Advanced Object-oriented Programming

Lecture 11

Utility Classes
(Re)Using Classes

One of the chief motivations for the Object Paradigm is code re-use.

Code in a method is written once, and can be used many times (each time the method is called).

A class, such as a class of Linked Lists, can also be re-used (each time a programmer needs to store lists of things).

Many classes can be useful in many different applications (think of all the GUI component classes: Button, TextField, Frame, etc.)

These are often bundled together in packages so that programmers can benefit from the hard work that others have put in to implementing these classes.
One of the chief motivations for the Object Paradigm is code re-use.

Code in a method is written once, and can be used many times (each time the method is called).

A class, such as a class of Linked Lists, can also be re-used (each time a programmer needs to store lists of things).

Many classes can be useful in many different applications (think of all the GUI component classes: Button, TextField, Frame, etc.)

These are often bundled together in packages so that programmers can benefit from the hard work that others have put in to implementing these classes.
(Re)Using Classes

One of the chief motivations for the Object Paradigm is code re-use.

Code in a method is written once, and can be used many times (each time the method is called).

A class, such as a class of Linked Lists, can also be re-used (each time a programmer needs to store lists of things).

Many classes can be useful in many different applications (think of all the GUI component classes: Button, TextField, Frame, etc.)

These are often bundled together in packages so that programmers can benefit from the hard work that others have put in to implementing these classes.
Packages

- Package `java.awt` (Abstract Windowing Toolkit) provides many useful heavyweight* classes for building GUIs.
- Package `javax.swing` (it don’t mean a thing) similarly provides useful lightweight† classes for GUIs.
- Package `java.util` contains useful utility classes such as linked lists, queues, vectors, etc.

* Heavyweight means implemented using system calls to the native window manager
† Lightweight means implemented in Java (on top of the awt)
Package **java.awt** (Abstract Windowing Toolkit) provides many useful heavyweight* classes for building GUIs.

Package **javax.swing** (it don’t mean a thing) similarly provides useful lightweight† classes for GUIs.

Package **java.util** contains useful utility classes such as linked lists, queues, vectors, etc.

* Heavyweight means implemented using system calls to the native window manager
† Lightweight means implemented in Java (on top of the awt)
Packages

- Package `java.awt` (Abstract Windowing Toolkit) provides many useful heavyweight* classes for building GUIs.
- Package `javax.swing` (it don’t mean a thing) similarly provides useful lightweight† classes for GUIs.
- Package `java.util` contains useful utility classes such as linked lists, queues, vectors, etc.

* Heavyweight means implemented using system calls to the native window manager
† Lightweight means implemented in Java (on top of the awt)
Packages

- Package `java.awt` (Abstract Windowing Toolkit) provides many useful heavyweight∗ classes for building GUIs.
- Package `javax.swing` (it don’t mean a thing) similarly provides useful lightweight† classes for GUIs.
- Package `java.util` contains useful utility classes such as linked lists, queues, vectors, etc.

∗ Heavyweight means implemented using system calls to the native window manager
† Lightweight means implemented in Java (on top of the awt)
Packages

- Package `java.awt` (Abstract Windowing Toolkit) provides many useful heavyweight* classes for building GUIs.
- Package `javax.swing` (it don’t mean a thing) similarly provides useful lightweight† classes for GUIs.
- Package `java.util` contains useful utility classes such as linked lists, queues, vectors, etc.

* Heavyweight means implemented using system calls to the native window manager
† Lightweight means implemented in Java (on top of the awt)
A *Vector* is a growable list.
I.e., it can hold any number of objects
It has operations to add to the start of the list, access the \(i\)th element, place an element at a given position, and much more.

It’s implemented using an array (default length 10); if it runs out of space, it creates another, larger array (by default, double the size), and copies the old array into the new one.

It’s generally very fast for accessing at any point, adding/removing at the ends, but adding can occasionally be (a little bit) slower because of the array-copying. Removing elements from the middle can also be slow because of the need to shuffle elements down the array to fill the gap.
Example: Vectors

A *Vector* is a growable list. I.e., it can hold any number of objects. It has operations to add to the start of the list, access the $i$th element, place an element at a given position, and much more.

