CS61BL

Lecture 2:
Inheritance
Interfaces
Iterators
Linked Lists
FAQ

• Q: Why isn’t Eclipse working for me?
• A: Come to office hours!
FAQ

• Q: Do we have lab on Friday?
• A: No lab or office hours on Friday. Enjoy your academic holiday!
FAQ

• Q: When will project 1 be released?
• A: Soon! Before 8pm tonight.
FAQ

• Q: What happens when you try to do \textit{blank} in Java?
• A: Try it out on your own!
Inheritance

• “Avoid reinventing the wheel”
  – Reuse code
• “B is a type of A”
  – B is a subclass of A
• All objects extend from Object class!

class B extends A {
    // rest of class body
}

Inheriting Methods

```java
public class Coin {
    public boolean flip() {
        return (Math.random() < 0.5);
    }
    // rest of class
}

public class Quarter extends Coin {
    // rest of class
}
```
Inheriting Methods

Quarter q = new Quarter();
boolean result = q.flip();
Methods inherited from Object

- boolean equals(Object o)
- String toString()
- int hashCode()
- Object clone()
- And more!
public class Coin {
    public boolean flip() {
        return (Math.random() < 0.5);
    }
    // rest of class
}
// rest of class

public class Quarter extends Coin {
    public boolean flip() {
        System.out.println("~Ping!~");
        return (Math.random() < 0.5);
    }
    // rest of class
}
public class Coin {
    public boolean flip() {
        return (Math.random() < 0.5);
    }
    // rest of class
}

public class Quarter extends Coin {
    public boolean flip() {
        System.out.println("~Ping!~");
        return super.flip();
    }
    // rest of class
}
Inheriting instance variables

```java
public class Coin {
    private double p;
    public Coin(double probHeads) {
        this.p = probHeads;
    }
    // rest of class
}
// rest of class

public class Quarter extends Coin {
    // rest of class
}
```
Inheriting instance variables

```java
public class Coin {
    protected double p;
    public Coin(double probHeads) {
        this.p = probHeads;
    }
    // rest of class
}
// rest of class

public class Quarter extends Coin {
    // rest of class
}
```
Inheriting instance variables

class Coin {
    protected double p;
    public Coin(double probHeads) {
        this.p = probHeads;
    }
    // rest of class
}

public class Quarter extends Coin {
    private double p;
    // rest of class
}

???
public class Coin {
    protected double p;
    public Coin(double probHeads) {
        this.p = probHeads;
    }
    // rest of class
}

public class Quarter extends Coin {
    private double p;
    private double value = 0.25;
    // rest of class
}
Inheriting constructors

public class Coin {
    protected double p;
    public Coin(double probHeads) {
        this.p = probHeads;
    }
    // rest of class
}

public class Quarter extends Coin {
    public Quarter (double probHeads) {
        super(probHeads);
    }
    // rest of class
}
Inheriting constructors

```java
public class Coin {
    protected double p;
    public Coin(double probHeads) {
        this.p = probHeads;
    }
    // rest of class
}
// rest of class

public class Quarter extends Coin {
    public Quarter (double probHeads) {
        // this line is incorrect
    }
    // rest of class
}
```
Inheriting constructors

```java
public class Coin {
    protected double p;
    public Coin() {
        this.p = 0.5;
    }
    // rest of class
}
// rest of class
```

```java
public class Quarter extends Coin {
    public Quarter (double probHeads) {
    }
    // rest of class
}
```
Static vs. Dynamic Type

Quarter q = new Quarter();

• Static type:
  – What compiler sees (compile time type binding)
  – Variable type

• Dynamic type:
  – Runtime type binding
  – Object type
Polymorphism Puzzles: #1

```java
Coin q = new Quarter();
```

- **Static type**
- **Dynamic type**

• Polymorphism
  - “Many forms”
  - Oracle’s Java documentation: “Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.”
Polymorphism Puzzles: #1

Coin q = new Quarter();

Static type ➔ Dynamic type

Will it compile?

