8.0 Introduction

• Objects allow us to bundle related functionality into a single type
  – Vector, Point, Color
• For some data types, we may have:
  – A formal definition of allowable behaviors
  – Differing implementations of those behaviors
• The formal definition is abstract
  – We can see the signatures of allowable behaviors
  – But the implementation is not there at all
  – In Java, these are interfaces or abstract classes
• A concrete class can then implement the interface
• In this module we look at some common ADTs:
  – List
  – Set
  – Map
• We use these as implemented in Java for this module
• We implement them ourselves in a later module
8.1 Lists

• A list is
  – An ordered collection of elements
  – Duplication of elements is allowed in a list

• Analogies
  – Things I need to do today
8.1 Lists

• A list is
  – An ordered collection of elements
  – Duplication of elements is allowed in a list

• Analogies
  – Things I need to do today
    • Eat, study, eat, study, exercise, eat, sleep
8.1 Lists

• A list is
  – An ordered collection of elements
  – Duplication of elements is allowed in a list

• Analogies
  – Things I need to do today
    • Eat, study, eat, study, exercise, eat, sleep
      – Notice the duplication
8.1 Lists

• A list is
  – An ordered collection of elements
  – Duplication of elements is allowed in a list

• Analogies
  – Things I need to do today
    • Eat, study, eat, study, exercise, eat, sleep
      – Notice the duplication
      – Notice the order is important
        » Exercise before I eat
        » Sleep is the last thing I do
8.1 Lists

• A list is
  – An ordered collection of elements
  – Duplication of elements is allowed in a list

• Analogies
  – Things I need to do today
    • Eat, study, eat, study, exercise, eat, sleep
      – Notice the duplication
      – Notice the order is important
  – Errands I need to run
    • Get food, drop food off at neighbor's, pick up mail
      – No duplication, but order is important
8.1 Lists

• A list is
  – An ordered collection of elements
  – Duplication of elements is allowed in a list

• Analogies
  – Things I need to do today
    • Eat, study, eat, study, exercise, eat, sleep
      – Notice the duplication
      – Notice the order is important
  – Errands I need to run
    • Get food, drop food off at neighbor's, pick up mail
      – No duplication, but order is important
      – Cannot drop off food until it is first obtained
8.1 Lists

• A list of what?
  – Colors, Strings, doubles, Accounts?

• List is a parametric type
  – It is parameterized by the kind of thing in the list

• Syntax:
  – List<Color> colors;
  – List<String> names;
  – List<Double> values;
  – List<Account> customers;

• Pronunciation
  – List of Doubles
  – List of Strings
  – List of Doubles
  – List of Accounts
8.1 Lists

• **A list of what?**
  – Colors, Strings, doubles, Accounts?

• **List is a parametric type**
  – It is parameterized by the kind of thing in the list

• **Syntax:**
  – `List<Color>` colors;
  – `List<String>` names;
  – `List<Double>` values;
  – `List<Account>` customers;

• **Pronunciation**
  – List of Doubles
  – List of Strings
  – List of Doubles
  – List of Accounts
8.1 Lists

• A list of what?
  – Colors, Strings, doubles, Accounts?

• List is a parametric type
  – It is parameterized by the kind of thing in the list

• Syntax:
  – `List<Color> colors;`
  – `List<String> names;`
  – `List<Double> values;`
  – `List<Account> customers;`

• Pronunciation
  – List of Doubles
  – List of Strings
  – List of Doubles
  – List of Accounts

We would normally use `double` for this type, but for Lists and such, we need the "object" form of `double`, which is spelled `Double`
8.1 Lists

• A list of what?
  – Colors, Strings, doubles, Accounts?
8.1 Lists

- A list of what?
  - Colors, Strings, doubles, Accounts?
- List is a parametric type
- A List<T> contains elements only of type T
8.1 Lists

• A list of what?
  – Colors, Strings, doubles, Accounts?

• List is a parametric type

• A List<T> contains elements only of type T
  – As we learn later, T can be a general type
  – Example: List<Number> can contain
    • Double, Integer, Long, etc.
8.1 Lists

• A list of what?
  – Colors, Strings, doubles, Accounts?
• List is a parametric type
• A List<T> contains elements only of type T
  – As we learn later, T can be a general type
  – Example: List<Number> can contain
    • Double, Integer, Long, etc.
  – We say that Number is polymorphic
    • Many-shaped
    • Could be Double, Integer, Long, etc.
8.1 Lists

• A list of what?
  – Colors, Strings, doubles, Accounts?

