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.*;
```

Two common implementations: `ArrayList` and `LinkedList`

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

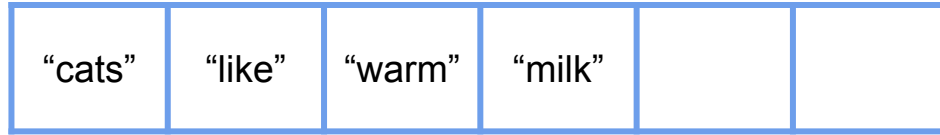- An array list uses an array internally, with extra space at the end...

"cats"		

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

"cats"	"like"	

"cats"	"like"	"milk"
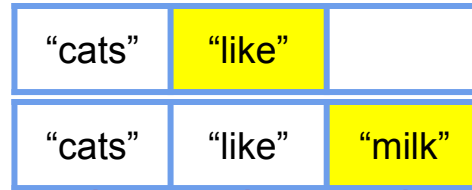
- When you run out of space, a bigger array is created and the elements copied across:

"cats"	"like"	"milk"	"every"	"day"	

Array lists

"cats"	"like"	"warm"	"milk"		

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

"cats"		"like"	"warm"	"milk"	

- Then insert

| "cats"	"do"	"like"	"warm"	"milk"	
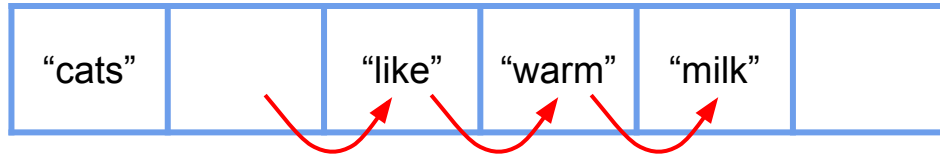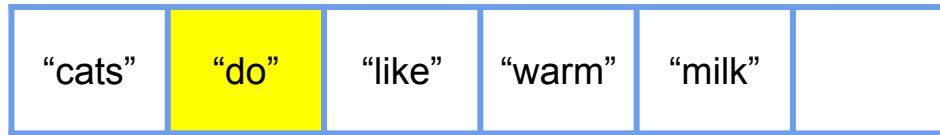
Should I use an array list?

- 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.

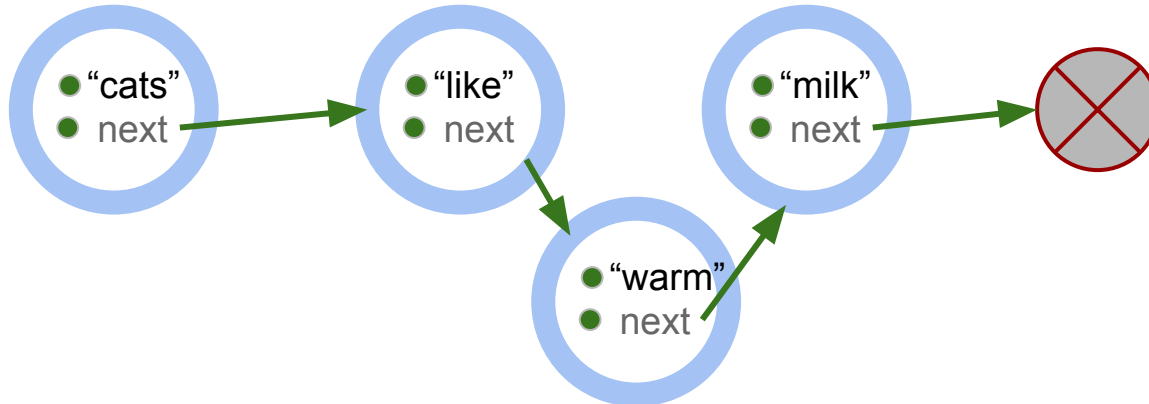Use an array list if you need random access to elements.

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
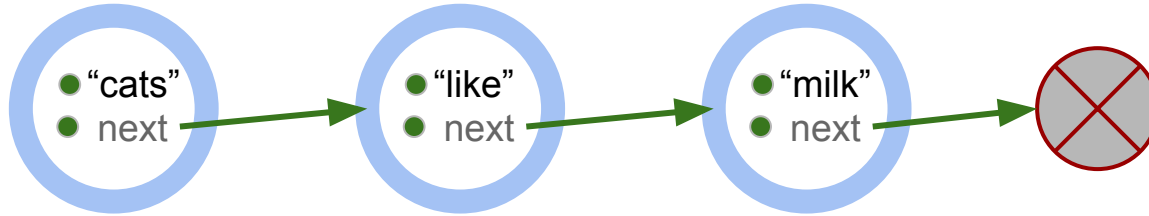
- To insert an element, just change two arrows

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 : words)
    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);
```
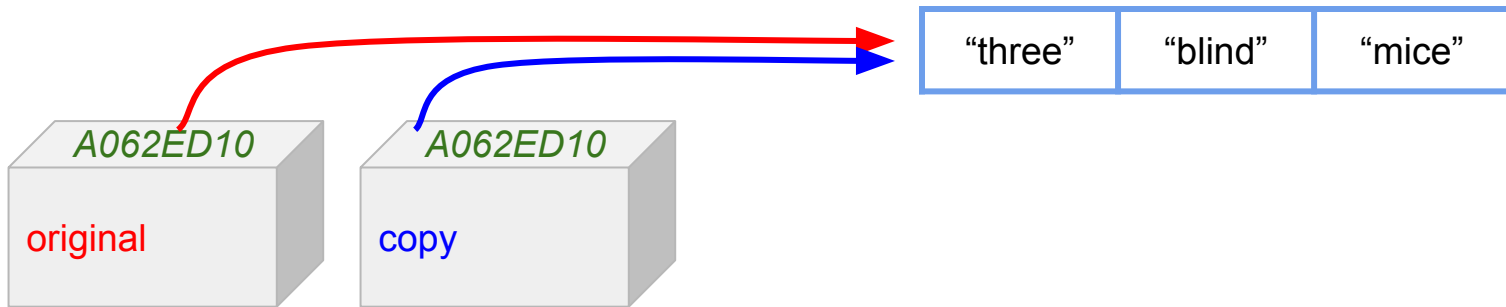
You now have two lists that contain the same elements.

