CSE 143
Lecture 20

Abstract Classes
Equals

slides created by Alyssa Harding
http://www.cs.washington.edu/143/

`ArrayIntList` review

- Recall our **`ArrayIntList`** class

```
// adds given value to end of list
void add(E value)

// adds given value at the given index
void add(E value, int index)

// returns size of list
int size()

// returns true if the list is empty,
// otherwise false
boolean isEmpty()
```

`ArrayIntList` review

- It's implemented with fields of an array and a size:

```
// post: returns the current number
// of elements in the list
public int size() {
   return size;
}
```

`ArrayIntList` review

- It's implemented with fields of an array and a size:

```
// post : returns the position of the first
//  occurrence of the given
//         value (-1 if not found)
public int indexOf(int value) {
  for (int i = 0; i < size; i++)
    if (elementData[i] == value)
      return i;
  return -1;
}
```

`ArrayIntList` review

- It's implemented with fields of an array and a size:

```
// post: returns true if the list
//       contains the given value,
//       false otherwise
public boolean contains(int value)
    return indexOf(value) >= 0;
}
```

`LinkedIntList` review

- Recall our `LinkedIntList` class

```
// adds given value to end of list
void add(E value)

// adds given value at the given index
void add(E value, int index)

// returns size of list
int size()

// returns true if the list is empty,
// otherwise false
boolean isEmpty()
```

`LinkedIntList` review

- It's implemented with a field for the front of the list:

```
// post: returns the current number
// of elements in the list
public int size() {
   int count = 0;
   ListNode current = front;
   while (current != null) {
      current = current.next;
      count++;
   }
   return count;
}
```

`LinkedIntList` review

- It's implemented with a field for the front of the list:

```
// post: returns the position of the first
//       occurrence of the given value
//       (-1 if not found)
public int indexOf(int value) {
  int index = 0;
  ListNode current = front;
  while (current != null) {
     if ( current.data == value )
        return index;
     current = current.next;
     index++;
  }
  return -1;
}
```

`LinkedIntList` review

- It's implemented with a field for the front of the list:

```
// post: returns true if the list
//       contains the given value,
//       false otherwise
public boolean contains(int value)
    return indexOf(value) >= 0;
}
```

`IntList` interface

- Both of these classes had all the same methods

- Recall that we can make an **`IntList`** interface that tells us what methods are required but not how they are implemented

```
public interface IntList {
   // adds given value to end of list
   void add(E value);

   // adds given value at the given index
   void add(E value, int index);

   // returns size of list
   int size();
   ...
```

`IntList` interface

- Both **`ArrayIntList`** and **`LinkedIntList`** can implement it

```
public class LinkedIntList implements IntList {
    ...
}

public class ArrayIntList implements IntList {
    ...
}
```

`IntList` interface

- Now we can declare our variables, return types, parameter types, etc as **`IntList`**

```
IntList list1 = new ArrayIntList();
list1.add(7);
list1.add(42);

IntList list2 = new LinkedIntList();
list2.add(8);
list2.add(50);
```

- Recall that this made our code more flexible and less redundant

`IntList` interface

- But this doesn't take care of all of our redundancy

- Both **`ArrayIntList`** and **`LinkedIntList`** have methods with exactly the same code in them
  - Check out **`contains(int value)`** and **`isEmpty()`**

`IntList` interface

- So where should we put this code? Let's try it in **`IntList`**

```
public interface IntList {
    public boolean contains(int value)
       return indexOf(value) >= 0;
   }
   ...
}
```

But this won't compile.

Abstract Classes

- Interfaces can only contain abstract methods
  - Methods without bodies of code
- We want to have a mix of abstract methods and normal methods
- Java allows us to do this with abstract classes

`AbstractIntList` class

- The class header specifies that the class is abstract:

```
public abstract class AbstractIntList {
   ...
}
```

- We want to make sure to use our interface:

```
public abstract class AbstractIntList
                              implements IntList {
   ...
}
```

AbstractIntList class

- We can also specify which methods should be abstract:

```
public abstract class AbstractIntList
                              implements IntList {
  public abstract void add(int value);

  public abstract void add(int value, int index);

  public abstract int size();

  ...
}
```

`AbstractIntList` class

- All the methods that are implementation-specific will be abstract since each specific class needs to implement them

- Since an abstract class has methods without bodies, we can't instantiate one

```
IntList list = new AbstractIntList();
```

`AbstractIntList` class

- Then we can fill in the methods we know how to implement:

```
public abstract class AbstractIntList
                              implements IntList {
  public abstract int indexOf(int value);

  public boolean contains(int value)
     return indexOf(value) >= 0;
  }
  ...
}
```

Methods with bodies are not declared abstract

`AbstractIntList` class

- Both **`ArrayIntList`** and **`LinkedIntList`** can extend it

```
public class LinkedIntList implements IntList {
    ...
}

public class ArrayIntList implements IntList {
    ...
}
```

Since **`AbstractIntList`** implements **`IntList`**, these both implement it as well. So our client code still works!

equals

- We've worked with **`Comparable`** to compare like objects
- But all **`Objects`** have an equals method
  - http://java.sun.com/javase/6/docs/api/java/lang/Object.html#equals(java.lang.Object)
- What do you think it means for one object to be equal to another?

equals

- The two `Objects` must be the same type of thing

```
String s1 = "hello";
Point p = new Point(42, 42);

s1.equals(p); // false
```

equals

- The two `Objects` must be considered equivalent

```
Point p1 = new Point(42, 42);
Point p2 = new Point(42, 42);
p1.equals(p2); // true
p1 == p1;      // true
p1 == p2;      // false
```

The `equals` method for `Points` checks whether they contain the same coordinates, while the equality operator checks that they are in the same location in memory

`equals`

- An **`AbstractIntList`** is a type of **`Object`**, so how would we write its **`equals`** method?

- We can look to Java's **`List`** interface for suggestions
  - http://java.sun.com/javase/6/docs/api/java/lang/Object.html#equals(java.lang.Object)

equals

- First, we need to make sure that the other object is an **`AbstractIntList`**:

```
public boolean equals(Object other) {
   if ( other instanceof IntList ) {
      ...
   }
   return false;
}
```

equals

- Then we need to make sure both lists are the same size:

```
public boolean equals(Object other) {
   if ( other instanceof IntList ) {
      IntList otherList = (IntList)other;
      if ( this.size() == otherList.size() ) {
         ...
      }
   }
   return false;
}
```

We can safely cast without runtime errors now that we know that `other` is an `IntList`

equals

- Finally, we need to check that the same elements occur in the same order:

```
public boolean equals(Object other) {
  if ( other instanceof IntList ) {
    IntList otherList = (IntList)other;
    if ( this.size() == otherList.size() ) {
      for (int i = 0; i < this.size(); i++) {
        if ( this.get(i) != otherList.get(i) )
          return false;
      }
      return true;
    }
  }
  return false;
}
```