities of the traversal.

Iterators

Iterators

An iterator is a control structure that defines an order of traversal in a data structure.

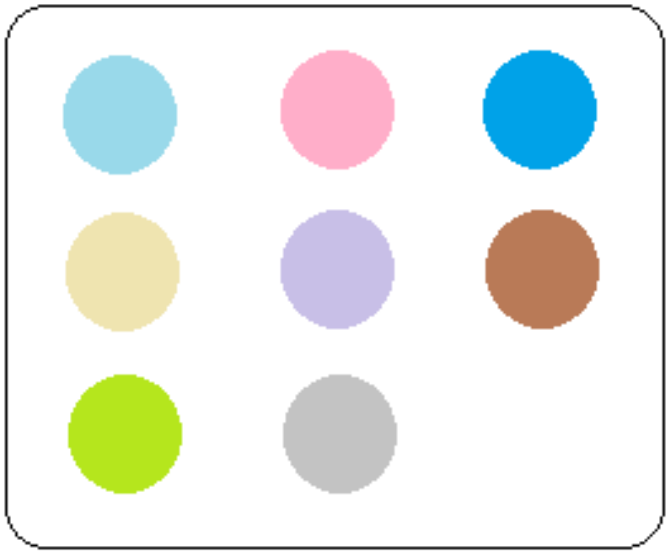
An iterator must be defined on a pre-defined data structure. An iterator hides the complexities of the traversal.

Iterators

Iterators

An iterator is a control structure that defines an order of traversal in a data structure.

An iterator must be defined on a pre-defined data structure. An iterator hides the complexities of the traversal.



Iterators

Iterators

An iterator is a control structure that defines an order of traversal in a data structure.

An iterator must be defined on a pre-defined data structure. An iterator hides the complexities of the traversal.

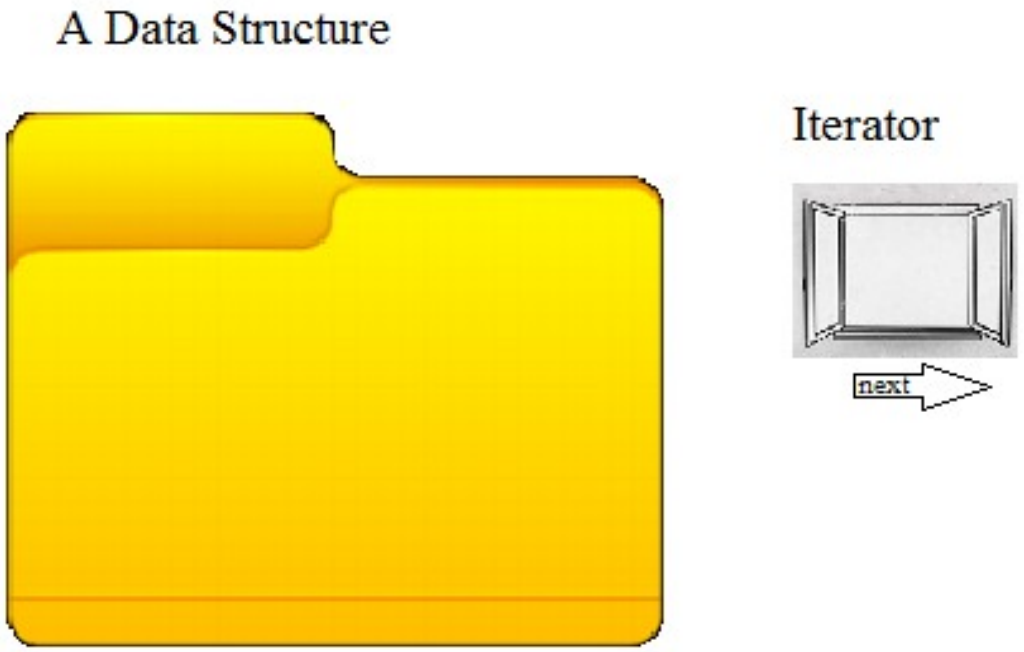
A Data Structure

Iterators

Iterators

An iterator is a control structure that defines an order of traversal in a data structure.

