

CSE1030 – Introduction to Computer Science II

Lecture #8 Aggregation & Composition II

Goals for Today

- Goals
 - Theory:
 - Composition versus Aggregation
- Practical:
 - More About Collections
 - Iterators
 - Shallow Copy versus Deep Copy

CSE1030 2

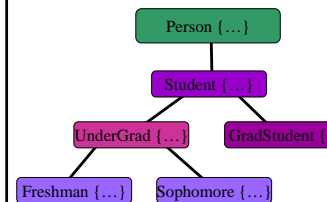
CSE1030 – Lecture #8

- Review: “is-a” versus “has-a”
- Theory: Composition versus Aggregation
- Iteration
- Shallow vs. Deep Copy
- We’re Done!

CSE1030 3

Review “is-a” versus “has-a”

- “is-a”
 - e.g., Class Hierarchy:
- “has-a”
 - e.g., Person Class:



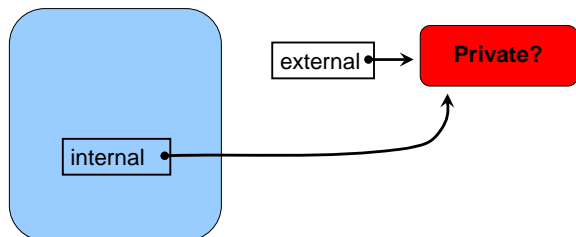
```
public class Person
{
    // attributes
    private String Name;
    private int Age;
    private int Weight;

    Person(String name, int age,
            int weight)
    {
        Name = name;
        Age = age;
        Weight = weight;
    }
    ...
}
```

CSE1030 4

Privacy Leaks

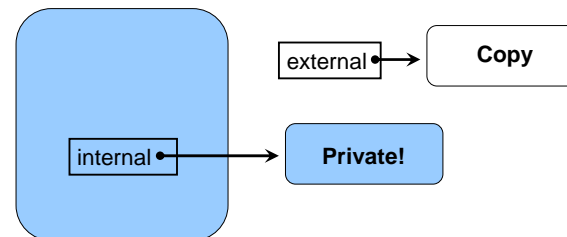
- When somebody “outside” gets a copy of an object meant to be securely “inside”...



CSE1030 5

Privacy Leaks

- This is fixed by using the Copy Constructor to copy sensitive objects on their way into and out of a class (i.e., in the accessor and mutator):



CSE1030 6

Privacy Leaks

```
import java.util.*;

public class PrivacyLeak
{
    private HashSet<Person> students
        = new HashSet<Person>();

    // constructor
    public PrivacyLeak()
    { students = new HashSet<Person>(); }

    // add
    public void add(Person p)
    { students.add(p); }
```

Privacy Leak

CSE1030 7

```
public static void main(String[] args)
{
    PrivacyLeak course = new PrivacyLeak();

    // create some students
    Person sally = new Person("Sally Single", 32);
    Person frank = new Person("Frank", 44);
    Person billy = new Person("Billy", 36);

    // add them to my collection
    course.add(sally);
    course.add(frank);
    course.add(billy);

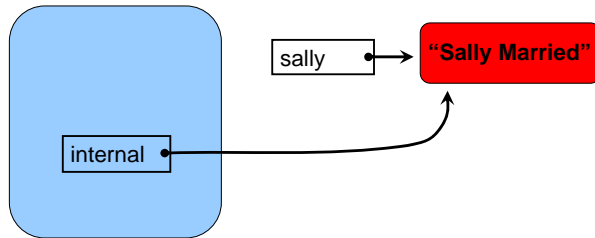
    // Sally gets married and changes her name...
    sally.setName("Sally Married");

    System.out.println("Class List:");
    for(Person p : course.students)
        System.out.println("    " + p.getName());
}
```

CSE1030 8

Output – Why?

Class List:
Billy
Frank
Sally Married



CSE1030 9

No Privacy Leak

```
import java.util.*;

public class PrivacyLeak
{
    private HashSet<Person> students
        = new HashSet<Person>();

    // constructor
    public PrivacyLeak()
    { students = new HashSet<Person>(); }

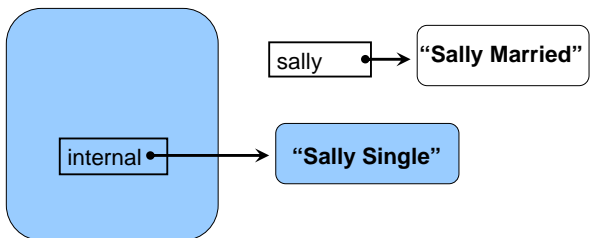
    // add
    public void add(Person p)
    { students.add(new Person(p)); }
```

No Privacy Leak

CSE1030 10

Output – Why?