- Yes! Coin is a superclass of Quarter. Variables can hold references to subclass objects. No problems here!
Polymorphism Puzzles: #2

Quarter q = new Coin();

Static type           Dynamic type

• Will it compile?
  – No! Quarter is a subclass of Coin. Not all Coins are Quarters. Can’t guarantee that a Coin object will be able to run Quarter methods at runtime.
Polymorphism Puzzles: #3

Coin c = new Quarter();
Quarter q = c;

• Will it compile?
  – No! c’s static type is Coin, so as far as the compiler knows, we’re trying to have the Quarter variable q hold a reference to a Coin object.
Polymorphism Puzzles: #4

Coin c = new Quarter();
Quarter q = (Quarter) c;

• Will it compile?
  – Yes! By casting, we’re telling the compiler “Trust me, I know what I’m doing.” We’re forcing the compiler to accept that c is actually a Quarter object.
Polymorphism Puzzles: #5

Coin c = new Coin();
Quarter q = (Quarter) c;

• Will it compile?
  – Yes, but...
Polymorphism Puzzles: #5

```java
Coin c = new Coin();
Quarter q = (Quarter) c;
```

• Will it run?
  – Nope. At runtime, Java realizes that you messed up with your casting. `c` isn’t actually a `Quarter` object. You lied to the compiler.
Polymorphism Puzzles: #6

```java
Coin c = new Quarter();
c.value;
```

• (Assume `value` is accessible)
• Will it compile?
  – No. The compiler thinks `c` is a `Coin`, and `Coin` objects don’t have a `value` instance variable.
Polymorphism Puzzles: #7

```java
Coin c = new Quarter();
c.flip();
```

• Will it compile?
  – Yes. Coin objects have a `flip` method, so this works fine.

• Whose `flip` method gets called?
  – At runtime, the method that belongs to the dynamic type of the object gets called.
instanceof

• Lets you know whether an object’s dynamic type is an instance of some class:

```java
Coin c = new Coin();
boolean b;
b = c instanceof Coin; // true
b = c instanceof Quarter; // false
```
instanceof

• Lets you know whether an object’s *dynamic type* is an instance of some class:

```java
Coin c = new Quarter();
Quarter q = new Quarter();
boolean b;
b = c instanceof Quarter; // true
b = q instanceof Coin; // true
```
FAQ

• Q: Can a class inherit from multiple parent classes?
  • A: No. (Possible in other languages: e.g. Python and C++)

• Q: Can multiple subclasses inherit from the same parent class?
  • A: Yes
FAQ

• Q: Why can’t the compiler just keep track of what an object’s dynamic type is?

• A:

Coin c;
if(Math.random() < 0.5) {
    c = new Coin();
} else {
    c = new Quarter();
}
ArrayList

• import java.util.ArrayList;

• Resizable arrays!

• Some useful methods:
  – add: add element to the middle of the ArrayList
  – get: returns an element from the ArrayList
  – remove: removes the element at a given index within the ArrayList
  – And more!
ArrayList: Usage Example

ArrayList a = new ArrayList();
a.add("Hello!");
a.add(0, new ArrayList());
a.remove(0);
String msg = a.get(0);
ArrayList: Usage Example

ArrayList a = new ArrayList();
a.add("Hello!");
a.add(0, new ArrayList());
a.remove(0);
String msg = a.get(0);
String msg = (String) a.get(0);
Class `ArrayList<E>`

Extends `AbstractList<E>`

Implements `List<E>`, `RandomAccess`, `Cloneable`, `Serializable`

Resizable-array implementation of the `List` interface. Implements all optional list operations.
Generics

• Oracle’s Java documentation: “Allows a type or method to operate on objects of various types while providing compile-time type safety”
• Basically, allows us to create ArrayLists of a specific type of Object:

```java
ArrayList<String> a = new ArrayList<String>();
```

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>add(E e)</td>
</tr>
<tr>
<td></td>
<td>Appends the specified element to the end of this list.</td>
</tr>
</tbody>
</table>
Interfaces

- A way to promise that a class has certain methods
- Abstract: can’t make an instance of an interface
- Abstract methods: no implemented method bodies
Interfaces: Syntax

interface InterfaceName {
    void methodName();

    int methodName2(int inputInt);
}

Interfaces: Syntax

```java
public class ClassName implements InterfaceName {
    public void methodName() {
        // implementation
    }

    public int methodName2(int inputInt) {
        // implementation
    }

    // rest of class
}
```
java.util