An iterator must be defined on a pre-defined data structure. An iterator hides the complexities of the traversal.

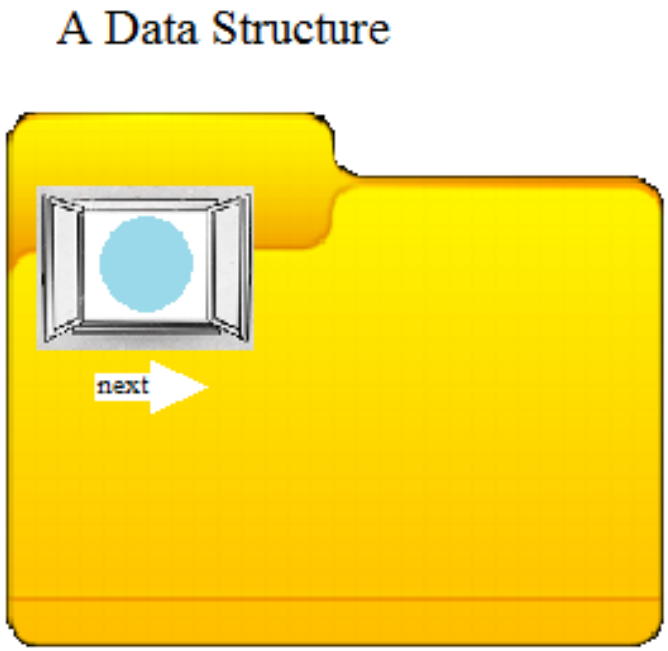


Iterators

Iterators

An iterator is a control structure that defines an order of traversal in a data structure.

An iterator must be defined on a pre-defined data structure. An iterator hides the complexities of the traversal.

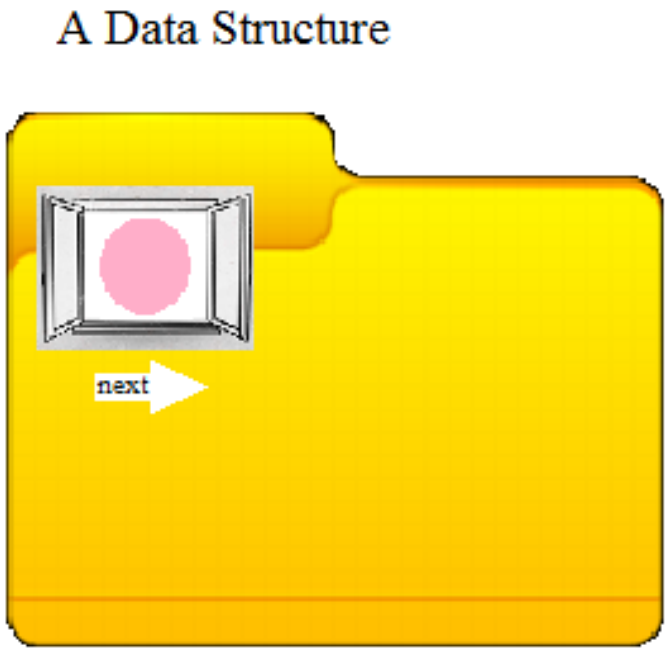


Iterators

Iterators

An iterator is a control structure that defines an order of traversal in a data structure.

An iterator must be defined on a pre-defined data structure. An iterator hides the complexities of the traversal.