It’s implemented using an array (default length 10); if it runs out of space, it creates another, larger array (by default, double the size), and copies the old array into the new one.

It’s generally very fast for accessing at any point, adding/removing at the ends, but adding can occasionally be (a little bit) slower because of the array-copying. Removing elements from the middle can also be slow because of the need to shuffle elements down the array to fill the gap.
Example: Vectors

A Vector is a growable list. I.e., it can hold any number of objects. It has operations to add to the start of the list, access the $i$th element, place an element at a given position, and much more.

It’s implemented using an array (default length 10); if it runs out of space, it creates another, larger array (by default, double the size), and copies the old array into the new one.

It’s generally very fast for accessing at any point, adding/removing at the ends, but adding can occasionally be (a little bit) slower because of the array-copying. Removing elements from the middle can also be slow because of the need to shuffle elements down the array to fill the gap.
Vectors

from the API

```java
public class Vector<E>
...
```

The Vector class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.

So what’s the `<E>`?
Vectors

```java
public class Vector<E> {
...
}
```

The Vector class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.

So what's the `<E>`?
Class Vector belongs to the Java ‘Collections Framework’, a group of interfaces and data structures for storing collections of things.

Before Java 1.5, none of these were homogeneous: i.e., they could store any instance of any class.

For example, a Vector could contain a String, an Integer, a BandCard and a Button in its list.

**pre-1.5 Vector methods**

```java
public void addElement(Object o) { ... }

public Object elementAt(int i) { ... }
```

(In Java, all classes are subclasses of Object.)
Generics

Class Vector belongs to the Java ‘Collections Framework’, a group of interfaces and data structures for storing collections of things.

Before Java 1.5, none of these were *homogeneous*: i.e., they could store any instance of any class.

For example, a Vector could contain a String, an Integer, a BandCard and a Button in its list.

<table>
<thead>
<tr>
<th>pre-1.5 Vector methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void addElement(Object o) { ...}</td>
</tr>
<tr>
<td>public Object elementAt(int i) { ...}</td>
</tr>
</tbody>
</table>

(In Java, *all* classes are subclasses of Object.)
Generics

Class Vector belongs to the Java ‘Collections Framework’, a group of interfaces and data structures for storing collections of things.

Before Java 1.5, none of these were *homogeneous*: i.e., they could store any instance of any class.

For example, a Vector could contain a String, an Integer, a BandCard and a Button in its list.

pre-1.5 Vector methods

```java
public void addElement(Object o) { ... }

public Object elementAt(int i) { ... }
```

(In Java, *all* classes are subclasses of Object.)
Generics

Class Vector belongs to the Java ‘Collections Framework’, a group of interfaces and data structures for storing collections of things.

Before Java 1.5, none of these were *homogeneous*: i.e., they could store any instance of any class.

For example, a Vector could contain a String, an Integer, a BandCard and a Button in its list.

<table>
<thead>
<tr>
<th>pre-1.5 Vector methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void addElement(Object o) { ...}</td>
</tr>
<tr>
<td>public Object elementAt(int i) { ...}</td>
</tr>
</tbody>
</table>

(In Java, *all* classes are subclasses of Object.)
Generics

Class Vector belongs to the Java ‘Collections Framework’, a group of interfaces and data structures for storing collections of things.

Before Java 1.5, none of these were *homogeneous*: i.e., they could store any instance of any class.

For example, a Vector could contain a String, an Integer, a BandCard and a Button in its list.

```
pre-1.5 Vector methods

public void addElement(Object o) { ...}
public Object elementAt(int i) { ...}
```

(In Java, *all* classes are subclasses of Object.)
Class Vector belongs to the Java ‘Collections Framework’, a group of interfaces and data structures for storing collections of things.

Before Java 1.5, none of these were *homogeneous*: i.e., they could store any instance of any class.

For example, a Vector could contain a String, an Integer, a BandCard and a Button in its list.