• List is a parametric type

• A List<T> contains elements only of type T
  – As we learn later, T can be a general type
  – Example: List<Number> can contain
    • Double, Integer, Long, etc.
  – We say that Number is polymorphic
    • Many-shaped
    • Could be Double, Integer, Long, etc.

• Object is the most general type of all
  – So, List<Object> can contain any type of object

• For now, we will use a very specific type
  – But we call it type T, where T is the parametric type
What are some examples of Lists?

Examples

- List of books I read this summer, in order of their completion (String)
- List of hourly temperature readings
  - Pause and ask student
  - Double
- List of Roulette Wheel results
  - Pause and ask student
  - Integer

Where would the following be useful?

- List of Boolean
- Pause
- Heads tails results

Back to a list of things to do

- What kinds of things would you want to do with a list?
- Ask student
- Live response at roundtable
8.1 Lists

• If we have a list, what can we do to it

– Add things to the list
– Find the $n$th thing on the list
– Find out where some element occurs on the list
– Remove something from the list
– Find out the length of the list

• No matter how the List is implemented, we should be able to perform these operations to any List

– Specific implementations may differ in how they carry out the above operations

• Let's define the interface, and then use some specific implementations of it
8.1 Lists

• If we have a list, what can we do to it
  – Add things to the list
8.1 Lists

- If we have a list, what can we do to it
  - Add things to the list
  - Find the $n^{th}$ thing on the list
8.1 Lists

• If we have a list, what can we do to it
  – Add things to the list
  – Find the $n^{th}$ thing on the list
  – Find out where some element occurs on the list
8.1 Lists

• If we have a list, what can we do to it
  – Add things to the list
  – Find the \( n^{\text{th}} \) thing on the list
  – Find out where some element occurs on the list
  – Remove something from the list
8.1 Lists

• If we have a list, what can we do to it
  – Add things to the list
  – Find the n\textsuperscript{th} thing on the list
  – Find out where some element occurs on the list
  – Remove something from the list
  – Find out the length of the list
8.1 Lists

• If we have a list, what can we do to it
  – Add things to the list
  – Find the \( n^{\text{th}} \) thing on the list
  – Find out where some element occurs on the list
  – Remove something from the list
  – Find out the length of the list

• No matter how the List is implemented, we should be able to perform these operations to any List
  – Specific implementations may differ in how they carry out the above operations

• Let's define the interface, and then use some specific implementations of it
8.1 Lists

public interface List<T> {

    - Uses interface instead of class

}
8.1 Lists

public interface List\<T\> {

    // Uses interface instead of class
    // Has the parametric type in the class name

}
8.1 Lists

```java
public interface List<T> {

    // No constructors!
    // Because this is not really a class, it just defines the signatures of available methods
}
```
public interface List<T> {

  // No constructors!
  // Because this is not really a class, it just defines the signatures of available methods

  // No instance variables
  // Because no implementations of methods are here to remember things

}

8.1 Lists
8.1 Lists

```java
public interface List<T> {
    public boolean add(T e);  // Add element e to the end of this list
    // Always returns true for a list
}
```
8.1 Lists

public interface List<T> {

    public boolean add(T e);

    public T get(int i);

    // Return the i<sup>th</sup> element in this list

}
public interface List<T> {

    public boolean add(T e);
    public T get(int i);
    public int indexOf(T e);

    // Returns the index at which the specified element first occurs in the list
    // 0 is the index of the first element
}
8.1 Lists

public interface List<T> {

    public boolean add(T e);

    public T get(int i);

    public int indexOf(T e);

    public boolean remove(T e);

}
8.1 Lists

public interface List<T> {
    public boolean add(T e);
    public T get(int i);
    public int indexOf(T e);
    public boolean remove(T e);
    public T remove(int i);
}

- Remove the first occurrence of the specified element
- Remove (and return) the element at the $i^{th}$ index in the list
public interface List<T> {

    public boolean add(T e);

    public T get(int i);

    public int indexOf(T e);

    public boolean remove(T e);

    public T remove(int i);

    public int size();

}
8.1 Lists

public interface List<T> { 

    public boolean add(T e);
    public T get(int i);
    public int indexOf(T e);
    public boolean remove(T e);
    public T remove(int i);
    public int size();

}
8.1 Lists

• Interface
  – Defines the signatures of the methods we want
    • "What" not "How"
    • No instance variables, constructors
    • Design without implementation
      – So that alternative implementations can satisfy the interface