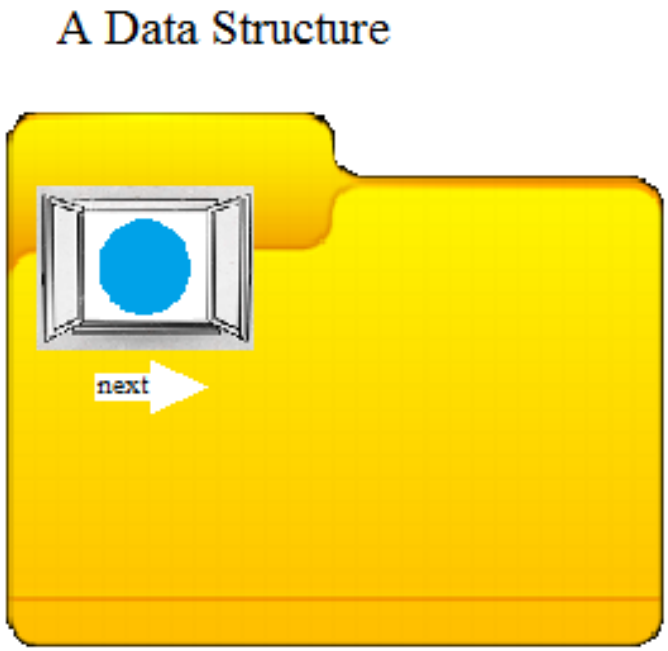


Iterators

Iterators

An iterator is a control structure that defines an order of traversal in a data structure.

An iterator must be defined on a pre-defined data structure. An iterator hides the complexities of the traversal.

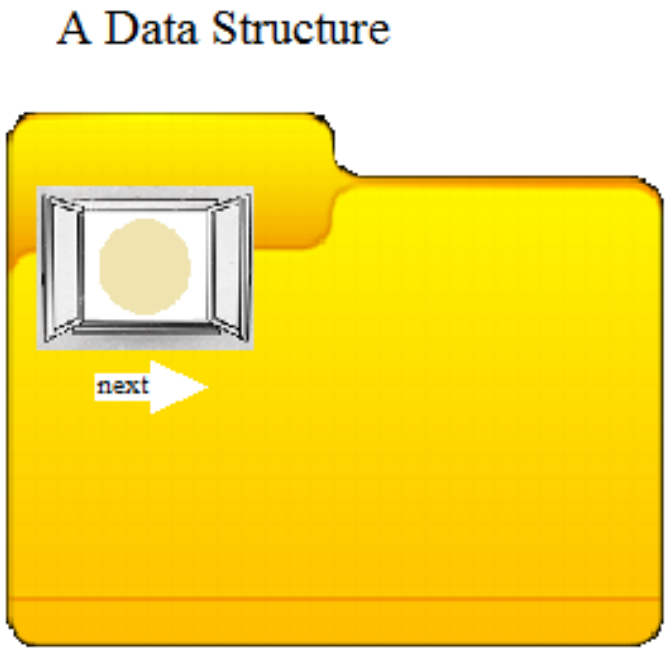


Iterators

Iterators

An iterator is a control structure that defines an order of traversal in a data structure.

An iterator must be defined on a pre-defined data structure. An iterator hides the complexities of the traversal.

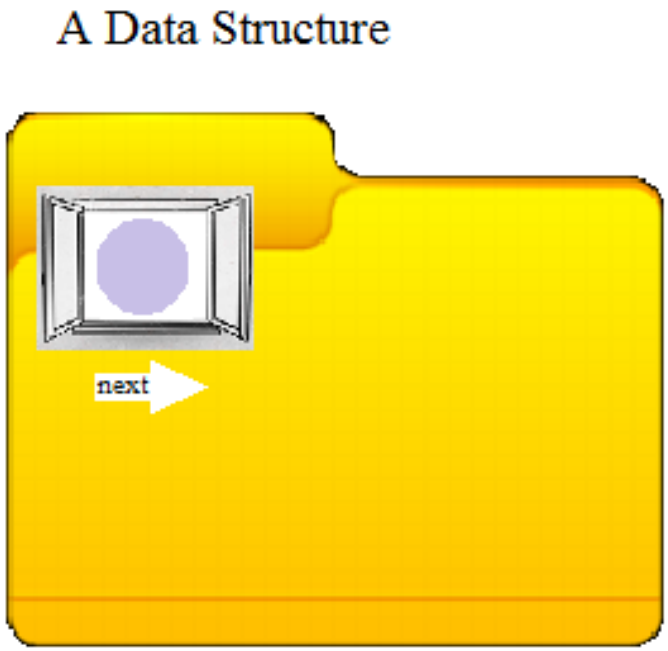


Iterators

Iterators

An iterator is a control structure that defines an order of traversal in a data structure.