Class List:
Billy
Frank
Sally Single



CSE1030 11

CSE1030 – Lecture #8

- Review: “is-a” versus “has-a”
- Theory: Composition versus Aggregation
- Iteration
- Shallow vs. Deep Copy
- We’re Done!

CSE1030 12

Big Theory Idea for Today

- There is an important distinction between code that **uses** an object, and the code that is **responsible for managing** an object
- Ideally: **Responsibility** implies **Ownership**
- The terms we use for this are **Aggregation** versus **Composition**
 - Aggregation = **Using** or **Servicing** an object
 - Composition = **Ownership** → **Responsibility**

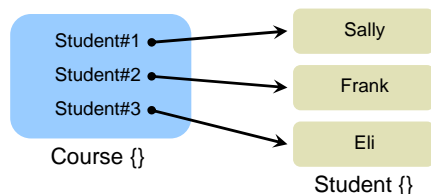
CSE1030 13

Big Theory Idea for Today

- Examples:
 - **Composition** (means **defining / constructing**)
 - Person owns Name
 - CreditCard owns Balance (and TotalBalance)
 - **Aggregation** (means **collecting**)
 - A Person doesn't own their Friend
 - CreditCard doesn't own the Interest Rate
- The idea is pure, but in the real world, the distinction is often arbitrary, and depends upon one's perspective

CSE1030 14

Course and Student Example:



- Student Name Changes
 - Who is responsible for the accuracy of the information?
 - Does updated information need to be propagated? (*next...*)
- Aggregation / Composition depends upon perspective
 - The Students own their objects (composition)
 - The Professor uses those same objects (aggregation)

CSE1030 15

Implications – General Rules

- If you are responsible for something
 - You should own it (composition)
 - You should control access to it
 - Private + appropriate accessor / mutator
 - Beware of **Privacy Leaks**
 - Which basically just means the owner isn't being responsible for changes (use: **Copy Constructor**)
- If you are only using something
 - Don't copy it if you don't have to (Be Efficient!)
 - Use it nicely (use accessors and mutators)
 - If you don't copy it, you get updates "for free"

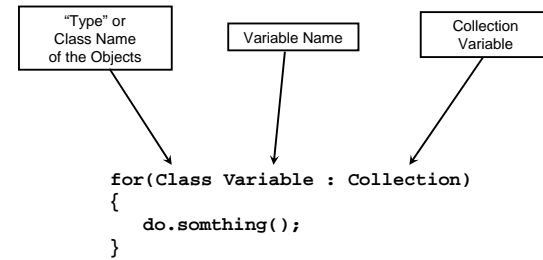
CSE1030 16

CSE1030 – Lecture #8

- Review: “is-a” versus “has-a”
- Theory: Composition versus Aggregation
- Iteration
- Shallow vs. Deep Copy
- We’re Done!

CSE1030 17

Automatic Iteration: “For-Each”



- This form of “for-loop” is called “for-each”
- It only works with objects that provide **iterators** (the Java Collections do)

CSE1030 18

```
import java.util.*;

public class set
{
    public static void main(String[] args)
    {
        // create a set to store my friends
        HashSet<Person> friends = new HashSet<Person>();

        // create some friends
        Person sally = new Person("Sally", 32);
        Person frank = new Person("Frank", 44);
        Person billy = new Person("Billy", 36);

        // add them to my collection
        friends.add(sally);
        friends.add(frank);
        friends.add(billy);
    }
}
```

CSE1030 19

```
System.out.println("I have " + friends.size()
                    + " friends");
System.out.println("Here they are:");
for(Person p : friends)
    System.out.println("    " + p.getName());
}
```

CSE1030 20

Output

```
> java set
I have 3 friends
Here they are:
  Sally
  Frank
  Billy
```

CSEI030 21

How Does Iteration Work?

- The class (in this case, `HashSet`) can produce something called an **Iterator**
- An Iterator provides a way to iterate (loop through) all of the items in the set
- Lets do it “manually” to see how it works...

CSEI030 22

Overview Package **Class** Use Tree Deprecated Index Help

Prev Class Next Class Frames No Frames All Classes

Summary Nested Field Constr Method Detail Field Constr Method

java.util

Class HashSet<E>

java.lang.Object
java.util.AbstractCollection<E>
java.util.AbstractSet<E>
java.util.HashSet<E>

Type Parameters:

- `E` - the type of elements maintained by this set

All Implemented Interfaces:

- Serializable, Cloneable, Iterable<E>, Collection<E>, Set<E>

Direct Known Subclasses:

- JobStateReasons, LinkedHashSet

```
public class HashSet<E>
  extends AbstractSet<E>
  implements Set<E>, Cloneable, Serializable
```

This class implements the `set` interface, backed by a hash table (actually a `HashMap` instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee

CSEI030 23

Method Detail

iterator

```
public Iterator<E> iterator()
```

Returns an iterator over the elements in this set. The elements are returned in no particular order.