• An implementation
  – Says it "implements" the interface
  – Must have actual implementations for all methods
  – If it is missing any, it cannot be instantiated
8.1 Lists

- So we have an interface
- What implementations exist? (easy to see online)
  - **ArrayList**
    - Places elements in an array
    - Insertion and deletion are relatively awkward
    - Good choice if the list is relatively fixed
    - Example: List of America's National Parks
      - Does not change frequently
      - Deletion is rare to non-existant
  - **LinkedList**
    - Grows and shrinks with ease
    - Uses "links" between its elements
    - Insertion and deletion are relatively easy
    - Example: People waiting in line to ride a roller coaster
      - Mostly the list grows, but it grows frequently
      - Occasionally people drop out—scared or don't want to wait

- **Choosing the right data structure is important**
  - A more rigorous, mathematical treatment awaits you in a course on Analysis of Algorithms
List<String> eating = new LinkedList<String>();

- We declare `eating` as a variable of type `List<String>`
  - We can then invoke any method defined in the `List` interface on `eating`
  - We could have declared `eating` to be a `LinkedList<String>` and the program would have worked just as well
  - But `List<String>` is more general, so we should use that when possible
8.1 Lists

```java
List<String> eating = new LinkedList<String>();
```

- Here, we cannot say `new List<String>()`
  - Because `List` is an interface, not a concrete class
  - Interfaces cannot be constructed
- So on the right-hand-side, we must assign some object of type `List<String>` or narrower
  - Because `LinkedList<String>` implements `List<String>`, it serves just fine
8.1 Lists

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

- Or we could use an ArrayList
  - Because it implements the List interface
  - Plug-compatible in terms of functionality
  - But time / space behavior may be different
8.1 Lists

List<String> eating = new LinkedList<String>();

eating.add("open mouth");
eating.add("insert food");
eating.add("chew");
eating.add("chew");
eating.add("swallow");

System.out.println(eating);
8.2 Exercise

- Investigate the ADT Set<T> and its concrete implementation HashSet<T>
- How does it differ from List?
- What do Set and List have in common
- Try

```java
Set<String> eating=new HashSet<String>();
```
- As before, add the following
  
```java
eating.add("open mouth");
eating.add("insert food");
eating.add("chew");
eating.add("chew");
eating.add("swallow");
```
- Print out the set, and what do you see?
8.2 Exercise

• **Video response**
  – Sets do not allow duplicates
  – The order of adding elements does not matter
8.3 Object equality

- For primitive types, equality is simple
  - `double`, `int`, `boolean`
  - These are equal if they have the same value
- For objects, equality is not as simple.
- By equality of `a` and `b`, do we mean
  - `a` and `b` are the exact same object?
  - `a` and `b` are equal in a deeper sense of the concept?
8.3 Object equality

By equality of \( a \) and \( b \), do we mean

\( a \) and \( b \) are the exact same object?

\[ a \neq b \]
8.3 Object equality

• Let's look at a simple class that represents an (x,y) pair
  – A Point

```java
public class Point {
    private int x, y;

    ...
}
```
8.3 Object equality

• Let's look at a simple class that represents an (x,y) pair
  – A Point
  – The one shown to the right is (5,10)

```java
public class Point {
    private int x, y;

    ...
}
```
8.3 Object equality

By equality of $a$ and $b$, do we mean

– $a$ and $b$ are the exact same object?
– $a$ and $b$ are equal in a deeper sense of the concept?
8.3 Object equality

By equality of a and b, do we mean
– a and b are the exact same object?
  • for this, we test a == b
8.3 Object equality

By equality of \( a \) and \( b \), do we mean

\( a \) and \( b \) are equal in a deeper sense of the concept?

• for this, we test \( a \text{.equals}(b) \)
8.3 Object equality

• All objects have the method
  – boolean equals(Object other)
  – Just as all objects have String toString()

• The default implementation is simply:

```java
public boolean equals(Object other) {
    return this == other;
}
```

• The above tests whether the two objects are exactly the same object
  – In other words, the live at the same address in the heap
8.3 Object equality

• But we can always override this definition and give our own meaning to equality

• For a String object, this is based on whether the strings:
  – Have the same length
  – Match, character-by-character

• For a Color object, equality is (essentially) based on whether the two objects have the same values for:
  – red
  – green
  – blue

• Because
  – a=Color.BLACK

• has the same amount of red, green, and blue as
  – b=new Color(0,0,0),

• a.equals(b) is true
8.3 Object equality

- Java objects we have used so far have reasonable definitions for
  - `boolean equals(Object o)`