**pre-1.5 Vector methods**

```java
public void addElement(Object o) { ... }
public Object elementAt(int i) { ... }
```

(In Java, all classes are subclasses of Object.)
A Non-homogeneous Vector

pre-1.5 code

```java
Vector stuff = new Vector();
stuff.add("hello");
stuff.add(new Integer(9));
stuff.add(new BandCard("The Clash", 6, 17, 9, 4, 12));
System.out.println("2nd element is: " + stuff.elementAt(1));
```
A Non-homogeneous Vector

Vector stuff = new Vector();
stuff.add("hello");
stuff.add(new Integer(9));
stuff.add(new BandCard("The Clash", 6,17,9,4,12));
System.out.println("2nd element is: " + stuff.elementAt(1));
A Non-homogeneous Vector

pre-1.5 code

Vector stuff = new Vector();
stuff.add("hello");
stuff.add(new Integer(9));
stuff.add(new BandCard("The Clash", 6,17,9,4,12));
System.out.println("2nd element is: " + stuff.elementAt(1));
A Non-homogeneous Vector

```java
Vector stuff = new Vector();
stuff.add("hello");
stuff.add(new Integer(9));
stuff.add(new BandCard("The Clash", 6,17,9,4,12));
System.out.println("2nd element is: "+ stuff.elementAt(1));
```
A Non-homogeneous Vector

Vector stuff = new Vector();
stuff.add("hello");
stuff.add(new Integer(9));
stuff.add(new BandCard("The Clash", 6,17,9,4,12));
System.out.println("2nd element is: "+ stuff.elementAt(1));
Generics

However, very often we want to work with homogeneous collections: lists of BandCards, for example.

If we know that everything in a Vector belongs to a particular class (i.e., the Vector is in fact homogeneous), we can use a cast to stop the compiler generating a type-error (we get a run-time error if we’re wrong):

```java
Vector hand = new Vector();
...
    // fill hand with BandCards

int len = v.size();
for (int i = 0 ; i < len ; i++) {
    System.out.println(
        ((BandCard)hand.elementAt(i)).getName());
}
```
Generics

However, very often we want to work with homogeneous collections: lists of BandCards, for example.

If we know that everything in a Vector belongs to a particular class (i.e., the Vector is in fact homogeneous), we can use a `cast` to stop the compiler generating a type-error (we get a run-time error if we’re wrong):

```java
Vector hand = new Vector();
...
// fill hand with BandCards
int len = v.size();
for (int i = 0 ; i < len ; i++) {
    System.out.println(
        ((BandCard)hand.elementAt(i)).getName());
}
```
Generics

However, very often we want to work with homogeneous collections: lists of BandCards, for example.

If we know that everything in a Vector belongs to a particular class (i.e., the Vector is in fact homogeneous), we can use a cast to stop the compiler generating a type-error (we get a run-time error if we’re wrong):

```java
Vector hand = new Vector();
...
// fill hand with BandCards
int len = v.size();
for (int i = 0 ; i < len ; i++) {
    System.out.println((BandCard)hand.elementAt(i)).getName();
}
```
Generics

However, very often we want to work with homogeneous collections: lists of BandCards, for example.

If we know that everything in a Vector belongs to a particular class (i.e., the Vector is in fact homogeneous), we can use a cast to stop the compiler generating a type-error (we get a run-time error if we’re wrong):

```java
Vector hand = new Vector();
...
// fill hand with BandCards
int len = v.size();
for (int i = 0 ; i < len ; i++) {
    System.out.println((BandCard)hand.elementAt(i)).getName());
}
```
Generics

However, very often we want to work with homogeneous collections: lists of BandCards, for example.

If we know that everything in a Vector belongs to a particular class (i.e., the Vector is in fact homogeneous), we can use a cast to stop the compiler generating a type-error (we get a run-time error if we’re wrong):

```java
Vector hand = new Vector();
...
  // fill hand with BandCards
int len = v.size();
for (int i = 0 ; i < len ; i++) {
  System.out.println((BandCard)hand.elementAt(i).getName());
}
```
Generics

This is not very nice code: if we wrote the code for storing a hand of cards ourselves (as we did in class CardHand), our methods such as add or remove explicitly state that they take a BandCard as parameter, or return a BandCard, and we have no need of casts in our code.

Because homogeneous collections are so common in programming, Java introduced ‘Generics’ in Java 1.5.