Specified by:

iterator in interface `Iterable<E>`

Specified by:

iterator in interface `Collection<E>`

Specified by:

iterator in interface `Set<E>`

Specified by:

iterator in class `AbstractCollection<E>`

Returns:

an iterator over the elements in this set

See Also:

`ConcurrentModificationException`

CSEI030 24

Overview Package **Classes** Use Tree Deprecated Index Help Java™ Platform Standard Ed. 7

Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method | Detail: Field | Constr | Method

java.util

Interface Iterator<E>

Type Parameters:

- `E` - the type of elements returned by this iterator

All Known Subinterfaces:

- ListIterator<E>, XMLEventReader

All Known Implementing Classes:

- BeanContextSupport, BCIterator, EventReaderDelegate, Scanner

```
public interface Iterator<E>
```

An iterator over a collection. `Iterator` takes the place of `Enumeration` in the Java Collections Framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

This interface is a member of the Java Collections Framework.

Since: 1.2

SE1030 25

Method Summary

Modifier and Type	Method and Description
boolean	<code>hasNext()</code> Returns <code>true</code> if the iteration has more elements.
E	<code>next()</code> Returns the next element in the iteration.
void	<code>remove()</code> Removes from the underlying collection the last element returned by this iterator (optional operation).

Method Detail

hasNext

```
boolean hasNext()
```

Returns `true` if the iteration has more elements. (In other words, returns `true` if `next()` would return an element rather than throwing an exception.)

Returns:

- `true` if the iteration has more elements

next

SE1030 26

```
import java.util.*;

public class set
{
    public static void main(String[] args)
    {
        // create a set to store my friends
        HashSet<Person> friends = new HashSet<Person>();

        // create some friends
        Person sally = new Person("Sally", 32);
        Person frank = new Person("Frank", 44);
        Person billy = new Person("Billy", 36);

        // add them to my collection
        friends.add(sally);
        friends.add(frank);
        friends.add(billy);
    }
}
```

CSE1030 27

```
System.out.println("I have " + friends.size()
                  + " friends");
System.out.println("Here they are:");
Iterator<Person> it = friends.iterator();
while(it.hasNext())
    System.out.println("    " + it.next().getName());
}
```

CSE1030 28

Comparison

```
System.out.println("I have " + friends.size()
                  + " friends");
System.out.println("Here they are:");
for(Person p : friends)
    System.out.println("    " + p.getName());
}
```

```
System.out.println("I have " + friends.size()
                  + " friends");
System.out.println("Here they are:");
Iterator<Person> it = friends.iterator();
while(it.hasNext())
    System.out.println("    " + it.next().getName());
}
```

CSE1030 29

Only Call `.next()` Once per Object

```
System.out.println("I have " + friends.size()
                  + " friends");
System.out.println("Here they are:");
for(Person p : friends) {
    System.out.println("    " + p.getName());
    System.out.println("    " + p.getAge());
}
```

```
System.out.println("I have " + friends.size()
                  + " friends");
System.out.println("Here they are:");
Iterator<Person> it = friends.iterator();
while(it.hasNext()) {
    Person = it.next();
    System.out.println("    " + p.getName());
    System.out.println("    " + p.getAge());
}
```

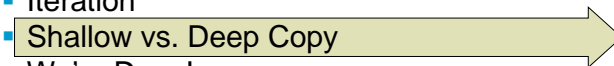
CSE1030 30

To Summarise Iterators

- They provide an easy way to access out data
- They are supported by all of the Java Collections
- The special “for-each” syntax makes them incredibly easy to use
 - Automatically retrieves the iterator
 - Reduces the amount of code we have to write

CSE1030 31

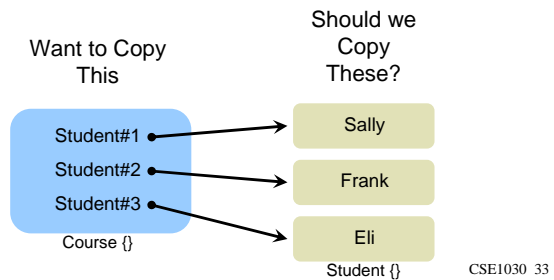
CSE1030 – Lecture #8

- Review: “is-a” versus “has-a”
- Theory: Composition versus Aggregation
- Iteration
- Shallow vs. Deep Copy 
- We’re Done!