Class ArrayList<E>

java.lang.Object
   java.util.AbstractCollection<E>
      java.util.AbstractList<E>
         java.util.ArrayList<E>

All Implemented Interfaces:
   Serializable, Cloneable, Iterable<E>, Collection<E> List<E>, RandomAccess

Direct Known Subclasses:
   AttributeList, RoleList, RoleUnresolvedList

public class ArrayList<E>
   extends AbstractList<E>
   implements List<E>, RandomAccess, Cloneable, Serializable

Resizable-array implementation of the List interface. Implements all optional list operations.
Interface Example: List<T>

List<String> a = new ArrayList<String>();

✓

ArrayList<String> a = new List<String>();

✗
Interface Example: Comparable\textless T\textgreater 

• Only required method:
  \[
  \text{int} \ \text{compareTo}(T \ o)
  \]

• String, Integer, Date all implement Comparable

• Usually have T class implement Comparable\textless T\textgreater 

• Useful for sorting
Interface Example: Iterable\<T>\n
- Only required method:

  
  ```java
  Iterator\<T> iterator();
  ```
Interface Example: Iterable<T>

• An iterator

  iterates over items in a collection
  – i.e. it returns them one at a time
  – Order is not guaranteed
  – Should not modify contents of what you’re iterating over while using an iterator
Interface Example: `Iterable<T>`

- `boolean hasNext()`
  - Returns whether the iterator has items left to return
  - Not guaranteed to be called before calls to `next()`
  - Should not change state!!!

- `T next()`
  - Returns the next item
  - Behavior undefined if `hasNext()` returns false

- `void remove()`
  - Optional (throw `UnsupportedOperationException` if not implemented)
  - Removes item most recently returned with `next()`
Using an Iterator

ArrayList<String> a;
a = new ArrayList<String>();
a.add("Hello!");
a.add("Greetings!");
a.add("Well met!");
Iterator<String> myIter = a.iterator();
String msg = myIter.next();
Using an Iterator

ArrayList<String> a;
a = new ArrayList<String>();
// add Strings to ArrayList
Iterator<String> myIter = a.iterator();
while(myIter.hasNext()) {
    String msg = myIter.next();
    // do something with msg
}
Using an Iterator

```java
ArrayList<String> a;
a = new ArrayList<String>();
// add Strings to ArrayList
Iterator<String> myIter = a.iterator();
while (myIter.hasNext()) {
    String msg = myIter.next();
    // do something with msg
}
```
Interface Example: Iterable<T>

• Only required method:

```java
Iterator<T> iterator();
```
Class ArrayList<E>

java.lang.Object
   java.util.AbstractCollection<E>
      java.util.AbstractList<E>
         java.util.ArrayList<E>

All Implemented Interfaces:
   Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:
   AttributeList, RoleList, RoleUnresolvedList

public class ArrayList<E>
   extends AbstractList<E>
   implements List<E>, RandomAccess, Cloneable, Serializable

Resizable-array implementation of the List interface. Implements all optional list operations.
Special Iterable syntax

- ArrayList implements Iterable!
- Regular arrays are Iterable too!
- E.g. `int[] arrayOfInts = new int[3];`
- Which means we can do this:

```java
for (String msg : a) {
    // do something with msg
}
```
Special **Iterable syntax**

• Instead of this:

```java
Iterator<String> myIter = a.iterator();
while (myIter.hasNext()) {
    String msg = myIter.next();
    // do something with msg
}
```
FAQ

• Q: Can a class implement multiple interfaces?
  • A: Yes!

class ClassName implements InterfaceName1, InterfaceName2, InterfaceName3 {

• Q: Can an interface implement another interface?
  • A: Sort of!

interface InterfaceName1 extends InterfaceName2, InterfaceName3 {

Midterm 1 Information

- Wednesday July 9, 3-5pm
- 2050 VLSB and 100 GPB
- Review Session
  - Sunday July 6, 1-3pm
  - 100 GPB
Midterm 1 Information

• All lab and quiz material up to & including next Monday’s (Generics and Iterators)
• Lectures up until now
• Next Tuesday’s lab: Optional Git lab
  – Quiz
  – Time to review / study
• Next Thursday’s lab: Linked Lists! (part 1)
  – No quiz
Board Game & Dessert Potluck