An iterator must be defined on a pre-defined data structure. An iterator hides the complexities of the traversal.

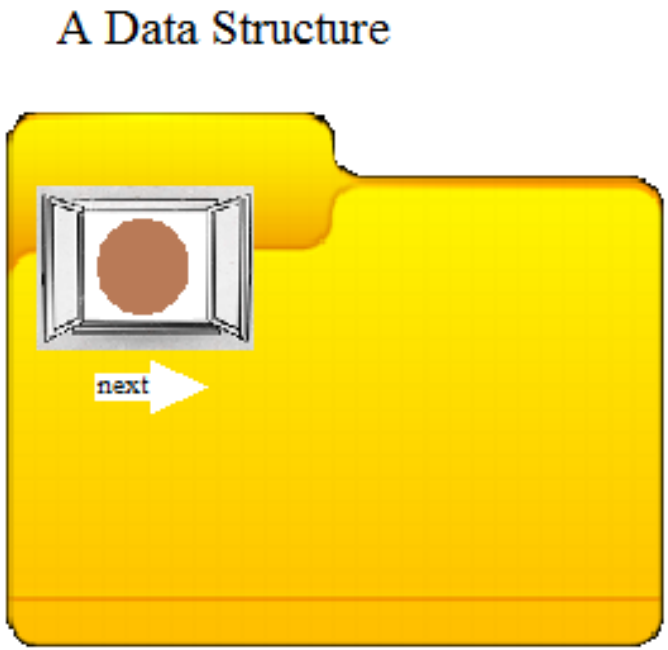


Iterators

Iterators

An iterator is a control structure that defines an order of traversal in a data structure.

An iterator must be defined on a pre-defined data structure. An iterator hides the complexities of the traversal.

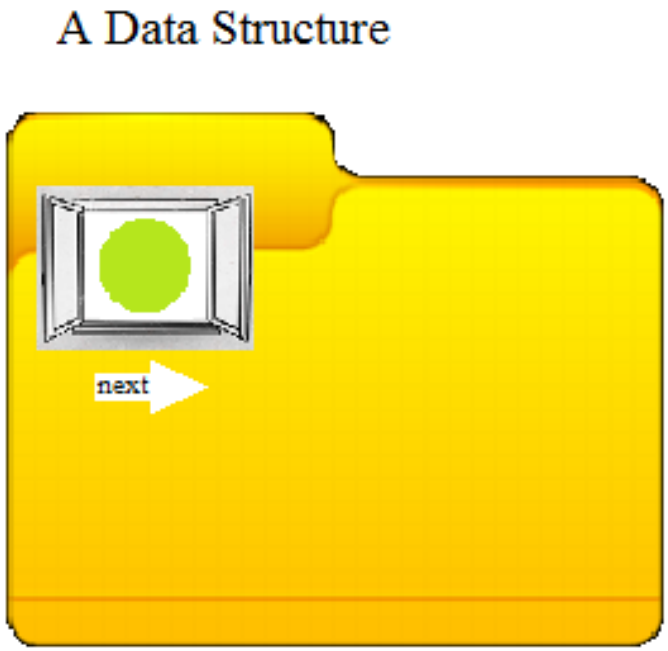


Iterators

Iterators

An iterator is a control structure that defines an order of traversal in a data structure.

An iterator must be defined on a pre-defined data structure. An iterator hides the complexities of the traversal.