- Object such as
  - Color
  - String
  - Double
  - Integer

- The `Set` implementation uses an object's `.equals(Object)` methods to tell whether an element is already in a `Set`
8.3 Object equality

- **Contract for boolean equals(Object o)**
  - reflexive
    - x "is as tall as" x
    - x "has same name as" x
8.3 Object equality

• **Contract for** boolean equals(Object o)
  – reflexive
    • a.equals(a) is always true
8.3 Object equality

• **Contract for boolean equals(Object o)**
  – reflexive
    • a.equals(a) is always true
  – symmetric
    • If you "are a sister of" Sue, Sue "is a sister of" you
    • x "same color as" y
      – then y "same color as" x
8.3 Object equality

• **Contract for boolean equals(Object o)**
  – **reflexive**
    • `a.equals(a)` is always true
  – **symmetric**
    • if `a.equals(b)` then `b.equals(a)`
8.3 Object equality

• **Contract for boolean equals(Object o)**
  
  – **reflexive**
  
    • `a.equals(a)` is always true
  
  – **symmetric**
  
    • if `a.equals(b)` then `b.equals(a)`
  
  – **transitive**
  
    • if a "same height as" b
    • and b "same height as" c
    • then a "same height as" c
8.3 Object equality

• **Contract for** `boolean equals(Object o)`
  
  – **reflexive**
  
  • `a.equals(a)` is always true
  
  – **symmetric**
  
  • if `a.equals(b)` then `b.equals(a)`
  
  – **transitive**
  
  • if `a.equals(b)` and `b.equals(c)`
    
    – then `a.equals(c)`
8.3 Object equality

- **Contract for boolean equals(Object o)**
  - **reflexive**
    - a.equals(a) is always true
  - **symmetric**
    - if a.equals(b) then b.equals(a)
  - **transitive**
    - if a.equals(b) and b.equals(c)
      - then a.equals(c)
  - **consistent**
    - a.equals(b) returns the same result if invoked multiple times, unless something changes about a or b
8.3 Object equality

What would be an inconsistent implementation?

```java
public boolean equals(Object o) {
    return Math.random() < 0.5;
}
```

- **consistent**
  - `a.equals(b)` returns the same result if invoked multiple times, unless something changes about `a` or `b`
8.3 Object equality

• These properties make sense for objects we consider in everyday life
  — Color
  — String
8.3 Object equality

• These properties make sense for objects we consider in everyday life
  – Color
  – String

• Moreover
  – a.equals(null) is always false
  – Objects of different concrete types are always different
    • if a is a Color and b is a String
8.3 Object equality

- These properties make sense for objects we consider in everyday life
  - Color
  - String

- Moreover
  - `a.equals(null)` is always false
  - Objects of different concrete types are always different
    - if `a` is a Color and `b` is a String
      - `a.equals(b)` is always false

- Eclipse can generate the equals method automatically
  - We just have to say which fields matter in terms of equality. This should be considered as we design objects.
public class Point {
    private int x, y;
    // Generated automatically by eclipse
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        Point other = (Point) obj;
        if (x != other.x) return false;
        if (y != other.y) return false;
        return true;
    }
}
public class Point {
    private int x, y;
    // Generated automatically by eclipse
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        Point other = (Point) obj;
        if (x != other.x) return false;
        if (y != other.y) return false;
        return true;
    }
}

8.3 Object equality

Because
    a.equals(a)
should always be true
8.3 Object equality

public class Point {
    private int x, y;
    // Generated automatically by eclipse
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        Point other = (Point) obj;
        if (x != other.x) return false;
        if (y != other.y) return false;
        return true;
    }
}

Because

    a.equals(null)

should always be false
public class Point {
    private int x, y;
    // Generated automatically by eclipse
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        Point other = (Point) obj;
        if (x != other.x) return false;
        if (y != other.y) return false;
        return true;
    }
}

8.3 Object equality

Because objects of differing types should never equal each other
public class Point {
    private int x, y;
    // Generated automatically by eclipse
    public boolean equals(Object obj) {
        Point other = (Point) obj;
        if (x != other.x) return false;
        if (y != other.y) return false;
        return true;
    }
}