This allows programmers to declare that their collections are homogeneous (e.g., they are working with a Vector of Strings, or a Vector of BandCards, etc.)
This is not very nice code: if we wrote the code for storing a hand of cards ourselves (as we did in class `CardHand`), our methods such as `add` or `remove` explicitly state that they take a BandCard as parameter, or return a BandCard, and we have no need of casts in our code.

Because homogeneous collections are so common in programming, Java introduced ‘Generics’ in Java 1.5. This allows programmers to declare that their collections are homogeneous (e.g., they are working with a Vector of Strings, or a Vector of BandCards, etc.)
Generics

This is not very nice code: if we wrote the code for storing a hand of cards ourselves (as we did in class CardHand), our methods such as add or remove explicitly state that they take a BandCard as parameter, or return a BandCard, and we have no need of casts in our code.

Because homogeneous collections are so common in programming, Java introduced ‘Generics’ in Java 1.5.

This allows programmers to declare that their collections are homogeneous (e.g., they are working with a Vector of Strings, or a Vector of BandCards, etc.)
Generics: Type Parameters

The main mechanism for this is **Type Parameters**

Type parameters already exist in the array types. For example

- `Integer[]` — arrays of `Integers`
- `String[]` — arrays of `Strings`
- `TextField[]` — arrays of `TextFields`
- `BandCard[]` — arrays of `BandCards`
Generics: Type Parameters

The main mechanism for this is **Type Parameters**

Type parameters already exist in the array types. For example

- `Integer[]` — arrays of `Integers`
- `String[]` — arrays of `Strings`
- `TextField[]` — arrays of `TextFields`
- `BandCard[]` — arrays of `BandCards`
Generics: Type Parameters

The main mechanism for this is **Type Parameters**

Type parameters already exist in the array types. For example

- `Integer[]` — arrays of `Integers`
- `String[]` — arrays of `Strings`
- `TextField[]` — arrays of `TextFields`
- `BandCard[]` — arrays of `BandCards`
Generics: Type Parameters

The main mechanism for this is **Type Parameters**.

Type parameters already exist in the array types. For example:

- `Integer[]` — arrays of `Integers`
- `String[]` — arrays of `Strings`
- `TextField[]` — arrays of `TextFields`
- `BandCard[]` — arrays of `BandCards`
Generics: Type Parameters

The new (from Java 1.5) notation for type parameters uses angle brackets. For example

- `Vector<Integer>` — Vectors of Integers
- `Vector<String>` — Vectors of Strings
- `Vector<TextField>` — Vectors of TextFields
- `Vector<BandCard>` — Vectors of BandCards

Similarly, `java.util.LinkedList<BandCard>`, etc.
The new (from Java 1.5) notation for type parameters uses angle brackets. For example

- Vector<Integer> — Vectors of Integers
- Vector<String> — Vectors of Strings
- Vector<TextField> — Vectors of TextFields
- Vector<BandCard> — Vectors of BandCards

Similarly java.util.LinkedList<BandCard>, etc.
Generics: Type Parameters

The new (from Java 1.5) notation for type parameters uses angle brackets. For example

- Vector<Integer> — Vectors of Integers
- Vector<String> — Vectors of Strings
- Vector<TextField> — Vectors of TextFields
- Vector<BandCard> — Vectors of BandCards

Similarly java.util.LinkedList<BandCard>, etc.
Generics: Type Parameters

The new (from Java 1.5) notation for type parameters uses angle brackets. For example

- Vector<Integer> — Vectors of Integers
- Vector<String> — Vectors of Strings
- Vector<TextField> — Vectors of TextFields
- Vector<BandCard> — Vectors of BandCards

Similarly java.util.LinkedList<BandCard>, etc.
Generics: Type Parameters

The new (from Java 1.5) notation for type parameters uses angle brackets. For example

- `Vector<Integer>` — Vectors of Integers
- `Vector<String>` — Vectors of Strings
- `Vector<TextField>` — Vectors of TextFields
- `Vector<BandCard>` — Vectors of BandCards