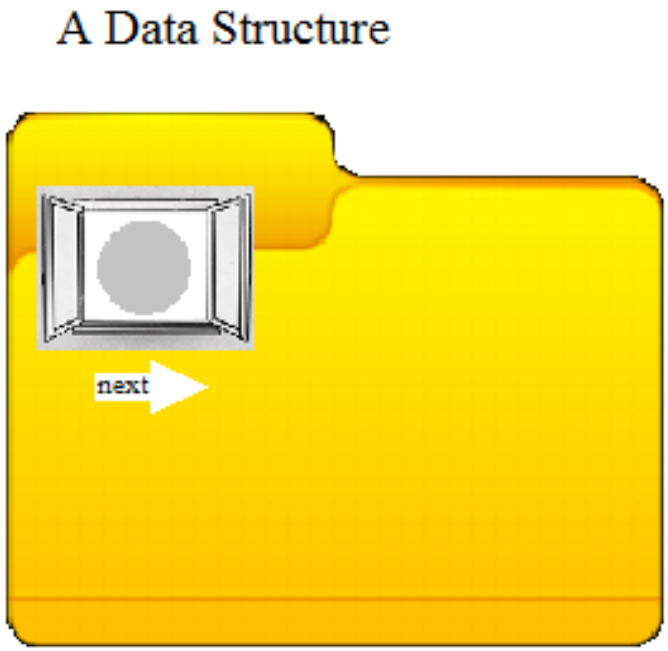


Iterators

Iterators

An iterator is a control structure that defines an order of traversal in a data structure.

An iterator must be defined on a pre-defined data structure. An iterator hides the complexities of the traversal.

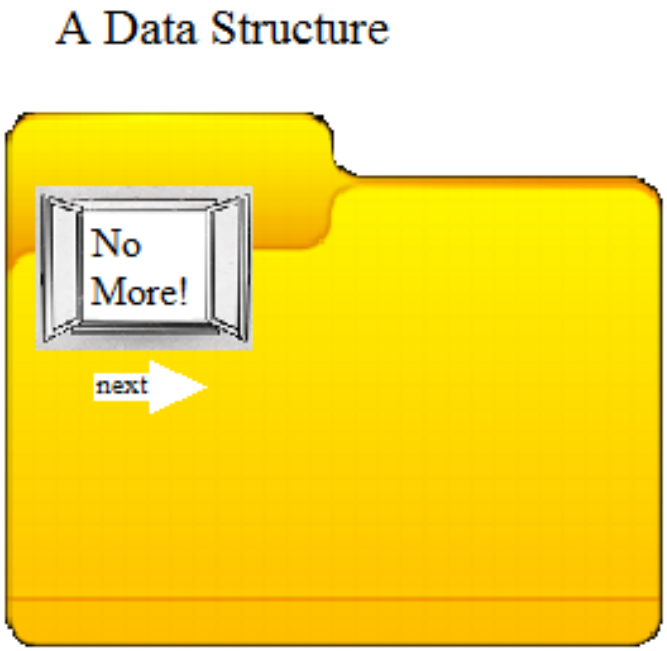


Iterators

Iterators

An iterator is a control structure that defines an order of traversal in a data structure.

An iterator must be defined on a pre-defined data structure. An iterator hides the complexities of the traversal.

