Week 5

Lists

Lists

Lists are data structures that:

- Store a sequence of elements - like an array.
- Allow new elements to be added - unlike an array!
- Allow elements to be removed - unlike an array!

They are implemented as classes which you import:

```
import java.util.*;
```

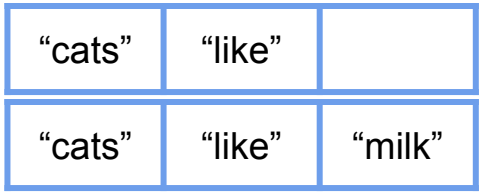
Examples

- A bank has many customers. You can add and remove customers.
- Diagram has many shapes. You can add and remove shapes.
- A deck has many cards. You can add and remove cards.

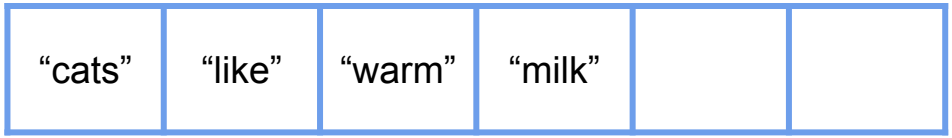
Array lists

- An array list uses an array internally, with extra space at the end...

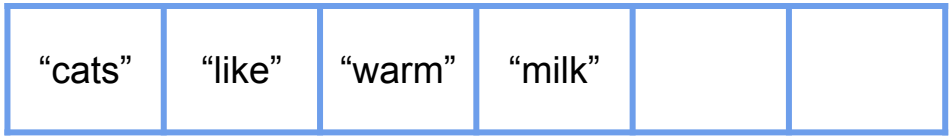
- ... so that you have room to add more elements



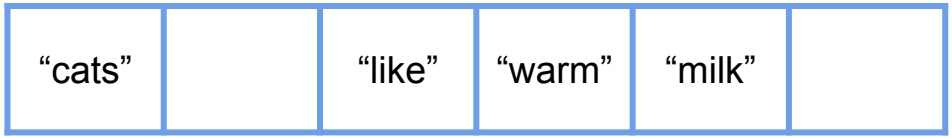
When you run out of space, a bigger array is created and the elements copied across:



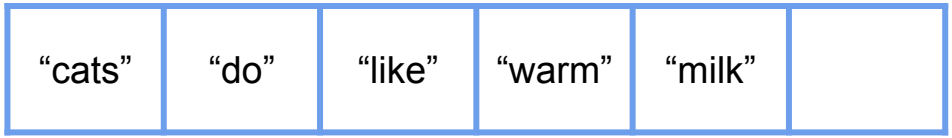
Array lists



- To insert an element, you must shift elements to the right



- Then insert



Should I use an array list?

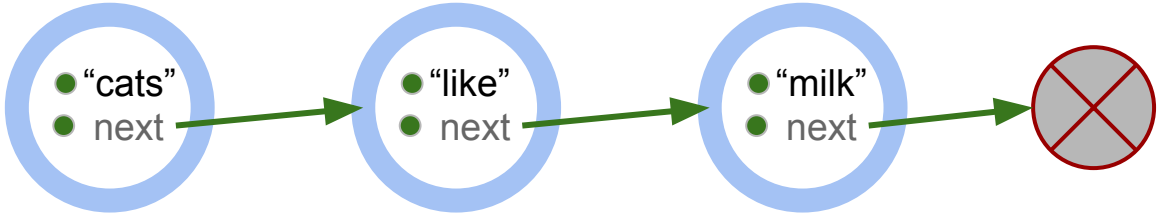
- Array lists provide instant access to any element. They are FAST.
- Adding elements to the end of an array list is reasonably fast.
- Inserting elements near the beginning of a list is slow.

Use an array list if you need random access to elements.