Similarly `java.util.LinkedList<BandCard>`, etc.
Generics: Type Parameters

```java
Vector<BandCard> hand = 
    new Vector<BandCard>();
hand.add(new BandCard("ELP",8,3,1,6,0));
hand.add(new BandCard("Yes",5,2,0,3,-1));
System.out.print(hand.elementAt(0).getName());
// compiler type error
hand.add(new Integer(6)); // not a BandCard
```
Generics: Type Parameters

post-1.5 code

Vector<BandCard> hand =
    new Vector<BandCard>();
hand.add(new BandCard("ELP", 8, 3, 1, 6, 0));
hand.add(new BandCard("Yes", 5, 2, 0, 3, -1));
System.out.print(hand.elementAt(0).getName());
// compiler type error
hand.add(new Integer(6)); // not a BandCard
Generics: Type Parameters

```java
Vector<BandCard> hand =
    new Vector<BandCard>();
hand.add(new BandCard("ELP", 8, 3, 1, 6, 0));
hand.add(new BandCard("Yes", 5, 2, 0, 3, -1));
System.out.println(hand.elementAt(0).getName()); // compiler type error
hand.add(new Integer(6)); // not a BandCard
```
Generics: Type Parameters

Vector<BandCard> hand =
    new Vector<BandCard>();

hand.add(new BandCard("ELP", 8, 3, 1, 6, 0));
hand.add(new BandCard("Yes", 5, 2, 0, 3, -1));
System.out.print(hand.elementAt(0).getName());
// compiler type error
hand.add(new Integer(6)); // not a BandCard
Generics: Type Parameters

```
Vector<BandCard> hand =
    new Vector<BandCard>();
hand.add(new BandCard("ELP",8,3,1,6,0));
hand.add(new BandCard("Yes",5,2,0,3,-1));
System.out.print(hand.elementAt(0).getName());
// compiler type error
hand.add(new Integer(6)); // not a BandCard
```
Generics: Type Parameters

```
Vector<BandCard> hand =
    new Vector<BandCard>();
hand.add(new BandCard("ELP",8,3,1,6,0));
hand.add(new BandCard("Yes",5,2,0,3,-1));
System.out.print(hand.elementAt(0).getName());
// compiler type error
hand.add(new Integer(6)); // not a BandCard
```
Generics: Type Parameters

post-1.5 code

Vector<BandCard> hand =
    new Vector<BandCard>();
hand.add(new BandCard("ELP", 8, 3, 1, 6, 0));
hand.add(new BandCard("Yes", 5, 2, 0, 3, -1));
System.out.print(hand.elementAt(0).getName());
// compiler type error
hand.add(new Integer(6));  // not a BandCard
Generics: Type Parameters

post-1.5 code

```java
Vector<BandCard> hand =
    new Vector<BandCard>();
hand.add(new BandCard("ELP", 8, 3, 1, 6, 0));
hand.add(new BandCard("Yes", 5, 2, 0, 3, -1));
System.out.print(hand.elementAt(0).getName());
// compiler type error
hand.add(new Integer(6)); // not a BandCard
```
The $E$ in $\text{Vector}\langle E \rangle$ is a \textit{formal} \textit{Type Parameter}

i.e., it stands as a ‘placeholder’ for whatever class is given as an \textit{actual} \textit{Type Parameter}.

Formal and actual parameters are already familiar from methods …
E’s a Type Parameter

The E in `Vector<E>` is a formal Type Parameter
i.e., it stands as a ‘placeholder’ for whatever class is given as an actual Type Parameter.

Formal and actual parameters are already familiar from methods . . .
formal/actual method parameters

**Formal** parameters are the ‘placeholders’ declared in method signatures (in the parameter list);

**Actual** parameters are the values used in method *calls* (‘invocations’).

```java
public void push(int v) {
    v = ;
    values[++pointer] = ;
}
```

```java
Stack st =
    new Stack();
int i = 2;
st.push(12);
st.push();
```

This is how parameter-passing is implemented.
formal/actual method parameters

**Formal** parameters are the ‘placeholders’ declared in method signatures (in the parameter list);

**Actual** parameters are the values used in method *calls* (‘invocations’).

```java
in class Stack

public void push(int v) {
    v = ;
    values[++pointer] = v;
}
```

some program:

```java
Stack st =
    new Stack();
int i = 2;
st.push(12);
st.push();
```

This is how parameter-passing is implemented
Formal parameters are the ‘placeholders’ declared in method signatures (in the parameter list); Actual parameters are the values used in method calls (‘invocations’).