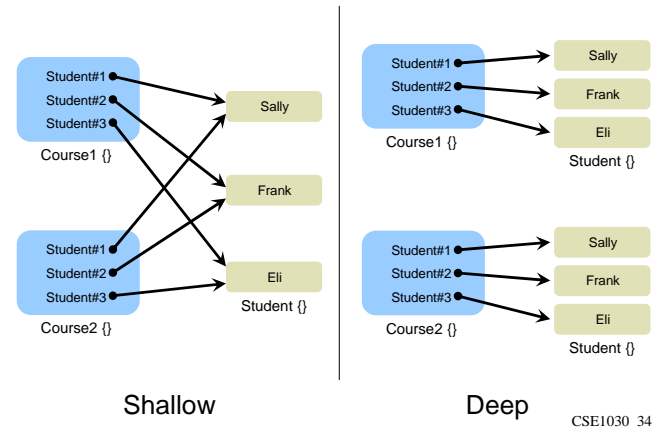
CSE1030 32

Shallow versus Deep Copy

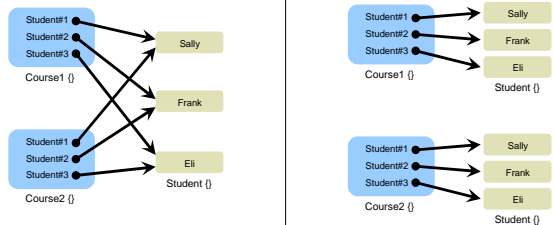
- If you are copying an object {course} that has aggregated other objects {student}, should you copy the aggregated objects {students} too?



Shallow versus Deep Copy



Shallow versus Deep Copy



- | | |
|--|---|
| <ul style="list-style-type: none"> Faster Uses Less Memory Aggregation Privacy Leak? | <ul style="list-style-type: none"> Slower Uses More Memory Composition Protects the Data? |
|--|---|

CSE1030 35

```
import java.util.*;

public class Course
{
    HashSet<Person> students = new HashSet<Person>();

    // constructor
    public Course() { students = new HashSet<Person>(); }

    // copy constructor
    public Course(Course course)
    {
        students = new HashSet<Person>();
        for(Person p : course.students)
        {
            students.add(p); ← Shallow Copy
        }
    }

    // add
    public void add(Person p) { students.add(p); }
}
```

CSE1030 36

```

public static void main(String[] args)
{
    Course course = new Course();

    // create some students
    Person sally = new Person("Sally Single", 32);
    Person frank = new Person("Frank", 44);
    Person billy = new Person("Billy", 36);

    // add them to my collection
    course.add(sally);
    course.add(frank);
    course.add(billy);
}

```

CSE1030 37

```

// make a copy
Course backup = new Course(course);

// Sally gets married and changes her name...
sally.setName("Sally Married");

System.out.println("Class List:");
for(Person p : course.students)
    System.out.println("  " + p.getName());

System.out.println("Backup Class List:");
for(Person p : backup.students)
    System.out.println("  " + p.getName());
}

```

CSE1030 38

Output – Shallow Copy

```

Class List:
  Billy
  Sally Married
  Frank
Backup Class List:
  Billy
  Sally Married
  Frank

```

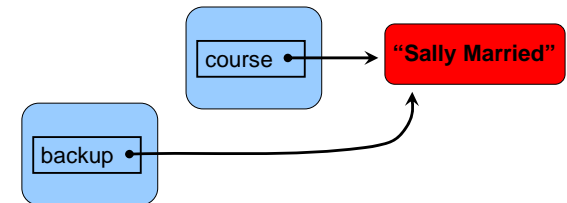
CSE1030 39

Output – Why?

```

Class List:
  Billy
  Frank
  Sally Married

```



CSE1030 40

```

// copy constructor
public Course(Course course)
{
    this();
    for(Person p : course.students)
        students.add(new Person (p));
}

```

Deep Copy

CSE1030 41

Output – Deep Copy

```

Class List:
  Billy
  Sally Married
  Frank
Backup Class List:
  Sally Single
  Billy
  Frank

```

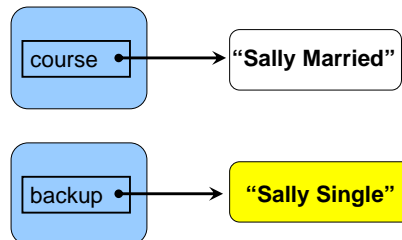
CSE1030 42

Output – Why?

```

Class List:
  Billy
  Frank
  Sally Single

```



CSE1030 43

Shallow vs. Deep Summary

- The “Shallow versus Deep” issue is very similar to a Privacy Leak and it also relates to Aggregation / Composition
 - **If you own the data**, you want to ensure it doesn't get changed without you knowing about it
 - **If you are using the data**, you probably want to use the latest (most accurate) data available
 - Be aware of the issues, and decide accordingly, by following the Inherent Relationships in the data

CSE1030 44

CSE1030 – Lecture #8

- Review: “is-a” versus “has-a”
- Theory: Composition versus Aggregation
- Iteration
- Shallow vs. Deep Copy
- We're Done!

CSE1030 45

Next topic...

Inheritance I

CSE1030 46