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**!

"three"	"blind"	"mice"

A062ED10

original

A062ED10

copy

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(String type) {
    this.type = type;
    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;
        }
    }
    public void add() {
        accounts.add(new Account(readType()));
    }
    public void view() {
        for (Account account : accounts)
            account.view();
    }
    ...
}
```
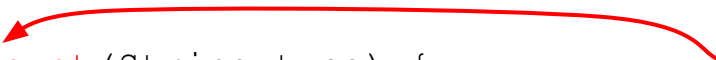
The "lookup" pattern

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

```
<for each item in the 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 Account 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() {
    String type = readType();
    // Check if there is an existing account of that type
    Account existingAccount = account(type);
    if (existingAcount == null)
        accounts.add(new Account(type));
    else
        System.out.println(type + " 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);
```
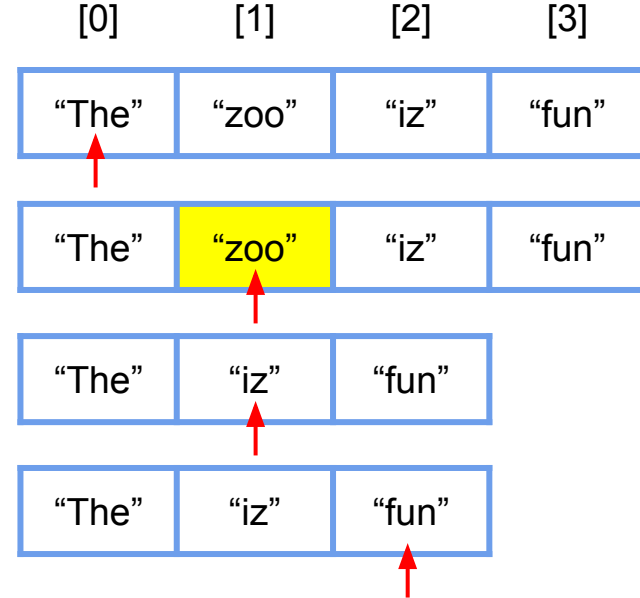
Why?

Remove all matches – NOT

- Position 0 doesn't match.

- Position 1 matches.

  So delete it.

- Position 2 doesn't match.
  (notice "iz" was skipped)

	[0]	[1]	[2]	[3]
	"The"	"zoo"	"iz"	"fun"
	"The"	"zoo"	"iz"	"fun"
	"The"	"iz"	"fun"	
	"The"	"iz"	"fun"	

Java's ConcurrentModificationException

- If you add or remove an element between iterations of a loop, Java will detect this and throw a ConcurrentModificationException.