In class Stack

```java
public void push(int v) {
    v = ;
    values[++pointer] = v;
}
```

Some program

```java
Stack st =
    new Stack();
int i = 2;
st.push(12);
st.push();
```

This is how parameter-passing is implemented
formal/actual method parameters

**Formal** parameters are the ‘placeholders’ declared in method signatures (in the parameter list);

**Actual** parameters are the values used in method *calls* (‘invocations’).

---

### in class Stack

```java
public void push(int v) {
    v = ;
    values[++pointer] = v;
}
```

### some program

```java
Stack st =
    new Stack();
int i = 2;
st.push(12);
st.push();
```

This is how parameter-passing is implemented
formal/actual method parameters

**Formal** parameters are the ‘placeholders’ declared in method signatures (in the parameter list);

**Actual** parameters are the values used in method *calls* ('invocations').

In class Stack:

```java
public void push(int v) {
    v = 12;
    values[++pointer] = v;
}
```

Some program:

```java
Stack st =
    new Stack();
int i = 2;
st.push(12);
st.push();
```

This is how parameter-passing is implemented.
formal/actual method parameters

Formal parameters are the ‘placeholders’ declared in method signatures (in the parameter list);

Actual parameters are the values used in method calls (‘invocations’).

in class Stack

```java
public void push(int v) {
    v = 12;
    values[++pointer] = 12;
}
```

some program

```java
Stack st = new Stack();
int i = 2;
st.push(12);
st.push();
```

This is how parameter-passing is implemented
formal/actual method parameters

Formal parameters are the ‘placeholders’ declared in method signatures (in the parameter list);

Actual parameters are the values used in method calls (‘invocations’).

in class Stack

```java
public void push(int v) {
    v = ;
    values[++pointer] = v;
}
```

some program

```java
Stack st =
    new Stack();
int i = 2;
st.push(12);
st.push(24 + i);
```

This is how parameter-passing is implemented
Formal parameters are the ‘placeholders’ declared in method signatures (in the parameter list);

Actual parameters are the values used in method calls (‘invocations’).

```
in class Stack
public void push(int v) {
    v = ;
    values[++pointer] = v;
}
```

```
some program
Stack st =
    new Stack();
int i = 2;
st.push(12);
st.push(24 + i);
```

This is how parameter-passing is implemented
formal/actual method parameters

**Formal** parameters are the ‘placeholders’ declared in method signatures (in the parameter list);

**Actual** parameters are the values used in method calls (‘invocations’).

---

**in class Stack**

```java
public void push(int v) {
    v = 26;
    values[++pointer] = v;
}
```

**some program**

```java
Stack st =
    new Stack();
int i = 2;
st.push(12);
st.push(26);
```

This is how parameter-passing is implemented
formal/actual method parameters

**Formal** parameters are the ‘placeholders’ declared in method signatures (in the parameter list);

**Actual** parameters are the values used in method *calls* (‘invocations’).

```
in class Stack
public void push(int v) {
    v = 26;
    values[++pointer] = 26;
}
```

```
some program
Stack st =
    new Stack();
int i = 2;
st.push(12);
st.push(26);
```

This is how parameter-passing is implemented.
In methods, **formal** parameters always have a **type**

```java
public void push(int v) { ...}
public void add(BandCard b) { ...}
public Node(BandCard b, Node t) { ...}
```

Each **formal** parameter is a variable of its declared **type** …

… anything you can do with **actual** values of that type, you can do with the **formal** parameter
formal parameters

In methods, **formal** parameters always have a **type**

<table>
<thead>
<tr>
<th>Some methods we’ve already seen</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void push(int v) { ...}</td>
</tr>
<tr>
<td>public void add(BandCard b) { ...}</td>
</tr>
<tr>
<td>public Node(BandCard b, Node t) { ...}</td>
</tr>
</tbody>
</table>

Each **formal** parameter is a variable of its declared **type** ...  