• Wednesday July 16, 7-11pm
• 430/438 Soda (Wozniak Lounge)
• Bring your favorite board/card game. Bring a dessert/drink to share, potluck-style.
• Open to all CS61BL staff & students for us to get to know each other better
• Have fun!
Linked Lists: Terminology

• Node:

class LLNode {
    Object myItem;
    LLNode myNext;
    // constructors
    // & methods
}
Linked Lists: Terminology

• Linked List:

class LinkedList {
    LLNode myHead;
    // constructors
    // & methods
}

Linked List

myHead

LLNode

Object myItem
LLNode myNext

LLNode

Object myItem
LLNode myNext

LLNode

Object myItem
LLNode myNext
Linked Lists: Terminology

• Constructive
  – Make a copy of the input linked list, apply changes, and return new linked list

• Destructive
  – Modify input linked list directly
Example: Insert Node (destructive)

// Assume we have reference to some node
// in the linked list: LLNode currentNode
LLNode toAdd = new LLNode(5);
Example: Insert Node (destructive)

// Assume we have reference to some node
// in the linked list: LLNode currentNode
LLNode toAdd = new LLNode(5);
currentNode.myNext = toAdd;
Example: Insert Node (destructive)

// Assume we have reference to some node
// in the linked list: LLNode currentNode
LLNode toAdd = new LLNode(5);
currentNode.myNext = toAdd;
Example: Insert Node (destructive)

// Assume we have reference to some node
// in the linked list: LLNode currentNode
LLNode toAdd = new LLNode(5);
currentNode.myNext = toAdd;
Example: Insert Node (destructive)

// Assume we have reference to some node
// in the linked list: LLNode currentNode
LLNode toAdd = new LLNode(5);
Example: Insert Node (destructive)

// Assume we have reference to some node
// in the linked list: LLNode currentNode
LLNode toAdd = new LLNode(5);
toAdd.myNext = currentNode.myNext;
Example: Insert Node (destructive)

// Assume we have reference to some node
// in the linked list: LLNode currentNode
LLNode toAdd = new LLNode(5);
toAdd.myNext = currentNode.myNext;
Example: Insert Node (destructive)

// Assume we have reference to some node in the linked list: LLNode currentNode

LLNode toAdd = new LLNode(5);
toAdd.myNext = currentNode.myNext;
currentNode.myNext = toAdd;

```java
LLNode
int myItem
5
LLNode myNext

LLNode toAdd

LLNode
int myItem
4
LLNode myNext

LLNode
int myItem
6
LLNode myNext
```
Example: Insert Node (destructive)

```java
// Assume we have reference to some node
// in the linked list: LLNode currentNode
LLNode toAdd = new LLNode(5);
toAdd.myNext = currentNode.myNext;
currentNode.myNext = toAdd;
```
Example: Insert Node (destructive)

// Assume we have reference to some node in the linked list: LLNode currentNode
LLNode toAdd = new LLNode(5);
toAdd.myNext = currentNode.myNext;
currentNode.myNext = toAdd;

```
LLNode
int myItem
5

LLNode myNext

LLNode toAdd
int myItem
LLNode myNext

LLNode currentNode
int myItem
4

LLNode myNext

LLNode
int myItem
6

LLNode myNext
```
Variant: Doubly-linked

- Node:

  ```
  class LLNode {
    Object myItem;
    LLNode myNext;
    LLNode myPrev;
    // constructors
    // & methods
  }
  ```
Advantages of doubly-linked

• Faster to traverse in reverse, esp. if you have reference to last node
  – How much faster?
  – Next lecture: asymptotic analysis
Variant: Circularly-linked
Uses for circularly-linked

• Iterating over something over and over again
  – E.g. which player’s turn in a board game
  – Can add/remove/modify nodes in middle

• Used in some process scheduling algorithms
Other additions?

• Keep reference to last node
• Keep count of number of nodes
• Java’s LinkedList class implements List<T>
Problem: Linked List of length 0

• How to represent?
• Have myHead node be null?
  – Null checks are messy
• Lab’s solution
  – Always have an empty “dummy” node that represents the end of linked list
Good luck next week.