```
Exception in thread "main" java.util.ConcurrentModificationException
    at java.util.LinkedList$ListItr.checkForComodification(LinkedList.java:966)
    at java.util.LinkedList$ListItr.next(LinkedList.java:888)
    at BadProgram.main(BadProgram.java:11)
```

Class containing the bug.

Method containing the bug.

Line number of the bug.

Remove all matches – two correct solutions

- **Solution #1**: Make a list of z words, then remove them all at once:
  ```
  LinkedList<String> zWords = zWords(list);
  list.removeAll(zWords);
  ```

- **Solution #2**: Use an iterator:
  ```
  for (Iterator<String> it = list.iterator(); it.hasNext();)
      if (it.next().contains("z"))
          it.remove();
  ```

The first solution is simpler but slower (loops over the list twice).
The second solution is more complex but more efficient (loops once).

Remove one match - two solutions

- **Solution #1**: Stop loop after removing to avoid an exception:

```
for (String word : list)
    if (word.contains("z")) {
        list.remove(word);
        break;
    }
```

- **Solution #2**: Use an iterator:

```
for (Iterator<String> it = list.iterator(); it.hasNext(); )
    if (it.next().contains("z")) {
        it.remove();
        break;
    }
```

Stop looping when item is removed.

Specification

A deck has 52 unique cards. Each card has a number between 1 to 13 and a suit which is one of "Clubs", "Diamonds", "Hearts" or "Spades".

The dealer can select the following menu options:

- shuffle the deck
- deal a card face up so that it shows a string in the format "7 of Spades"

Sample I/O

```
Choice (s/d/x): s
Choice (s/d/x): d
7 of Hearts
Choice (s/d/x): d
Ace of Spades
Choice (s/d/x): x
```

Design

Classes	Dealer	Deck	Card
Fields	deck	cards	number suit
Goals			
use	use()		
shuffle	shuffle()	shuffle()	
deal	deal()	removeCard()	toString()

Code: classes and fields

```java
import java.util.*;

public class Deck {
    private LinkedList<Card> cards = new LinkedList<Card>();
}

public class Card {
    private int number;
    private String suit;
}
```

Code: Constructors

```java
public class Dealer {
    public Dealer() {
        deck = new Deck();
    }
}
public class Deck {
    public Deck() {
        String[] suits = { "Clubs", "Diamonds", "Hearts", "Spades" };
        for (String suit : suits)
            for (int number = 1; number <= 13; number++)
                cards.add(new Card(number, suit));
    }
}
```

```java
public class Card {
    public Card(int number, String suit) {
        this.number = number;
        this.suit = suit;
    }
}
```

Goal: shuffle

```java
public class Dealer {
    public void shuffle() {
        deck.shuffle();
    }
}
public class Deck {
    public void shuffle() {
        Collections.shuffle(cards);
    }
}
```

Goal: deal

```
public class Dealer {
    public void deal() {
        System.out.println(deck.removeCard());
    }
}


public class Deck {
    public Card removeCard() {
        return cards.removeFirst();
    }
}
```

```
public class Card {
    @Override
    public String toString() {
        return number + " of " + suit;
    }
}
```

Specification

A stadium sells tickets for seats. There are 300 "front" seats at $400 each, 1500 "middle" seats at $100 each, and 200 "back" seats at $60 each. The user can sell seats, view seat availability and view income earned.

Sample I/O

```
Choice (s/v/i/x): s
Group: middle
Number of middle seats: 3
Choice (s/v/i/x): s
Group: back
Number of back seats: 1
Choice (s/v/i/x): v
300 front seats @ $400.00
1497 middle seats @ $100.00
199 back seats @ $60.00
Choice (s/v/i/x): i
The income is $360.00
Choice (s/v/i/x): x
```

DEMO