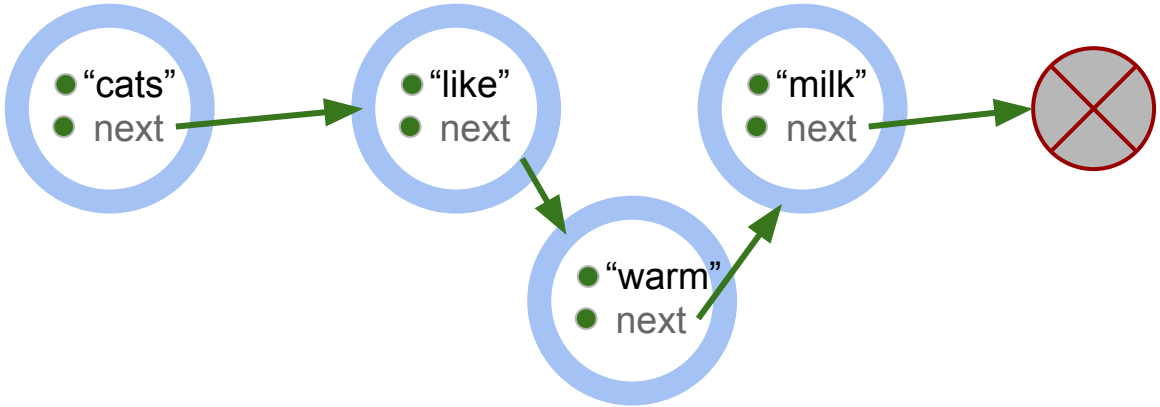
Don’t use an array list if you often need to insert elements near the beginning.

Linked lists

- Elements are stored in objects that are linked together



- To insert an element, just change two arrows



Should I use a linked list?

- Linked lists provide SLOW access to random elements.
- Adding elements to the beginning or end is FAST.
- Linked lists require more memory to store the “links”.

Use a linked list if you add and remove elements often.

Don’t use a linked list if you need fast random access to any element.

Don’t use a linked list if you have a large data set and limited memory.

Usage prior to Java 5

- In the good ole' days, using a list was simple:

```
LinkedList list = new LinkedList();
list.add("cats");
list.add("like");
list.add("milk");
```

- But unsafe:

```
list.add(new Burger());
list.add(new Dog());
```

- Java 4 and earlier could not restrict elements to a particular type.

Generics

- Java 5 added “generics”, also known as “type parameters”:

```
LinkedList<String> list = new LinkedList<String>();
list.add(“cats”);
list.add(“like”);
list.add(“milk”);
```

- We pass <String> as the type parameter to the class LinkedList.
- Now, a linked list of <String> will permit only string elements:

```
list.add(new Burger());   ← compile error
```

- Safe!

Type parameters vs Method parameters

- Method parameters go after a method and use round brackets:

```
System.out.println("zoo");
repeat(5, "* ");
```

- Type parameters go after a type and use angled brackets:

```
LinkedList<Customer> customers;
ArrayList<Card> cards;
TreeSet<String> symbols;
```

- Type parameters must be classes. For primitives, use class wrappers:

```
LinkedList<Integer> ages;
ArrayList<Double> rainfall;
```

LinkedList‹X› and ArrayList‹X› methods

Method	Description
add(X element)	Add an element of type X to the end
add(int i, X element)	Add an element of type X at position i
remove(X element)	Remove this element
remove(int i)	Remove the element at position i
set(int i, X element)	Replace the element at position i
X get(int i)	Return the element at position i
int size()	Return the size of the list
clear()	Remove all elements

For more, see: https://docs.oracle.com/javase/7/docs/api/java/lang/List.html

Looping over a list

Use a for-each loop.

```
LinkedList<String> words = new LinkedList<String>();
words.add(“one”);
words.add(“two”);
words.add(“three”);

for (String word : list)
    System.out.println(word);
```

Copying a list

```
LinkedList<String> original = new LinkedList<String>();
-- add elements to original --

LinkedList<String> copy = new LinkedList<String>();
for (String word : original)
    copy.add(word);
```