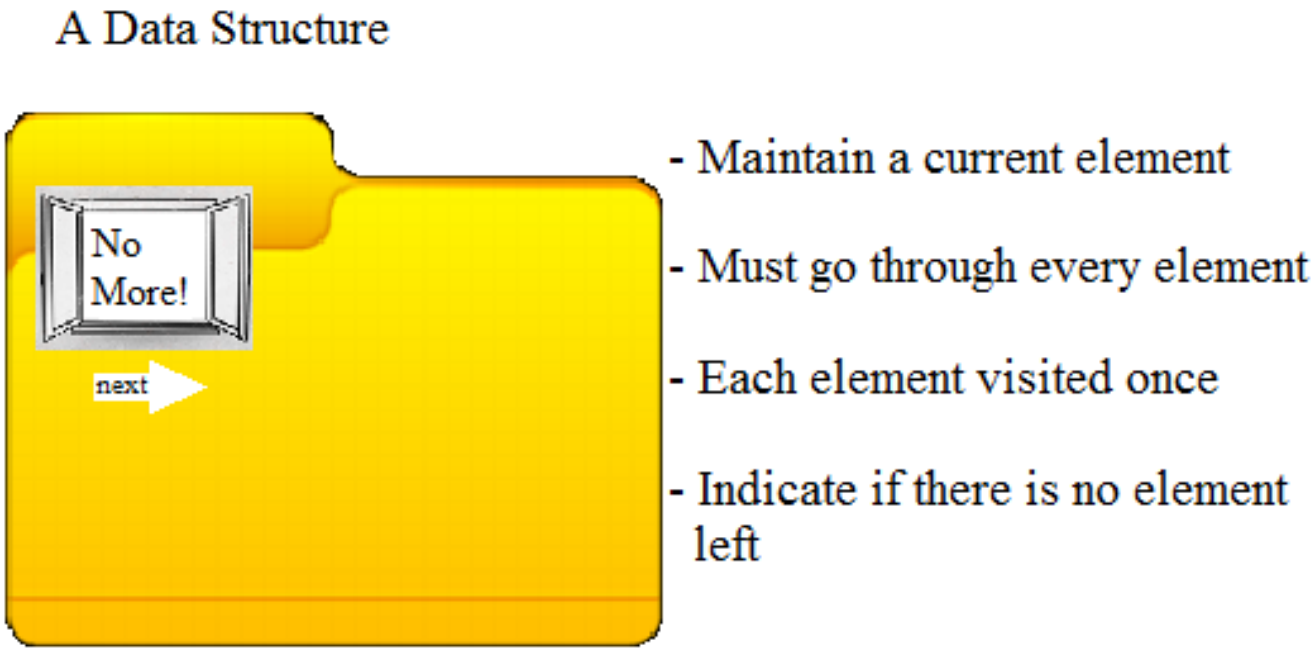


Iterators

Iterators

An iterator is a control structure that defines an order of traversal in a data structure.

An iterator must be defined on a pre-defined data structure. An iterator hides the complexities of the traversal.



Iterators

The Java Iterator Interface

```
public interface
java.util.Iterator<E>{

        public boolean hasNext();
        //  post: returns true if the iterator has
        //  more elements

        public E next();
        //  post: returns the next element in the iteration

}
```

Iterators

Example 5: WordList Iterator

Add a get(int i) method to WordList.

```
public class WordListIterator<E> implements Iterator<E>{
protected WordList data;
protected int current;
public WordListIterator(WordList list){ data = list;
current=0;
}
public boolean hasNext(){
return current < data.size();
}
public E next(){
return data.get(current++);
}
}
```

Iterators

The Iterable Interface

```
public interface Java.lang.Iterable<T>{

    public Iterator<T> iterator();
    //  post: returns an iterator over elements of type T
}
```

Iterators

The Iterable Interface

```
public interface Java.lang.Iterable<T>{

    public Iterator<T> iterator();
    //  post: returns an iterator over elements of type T
}
```

To use a WordListIterator, we

- make WordList implement Iterable interface; and
- add in WordList an iterator() methodand

Iterators

The Iterable Interface

```
public interface Java.lang.Iterable<T>{

    public Iterator<T> iterator();
    //  post: returns an iterator over elements of type T
}
```

To use a WordListIterator, we:

- make WordList implement Iterable interface; and
- add in WordList an iterator() method

```
public class WordList implements Iterable<E> {
...
public Iterator<E> iterator()
// post:          return an iterator of this vector
{
        return new
  }     WordListIterator(this);
}
```

Iterators

Example 5: WordList Iterator

```
WordList list= new WordList(); list.add("Hello");
list.add("world");
Iterator<String> it=list.iterator();
```

Iterators

Example 5: WordList Iterator

```
WordList list= new WordList(); list.add("Hello");
list.add("world");
Iterator<String> it=list.iterator();
```

Note. There are two ways to print out all words in list:

- while(it.hasNext()) System.out.print(i.next()+" ");
- for (String word : list) System.out.print(word+" ");

Since Vector now implements Iterable, we can use for-loop in this way for v.