8.3 Object equality

If we reach this point, then the parameter obj is also a Point object. This cast lets us treat it as such under the name other
public class Point {
    private int x, y;
    // Generated automatically by eclipse
    public boolean equals(Object obj) {
        if (this == obj) return true;  // If we reach this point, then the parameter obj is also a Point object. This cast lets us treat it as such under the name other. We told eclipse that we want equality based on the instance variables x and y
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        Point other = (Point) obj;
        if (x != other.x) return false;
        if (y != other.y) return false;
        return true;
    }
}
public class Point {
    private int x, y;
    // Generated automatically by eclipse
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        Point other = (Point) obj;
        if (x != other.x) return false;
        if (y != other.y) return false;
        return true;
    }
}
public class Point {
    private int x, y;
    // Generated automatically by eclipse
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        Point other = (Point) obj;
        if (x != other.x) return false;
        if (y != other.y) return false;
        return true;
    }
}

If we reach this point, then the parameter obj is also a Point object. This cast lets us treat it as such under the name other. We told eclipse that we want equality based on the instance variables x and y.

```java
if (x != other.x) return false;
if (y != other.y) return false;
return true;
```
public class Point {
    private int x, y;
    // Generated automatically by eclipse
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        Point other = (Point) obj;
        if (x != other.x) return false;
        if (y != other.y) return false;
        return true;
    }
}

8.3 Object equality

If we reach this point, then the parameter obj is also a Point object. This cast lets us treat it as such under the name other. We told eclipse that we want equality based on the instance variables x and y.

If we make it to this point, the two objects (this and other) agree on their x and y values, so we return true.

    return true;
}
8.3 Object hashCode

- Suppose you have a messy house and it takes you forever to find something
8.3 Object hashCode

- Suppose you have a messy house and it takes you forever to find something
- One possible strategy is to get some bins
  - A given object (say, your cell phone) is always put in the same bucket so it is easier to find
  - Requires remembering the bucket in which the phone goes
8.3 Object hashCode

• Suppose you have a messy house and it takes you forever to find something
• One possible strategy is to get some bins
  – A given object (say, your cell phone) is always put in the same bucket so it is easier to find
  – Requires remembering the bucket in which the phone goes
  – But cuts the search space down to just 1/6 buckets

![Diagram showing a bucket labeled 'phone' among five buckets numbered 0 to 5.]
8.3 Object hashCode

• Suppose you have a messy house and it takes you forever to find something
• One possible strategy is to get some bins
• Java has this concept implemented for every object
  – Called an object's `hashCode()`
  – Maps an object to a (possibly large) integer
8.3 Object hashCode

- Suppose you have a messy house and it takes you forever to find something
- One possible strategy is to get some bins
- Java has this concept implemented for every object
  
  - `phone.hashCode() % 6` computes which bucket

```
phone
0 1 2 3 4 5
```
8.3 Object hashCode

- Bigger picture
  - `phone.hashCode() % 6` computes which bucket
8.3 Object hashCode

- **Bigger picture**
  - `phone.hashCode() % 6` computes which bucket
  - `phone.equals(thing)` is then applied to each `thing` in that bucket to see if the phone is really there
8.3 Object hashCode

• Bigger picture

  - `phone.hashCode() % 6` computes which bucket
  - `phone.equals(thing)` is then applied to each thing in that bucket to see if the phone is really there

Bucket 2

- pencil
- Taj Mahal
- phone
- moss-covered 3-handled family credenza
8.3 Object hashCode

- `phone.hashCode() % 6` computes which bucket
- `phone.equals(thing)` is then applied to each thing in that bucket to see if the phone is really there
8.3 Object hashCode

• Bigger picture
  – `phone.hashCode() % 6` computes which bucket
  – `phone.equals(thing)` is then applied to each `thing` in that bucket to see if the phone is really there
8.3 Object hashCode

- `phone.hashCode() % 6` computes which bucket
- `phone.equals(thing)` is then applied to each `thing` in that bucket to see if the phone is really there
8.3 Object hashCode

- Let's consider a `Point` object
  - whose `hashCode()` is generated based on its x and y coordinates
8.3 Object hashCode

• Some possible hashCode() implementations
  – All objects map to same bucket, not good

```java
public int hashCode() {
    return 0;
}
```

(4,3).hashCode()
8.3 Object hashCode

• Some possible `hashCode()` implementations
  – All objects map to same bucket, not good

```java
public int hashCode() {
    return 0;
}
```
8.3 Object hashCode

- Some possible `hashCode()` implementations
  - All objects map to same bucket, not good

```java
public int hashCode() {
    return 0;
}
```

```
(4,3).hashCode()  (3,4).hashCode()  (8,2).hashCode()

0  1  2  3  4  5
```
8.3 Object hashCode