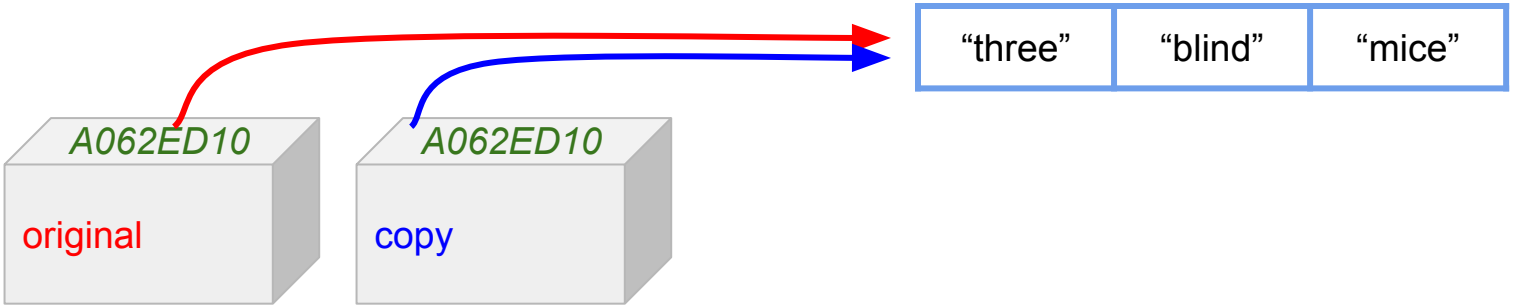
You now have two lists that contain the same elements.

Copying a list - NOT

```
LinkedList<String> original = new LinkedList<String>();
-- add elements to original --

LinkedList<String> copy = original;
```

You now have two variables that point to the **same list**!



Copying a list with addAll

Method	Description
addAll(Collection<X> elements)	Add a collection of elements to this list

```
LinkedList<String> original = new LinkedList<String>();
-- add elements to original --

LinkedList<String> copy = new LinkedList<String>();
copy.addAll(original);
```

A list of accounts

Specification: A customer has a list of accounts. The customer can add an account or view a list of accounts.

Supplier solution

```
public class Account {
  private String type;
  private double balance;
  public Account() {
    type = readType();
    balance = readBalance();
  }
  public void use() {
    char choice;
    while ((choice = readChoice()) != ‘x’) {
      switch (choice) {
      case ‘d’: deposit(); break;
      case ‘w’: withdraw(); break;
      case ‘v’: view(); break;
      default: help(); break;
      }
    }
  }
  private void deposit() {
    balance += readAmount();
  }
  private void withdraw() {
    balance -= readAmount();
  }
  public void view() {
    System.out.println(this);
  }
  @Override
  public String toString() {
    return type + “ account has $” + formatted(balance);
  }

  // read and format functions not shown
}
```

Client solution

```
public class Customer {
    public static void main(String[] args) { new Customer().use(); }

    private LinkedList<Account> accounts = new LinkedList<Account>();
    public void use() {
        char choice;
        while ((choice = readChoice()) != 'x') {
        case 'a': add(); break;
        case 'v': view(); break;
        default: help(); break;
        }
    }
    ...
```

Client solution

```
    public void add() {
        customers.add(new Customer());
    }

    public void view() {
        for (Customer customer : customers)
            customer.view();
    }
}
```

The “lookup” pattern

Goal: Find and return an element in a list. Return null if not found.

```
for (<element type> <item> : <list>)
    if (<this is the item I want>)
        return <item>;
return null;
```

Example: Find a particular kind of account. e.g. account(“Savings”)

```
private Account account(String type) {
    for (Account account : accounts)
        if (type.equals(account.getType()))
            return account;
    return null;
}
```

The name of a lookup function is the name of what is being returned.

The match function

- Design rules
  - Code is defined in the same class as the fields.
  - Hide the detail, export the behaviour.
  - **Push it right**, from client to supplier.

Client

```
public class Customer {
  private Account account(String type) {
    for (Account account : accounts)
      if (type.equals(account.getType()))
        return account;
    return null;
  }
}
```

Supplier

```
public class Account {
    private String type;

    ...

}
```

The match function

- Push the “match” code into the supplier:

Client

```
public class Customer {
  private Account account(String type) {
    for (Account account : accounts)
      if (account.hasType(type))
        return account;
    return null;
  }
}
```

Supplier

```
public class Account {
  private String type;

  public boolean hasType(String type) {
    return type.equals(this.type);
  }
}
```

Select an account

Specification: A customer has a list of accounts. The customer can add an account or view a list of accounts.