... anything you can do with **actual** values of that type, you can do with the **formal** parameter
formal parameters

In methods, formal parameters always have a type

<table>
<thead>
<tr>
<th>Some methods we’ve already seen</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void push(int v) { ...}</td>
</tr>
<tr>
<td>public void add(BandCard b) { ...}</td>
</tr>
<tr>
<td>public Node(BandCard b, Node t) { ...}</td>
</tr>
</tbody>
</table>

Each formal parameter is a variable of its declared type …

… anything you can do with actual values of that type, you can do with the formal parameter
formal parameters

In methods, formal parameters always have a type

<table>
<thead>
<tr>
<th>some methods we’ve already seen</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void push(int v) { ...}</td>
</tr>
<tr>
<td>public void add(BandCard b) { ...}</td>
</tr>
<tr>
<td>public Node(BandCard b, Node t) { ...}</td>
</tr>
</tbody>
</table>

Each formal parameter is a variable of its declared type . . .

. . . anything you can do with actual values of that type, you can do with the formal parameter
formal parameter types

For example, here’s a not very useful method using a `Point`:

```java
public static void pointless(Point p) {
    int x = p.getX(); // x == 2
    p.move(0, x)
}
```

in `Point.main(String[])`

```java
Point myPoint = new Point(2,4);
Point.pointless(myPoint);
System.out.print("yCoord of myPoint is: "+ myPoint.yCoord);
```

We’d expect to see "yCoord of p is: 6"
formal parameter types

For example, here’s a not very useful method using a `Point`:

```java
public static void pointless(Point p) {
    int x = p.getX();  // x == 2
    p.move(0, x)
}
```

in `Point.main(String[])`

```java
Point myPoint = new Point(2, 4);
Point.pointless(myPoint);
System.out.print("yCoord of myPoint is: " +
    myPoint.yCoord);
```

We’d expect to see "yCoord of p is: 6"
formal parameter types

For example, here’s a not very useful method using a `Point`:

```java
public static void pointless(Point p) {
    int x = p.getX();  // x == 2
    p.move(0, x)
}
```

in `Point.main(String[])`

```java
Point myPoint = new Point(2,4);
Point.pointless(myPoint);
System.out.print("yCoord of myPoint is: "+
    myPoint.yCoord);
```

We’d expect to see "yCoord of p is: 6"
formal parameter types

For example, here’s a not very useful method using a Point:

```java
public static void pointless(Point p) {
    int x = p.getX(); // x == 2
    p.move(0, x)
}
```

In Point.main(String[]):

```java
Point myPoint = new Point(2,4);
Point.pointless(myPoint);
System.out.print("yCoord of myPoint is: "+myPoint.yCoord);
```

We’d expect to see "yCoord of p is: 6"
formal parameter types

For example, here’s a not very useful method using a `Point`:

```java
public static void pointless(Point p) {
    int x = p.getX();  // x == 2
    p.move(0, x)
}
```

in `Point.main(String[])`

```java
Point myPoint = new Point(2, 4);
Point.pointless(myPoint);
System.out.print("yCoord of myPoint is: "+
                myPoint.yCoord);
```

We’d expect to see "yCoord of p is: 6"
formal parameter types

For example, here’s a not very useful method using a `Point`:

```java
public static void pointless(Point p) {
    int x = p.getX(); // x == 2
    p.move(0,x)
}
```

Add this to class `Point`.

```java
in Point.main(String[])()
Point myPoint = new Point(2,4);
Point.pointless(myPoint);
System.out.print("yCoord of myPoint is: " +
myPoint.yCoord);
```

We’d expect to see "yCoord of p is: 6"
formal parameter types

For example, here’s a not very useful method using a Point:

```java
public static void pointless(Point p) {
    int x = p.getX(); // x == 2
    p.move(0,x)
}
```

```java
Point myPoint = new Point(2,4);
Point.pointless(myPoint);
System.out.print("yCoord of myPoint is: " + myPoint.yCoord);
```

We’d expect to see "yCoord of p is: 6"
formal parameter types

For example, here’s a not very useful method using a Point:

```java
public static void pointless(Point p) {
    int x = p.getX(); // x == 2
    p.move(0, x)
}
```

```
in Point.main(String[])
Point myPoint = new Point(2, 4);
Point.pointless(myPoint);
System.out.print("yCoord of myPoint is: " + myPoint.yCoord);
```