- Some possible `hashCode()` implementations
  - How about this?

```java
public int hashCode() {
    return x+y;
}
```
8.3 Object hashCode

- Some possible `hashCode()` implementations
  - How about this?

```java
public int hashCode() {
    return x+y;
}
```

Diagram:
- `(4,3) .hashCode()`
- `(3,4) .hashCode()`
- `(8,2) .hashCode()`
8.3 Object hashCode

• Some possible hashCode() implementations
  – How about this? Better, but (x,y) and (y,x) end up in same bucket

```java
public int hashCode() {
    return x+y;
}
```

- `(4,3)` .hashCode()
- `(3,4)` .hashCode()
- `(8,2)` .hashCode()
8.3 Object hashCode

• Some possible `hashCode()` implementations
  – How about this one? Spreads out the objects much more evenly

```java
public int hashCode() {
    return 31*x + 17*y;
}
```

(8,2) .hashCode()

(3,4) .hashCode()

(4,3) .hashCode()
8.3 Object hashCode

- Some possible `hashCode()` implementations
  - How about this one? Spreads out the objects much more evenly

```java
public int hashCode() {
    return 31*x + 17*y;
}
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3,4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4,3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(8,2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
8.3 Object hashCode

- Some possible `hashCode()` implementations
  - How about this one? Spreads out the objects much more evenly

```java
public int hashCode() {
    return 31*x + 17*y;
}
```

![Diagram showing object positions and hashCode calculations]
8.3 Object hashCode

- Impossible to do a perfect job
  - Many points and only 6 buckets
8.3 Object hashCode

• Impossible to do a perfect job
  – Many points and only 6 buckets
8.4 Common mistake with equals

• Code doesn't work because equals was improperly defined by hand
  – wrong signature
  – also no hashCode override

• To override a method, its signature must match exactly

• Show student the right thing to do
  – Say what you want to count for equality
  – Use eclipse to generate the methods
8.5 Lists and Sets of Points

• Video intro: use Lists and Sets now but of Point objects
• Make a point object with instance variables
  – x and y, both integers
• Use eclipse to generate
  – .equals (and it also generates hashCode – this is fine)
• Add different points to the List and Set, but with some duplicates added on purpose
• What do you see?
• Response goes over what they should see
  – Say something about hashCode
8.5 A StockHolding object

- Video intro: another object, and we'll let eclipse generate equals and hashCode
- A Stock has-a
  - String identifying its code (final)
  - String identifying its owner (who holds the stock) (final)
  - A number of shares (integer), may change over time
  - A price per share (integer), may change over time
- Question card:
  - On what should equality be based?
  - Code this up and generate its equals and hashCode using eclipse
- Response: show my solution
8.6 Interfaces for behavioral abstraction

• We saw that List<T> is an interface
  – From the top-down
    • List<T> defines what we want a list to do
      – add, remove, size, etc.
  – From the bottom-up
    • List<T> defines the features common among its implementations
      – LinkedList, ArrayList, etc.

• We next look at creating the interface from separate objects
  – So as to leverage the common behaviors of those objects
8.6 Interfaces for behavioral abstraction

- **Object 1 is a BankAccount**
  - *We have seen this previously*
    - deposit, withdraw, getBalance

- **Object 2 is a StockHolding**
  - *has-a*
    - number of shares (does not change)
    - price per share (can change over time)
  - *behaviors*
    - change the price per share, get current value

- **These two objects are fine as separate objects**
8.6 Interfaces for behavioral abstraction

- **Object 1 is a BankAccount**
  - *We have seen this previously*
    - deposit, withdraw, getBalance

- **Object 2 is a StockHolding**
  - *has-a*
    - number of shares (does not change)
    - price per share (can change over time)
  - *behaviors*
    - change the price per share, get current value

- **These two objects are fine as separate objects**
  - *Now suppose I want to find out your current liquid worth in dollars based on stocks and bank accounts*
8.6 Interfaces for behavioral abstraction

• Why?
• Goal:

```java
List<Valuable> assets=new LinkedList<Valuable>();
```

Construct an empty list of objects that implement the `Valuable` interface. This list is actually a `LinkedList` of such objects.

What should it mean to be `Valuable`? It means that we can find out the current value of the object (say, in dollars), whether it is a `BankAccount` or a `StockHolding`.

We will insist that `BankAccount` and `StockHolding` objects implement the `Valuable` interface.
8.6 Interfaces for behavioral abstraction

• Why?
• Goal:

```java
List<Valuable> assets=new LinkedList<Valuable>();
assets.add(new BankAccount(100));
assets.add(new StockHolding(5, 10));
```

We can add anything that implements the Valuable interface to this list

Here, we add a BankAccount that starts with 100 dollars
8.6 Interfaces for behavioral abstraction

• Why?
• Goal:

```java
List<Valuable> assets=new LinkedList<Valuable>();
assets.add(new BankAccount(100));
assets.add(new StockHolding(5, 10));
```

We can add anything that implements the `Valuable` interface to this list.

Here, we add a `BankAccount` that starts with 100 dollars and a `StockHolding` of 5 shares at initial value 10 dollars per share.
8.6 Interfaces for behavioral abstraction

• Why?
• Goal:

```java
List<Valuable> assets=new LinkedList<Valuable>();
assets.add(new BankAccount(100));
assets.add(new StockHolding(5, 10));
int worth = 0;
for (int i=0; i<assets.size(); ++i) {
    Valuable asset = assets.get(i);
    worth += asset.getLiquidValue();
}
```

We can now iterate over the list, adding up the current liquid value of each element.
8.6 Interfaces for behavioral abstraction

- Why?
- Goal:

```java
List<Valuable> assets=new LinkedList<Valuable>();
assets.add(new BankAccount(100));
assets.add(new StockHolding(5, 10));
int worth = 0;
for (int i=0; i<assets.size(); ++i) {
    Valuable asset = assets.get(i);
    worth += asset.getLiquidValue();
}
System.out.println("You now have \$" + worth);
```

We can now iterate over the list, adding up the current liquid value of each element. This loop starts at 0 and goes up to but not including the size of the list.
8.6 Interfaces for behavioral abstraction

• Why?
• Goal:

```
List<Valuable> assets=new LinkedList<Valuable>();
assets.add(new BankAccount(100));
assets.add(new StockHolding(5, 10));
int worth = 0;
for (int i=0; i<assets.size(); ++i) {
    Valuable asset = assets.get(i);
    worth += asset.getLiquidValue();
}
System.out.println("You now have "+worth);
```

We retrieve element i from the list, and it implements the Valuable interface, whether it is a BankAccount or a StockHolding object.
8.6 Interfaces for behavioral abstraction

• Why?
• Goal:

```java
List<Valuable> assets=new LinkedList<Valuable>();
assets.add(new BankAccount(100));
assets.add(new StockHolding(5, 10));
int worth = 0;
for (int i=0; i<assets.size(); ++i) {
    Valuable asset = assets.get(i);
    worth += asset.getLiquidValue();
}
System.out.println("You now have "+ worth);
```

We add to `worth` the value of this asset, so that the loop causes `worth` to eventually contain the value of all of the assets.
8.6 Interfaces for behavioral abstraction

• Why?
• Goal:

```java
List<Valuable> assets = new LinkedList<Valuable>();
assets.add(new BankAccount(100));
assets.add(new StockHolding(5, 10));
int worth = 0;
for (int i = 0; i < assets.size(); ++i) {
    Valuable asset = assets.get(i);
    worth += asset.getLiquidValue();
}
System.out.println("You now have $" + worth);
```

Finally, we can print the result of the computation.
8.6 Interfaces for behavioral abstraction

• Why?
• Goal:

```java
List<Valuable> assets=new LinkedList<Valuable>();
assets.add(new BankAccount(100));
assets.add(new StockHolding(5, 10));
int worth = 0;
for (int i=0; i<assets.size(); ++i) {
    Valuable asset = assets.get(i);
    worth += asset.getLiquidValue();
}
System.out.println("You now have $" + worth);
```

Java offers a cleaner way to state this iteration.
8.6 Interfaces for behavioral abstraction

• Why?
• Goal:

```java
List<Valuable> assets = new LinkedList<Valuable>();
assets.add(new BankAccount(100));
assets.add(new StockHolding(5, 10));
int worth = 0;
for (Valuable asset : assets) {
    //
    worth += asset.getLiquidValue();
}
System.out.println("You now have $" + worth);
```

Java offers a cleaner way to state this iteration. Objects such as List and Set offer an iterator that can be used in this way.
8.6 Interfaces for behavioral abstraction

- Why?
- Goal:

```java
List<Valuable> assets=new LinkedList<Valuable>();
assets.add(new BankAccount(100));
assets.add(new StockHolding(5, 10));
int worth = 0;
for (Valuable asset : assets) {
    //
    worth += asset.getLiquidValue();
}
System.out.println("You now have $" + worth);
```

This declares a new variable asset, of type Valuable
### 8.6 Interfaces for behavioral abstraction

- **Why?**
- **Goal:**