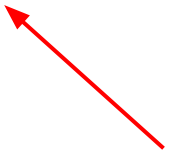
A customer can also select an account to use and remove an account.

The idea:

- To use an account, we need to look it up in the list.
- To remove an account, we need to look it up in the list.

Supplier solution

```
public class Account {
    private String type;
    ...
    public boolean hasType(String type) {
        return type.equals(this.type);
    }
}
```



The match function

Client solution

```
public class Customer {
    public void use() {
        char choice;
        while ((choice = readChoice()) != 'x') {
            switch (choice) {
            case 's': select(); break;
            case 'r': remove(); break;
            ...
        }
    }
    ...
```

Client solution

```
private void select() {
     Account account = account(readType());
     if (account != null)
          account.use();
     else
          System.out.println("Account not found");
}

private boolean account(String type) {
     for (Account account : accounts)
          if (account.hasType(type))
               return account;
     return null;
}
```

The lookup function

Client solution

```
    private void remove() {
        Account account = account(readType());
        if (account != null)
            accounts.remove(account);
        else
            System.out.println("Account not found");
    }
}
```

Adding unique elements

- **Specification**: Each account is uniquely identified by its type. Add an account only if no account of that type already exists.

```
public void add() {
    // Create a new account
    Account account = new Account();
    // Check if there is an existing account of that type
    Account existingAccount = account(account.getType());
    if (existingAcount == null)
        accounts.add(account);
    else
        System.out.println(account.getType() + " account already exists");
}
```

Find all matches

Specification: Find all words in a list that contain “z”.

Solution: Create a new list and add the matching words.

```
private LinkedList<String> zWords(LinkedList<String> words) {
     LinkedList<String> matches = new LinkedList<String>();
     for (String word : words)
          if (word.contains(“z”))
               matches.add(word);
     return matches;
}
```

Remove all matches

- **Specification**: Remove all words in a list that contain a ‘z’.

This solution does not work:

```
for (String word : list)
    if (word.contains(“z”))
        list.remove(word);
```

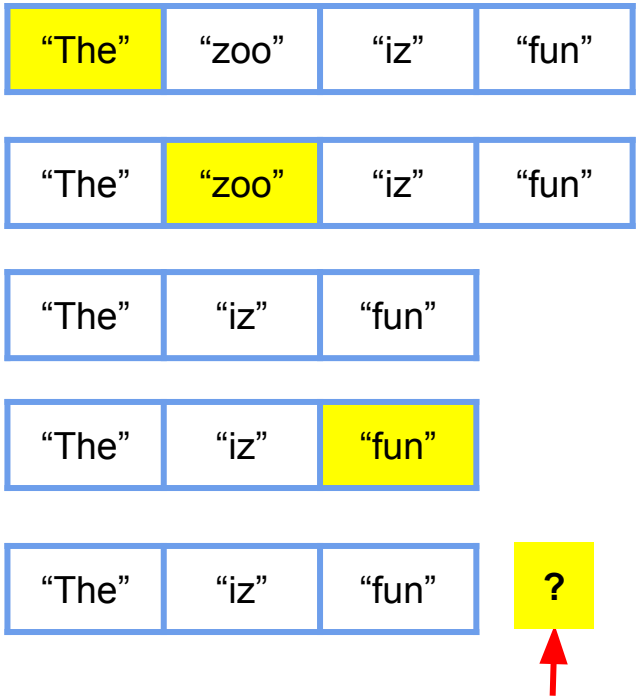
Why?

Remove all matches - there are 4 elements

- Look at the **first** element. It doesn't match.
- Look at the **second** element. It matches.

So delete it.

- Look at the **third** element. It doesn't match.
- Look at the **fourth** element. CRASH.
  (there is no fourth element)



Remove all matches - correct solution

- Find a list of z words first:

```
LinkedList<String> zWords = zWords(list);
```

- Now remove them from the original list:

```
list.removeAll(zWords);
```