We’d expect to see "yCoord of p is: 6"
formal parameter types

For example, here’s a not very useful method using a `Point`:

```
public static void pointless(Point p) {
    int x = p.getX();    // x == 2
    p.move(0, x)
}
```

```
in Point.main(String[])}
Point myPoint = new Point(2,4);
Point.pointless(myPoint);
System.out.print("yCoord of myPoint is: " +
    myPoint.yCoord);
```

We’d expect to see "yCoord of p is: 6"
formal parameter types

For example, here’s a not very useful method using a `Point`:

```
public static void pointless(Point p) {
    int x = p.getX();  // x == 2
    p.move(0, x)
}
```

in `Point.main(String[])`

```
Point myPoint = new Point(2, 4);
Point.pointless(myPoint);
System.out.print("yCoord of myPoint is: " +
               myPoint.yCoord);
```

We’d expect to see "yCoord of p is: 6"
formal/actual type parameters

E is a formal *type* parameter when the class is *declared*

```java
public class Vector<E> {
    ...
    public void addElement(E o) { ... }
    public E elementAt(int i) { ... }
    ...
}
```
formal/actual type parameters

$E$ is a formal type parameter when the class is declared

```java
public class Vector<E> {
 ...
    public void addElement(E o) { ...}
    public E elementAt(int i) { ...}
 ...
}
```
formal/actual type parameters

When given an actual parameter, e.g., `Vector<BandCard>`, the actual parameter replaces the formal parameter, so the class looks like this:

```java
public class Vector<BandCard> {
    ...
    public void addElement(BandCard o) { ... }
    public BandCard elementAt(int i) { ... }
    ...
}
```

NB - this replacement continues even within the method bodies.
formal/actual type parameters

When given an actual parameter, e.g., \texttt{Vector<BandCard>}, the actual parameter replaces the formal parameter, so the class looks like this:

```java
public class Vector<BandCard> {
    ...
    public void addElement(BandCard o) { ... }
    public BandCard elementAt(int i) { ... }
    ...
}
```

NB - this replacement continues even within the method bodies.
public class Vector<E> {
    public Vector() {
        E[] elementData = ...;
    }
    public E elementAt(int i) {
        E theElement = elementData[i];
        ...
    }
    ...
}
public class Vector<BandCard> {

    public Vector() {
        BandCard[] elementData = ...;
    }

    public BandCard elementAt(int i) {
        BandCard theElement = ...;
        ...
    }

    ...
}
The constructor does not take a type parameter when it is declared:

```java
public Vector() {
    ...
}
```

The constructor *does* take an *actual* type parameter when it is called:

```java
Vector<BandCard> hand = new Vector<BandCard>();
```
Notes

- The constructor does not take a type parameter when it is declared:

```java
public Vector() {
    ...
}
```

- The constructor does take an actual type parameter when it is called:

```java
using class Vector

Vector<BandCard> hand =
    new Vector<BandCard>();
```
Our implementations of linked lists and doubly-linked lists could have been written more generally:

```java
public class LList<A> {
    private BiNode listStart;
    public void addend(A a) {
        ...new BiNode(a) ...
    }
    ...
    private class BiNode {
        private A value;
        private BiNode tail;
        ...
    }
}
```
Our implementations of linked lists and doubly-linked lists could have been written more generally:

```java
public class LList<A> {
    private BiNode<A> listStart;
    public void addend(A a) {
        ...new BiNode<A>() ...
    }
    ...
    private class BiNode<B> {
        private B value;
        private BiNode<B> tail;
        ...
    }
}
```
Scope of Classes

**public** — visible inside and outside its package
(i.e., it can be imported into another package)

**default** — visible only inside its own package
(i.e., it *cannot* be imported into another package)

**private** — if a class is declared private within another class,
it is visible only within that class

Thus class **BiNode** is visible only within class **LLList**
it is a ‘helper’ class that is used only to implement linked lists.
That’s All, Folks!

Summary

- class Vector
- type parameters
- formal/actual parameters

Next:

documentation