```java
List<Valuable> assets=new LinkedList<Valuable>();
assets.add(new BankAccount(100));
assets.add(new StockHolding(5, 10));
int worth = 0;
for (Valuable asset : assets) {
    //
    worth += asset.getLiquidValue();
}
System.out.println("You now have $" + worth);
```

This declares a new variable `asset`, of type `Valuable`, which takes on each element of the `assets` list in turn.
8.6 Interfaces for behavioral abstraction

- Why?
- Goal:

```java
List<Valuable> assets = new LinkedList<Valuable>();
assets.add(new BankAccount(100));
assets.add(new StockHolding(5, 10));
int worth = 0;
for (Valuable asset : assets) {
    //
    worth += asset.getLiquidValue();
}
System.out.println("You now have $" + worth);
```

For a List, the elements will be retrieved in their list order. If this were a Set, there is no implied ordering of its elements.
8.6 Interfaces for behavioral abstraction

• How do we achieve this goal?
• BankAccount offers
  – `getBalance()`
• StockHolding offers
  – `getCurrentValue()`
• So we define a new method
  – `getLiquidValue()`
    • in terms of what each object offers
• And we add the interface
  – `Valuable`
    • offers the method `getLiquidValue()`
• Both objects then implement that interface
8.6 Interfaces for behavioral abstraction

```java
int getBalance()
deposit(int)
boolean withdraw(int)
```

BankAccount
8.6 Interfaces for behavioral abstraction

```java
public interface BankAccount {
    int getBalance();
    void deposit(int amount);
    boolean withdraw(int amount);
}

public interface StockHolding {
    int getCurrentValue();
    void setPricePerShare(int price);
}
```
8.6 Interfaces for behavioral abstraction

```java
public interface BankAccount {
    int getBalance();
    void deposit(int amount);
    boolean withdraw(int amount);
    int getLiquidValue();
}

public interface StockHolding {
    int getCurrentValue();
    void setPricePerShare(int price);
    int getLiquidValue();
}
```
8.6 Interfaces for behavioral abstraction

```
BankAccount

- deposit(int)
- withdraw(int)
- getBalance()
- getCurrentValue()
- getLiquidValue()

StockHolding

- setPricePerShare(int)
- getLiquidValue()
```

```
8.6 Interfaces for behavioral abstraction

- int getBalance()
- deposit(int)
- boolean withdraw(int)
- int getLiquidValue()

- int getCurrentValue()
- setPricePerShare(int)
- int getLiquidValue()
8.6 Interfaces for behavioral abstraction

Viewed as their own object types, all methods defined on an object are available, including the newly added method.
8.6 Interfaces for behavioral abstraction

Viewed through the interface, only the interface's methods are available.

```java
int getLiquidValue()
```

```java
int getLiquidValue()
```
Although `a` and `v` both reference the same object

- On `a`, we can call any method defined for a `BankAccount`
- On `v`, we can only call those methods defined for a `Valuable`
Although \( a \) and \( v \) both reference the same object

- On \( a \), we can call any method defined for a BankAccount
- On \( v \), we can only call those methods defined for a Valuable
8.6 Interfaces for behavioral abstraction

Although a and v both reference the same object

- On a, we can call any method defined for a BankAccount
- On v, we can only call those methods defined for a Valuable
8.6b Using the interface

- Define a class `PersonalProperty`
  - has-a `initialValue` (integer)
  - has-a `yearsOld` (integer)
  - define a method `int depreciatedValue()`
    - returns the value according to the formula: `initialValue x (0.80)^yearsOld`
    - and cast to an int value

- **Remember**
  - `Math.pow(a,b)` computes $a^b$

- **Question Card:** adapt this class to implement `Valuable`

- **Response:** show how this is done
8.7 Conclusion

- **ADTs** are defined in terms of their behaviors
  - They allow for multiple implementations
  - Java interfaces express the common behaviors

- **List, Set, Map** are useful **ADTs**
  - We have see List in great detail
  - And Set in some detail
  - Map deserves further exploration

- **Objects** need to be able to compare themselves against other objects for equality
  - `.equals` method
  - contract
  - implementation (automatically using eclipse)

- **Interfaces** can be define bottom-up to extract a common behavior of interest
  - Refactor existing code to introduce common methods
  - Capture common methods with an interface