Labs

- Today is a review lab
  - Look over any of the labs, finish last Wednesday's if you haven't yet
- Wednesday is a demo lab
  - You will demo your music player GUI
  - Sign up for a slot if you haven't yet
- Since you will be doing the demo in the lab with a TA, today's lab is a good chance to make sure that your GUI works on a lab machine

Final Exam

- The final exam is this Friday, 9:00 AM in DMP 110 (usual time, usual location)
- Let me know immediately if you have a conflict (and a valid reason) and I will arrange for you to sit the alternate exam (probably Thursday)

Object Serialization

- Suppose you are writing a program that allows the user to store the names, telephone numbers and addresses of their contacts.
- When the user enters data, they expect it will be available to them the next time they run the program.
- To do this, the program needs to store the data (likely on a hard disk) from one session to the next.
- You can do this easily using object serialization.

Saving Objects

- There are actually two approaches we could take
- If you want to save the data of an object so that it can be used by other programs, you can just write to a plain text file, writing the value of each instance variable for that object in some sort of consistent format
- We now know how to write text to a file
- This data could then be used by a spreadsheet, database or other program

Saving Objects

- But if we want to be able to save an object and reload it in Java at a later date, it is much easier to use serialization

Serialization: Object Streams

- Java's serialization API supports the saving of the state of an object to a sequence of bytes; those bytes can later be used to restore the object
- The ability to save an object is sometimes called "persistent objects"
- Serialization makes it possible to save an object, stop the program, restart it, and then restore the object
- To make objects of a class serializable, you just need to implement the `Serializable` interface. `Serializable` is a **marker interface** (that means it has no methods)
  - E.g., to make the Account class serializable:
    `class Account implements Serializable {`
    `… }`

Saving an Object

- To save serializable objects in a file we need to
  - associate a `FileOutputStream` with the file
  - wrap an `ObjectOutputStream` around it
  - use `writeObject()` to store the objects sequentially
  - e.g., if `Account` implements Serializable and `a1, a2` are accounts we can save them in the file named ***account.dat***

```
ObjectOutputStream out = new ObjectOutputStream(
      new FileOutputStream("account.dat"));
out.writeObject(a1);
out.writeObject(a2);
```

- Most Java library classes are serializable (but not all)
  - Java takes care of serializing the variables in the class, etc.

Saving an Object

- Notice another example of chaining
- ObjectOutputStream is chained to FileOutputStream
- FileOutputStream knows how to connect to (and create) a file
- ObjectOutputStream lets us write objects, but it can't directly connect to a file

Chaining

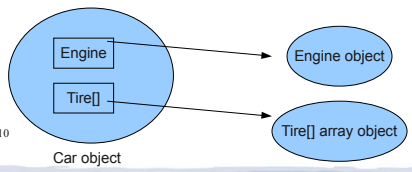
- By the way, why don't we just have a single stream that does exactly what we want?
- For one, it's good OO to have these specialized classes. Each class does one thing well.
- It also gives us a lot of flexibility as far as combining connection and processing streams.

What gets saved?

- All of an object's instance variables get saved
  - Why don't we save methods as well? What makes an object unique?
- This seems straightforward in the case of primitive values like 37 and 70, but what about instance variables that are object references?
- For example, what if our Car object has instance variables that refer to an Engine object and a Tire[] array?

What gets saved?

- Serialization saves the entire object graph. All objects referenced by instance variables, starting with the object being serialized
- So when you save the Car object, the Engine object and Tire objects are also saved



Serialization is “all or nothing”

- Either the entire object graph is serialized correctly or serialization fails
- This poses a problem:
  - We can't serialize an object if it contains an object reference for a class that is not serializable
  - For example, if we design a class Pond that implements Serializable, but it contains objects of the class Duck and Duck does not implement Serializable, then we will get an exception when we try to serialize Pond

Serialization is “all or nothing”

- But what if someone else designed the Duck class and it's not possible for us to make it Serializable?
- One option is to mark it as `transient`
- Anything marked as transient will be skipped over during the serialization process
  - `transient String currentID;`

Transient

- If we mark some instance variables as transient, what happens when we bring the object back to life (deserialize it)?
- Those instance variables will be brought back as `null` (primitives are brought back w/ default values)
- Your options then are to
  - reinitialize that null instance variable back to some default state
  - Or, if it's important that it have the same key values that it had before, then save those values so that you can create a new instance variable that's identical to the original, e.g. a new Duck with the same colour and size

Serialization is “all or nothing”

- Another option is to subclass the non-serializable class and make that subclass implement Serializable

Saving Objects

- If you try to save an object multiple times, the object will only get written once during serialization but there can be multiple references that will be resolved during deserialization

Reading Objects from a File

- To read back in the objects we have saved in a file we need to
  - associate a `FileInputStream` with that file
  - wrap an `ObjectInputStream` around it
  - use `readObject()` to get the objects sequentially, in the order they were saved
  - e.g., to get the first two accounts stored in ***account.dat***

```
ObjectInputStream in =
   new ObjectInputStream(
      new FileInputStream("account.dat"));
Account a1 =  (Account) in.readObject();
Account a2 =  (Account) in.readObject();
```


Reading Objects from a File

- If you try to read back more objects than you wrote, you'll get an exception
- The return type of readObject() is Object, so you need to cast it back to the type you know it really is
- A new object is given space on the heap, but the serialized object's constructor does not run
  - Why not? What might happen to its values?

Reading Objects from a File

- However, if the object has a non-serializable class somewhere up its inheritance tree, the constructor for that non-serializable class will run along with any constructors above that (even if they're serializable)

Reading Objects from a File

- Java needs to be able to find the Class of the objects you are reading in
  - Remember, the class itself did not get saved, just the objects
- If you change the definition of the class in between saving an object and reading it back (it could be days or weeks or years before you read it back!), a `java.io.InvalidClassException` may be thrown because the version of the class is not compatible with the class of the saved object

Ponds and Frogs

```
public class Frog {
private String name;
private int age;

public Frog(String name, int age)
{
this.name = name;
this.age = age;
}

public Frog()
{
this.name = "kermit";
this.age = 1;
}
```

We have a Frog class with a couple of constructors. Notice that the no-arg constructor sets the name and age of the Frog to some defaults: “kermit” and 1

Frog, cont'd

```
public String getName()
{
return name;
}

public int getAge()
{
return age;
}
}
```

Our Frog class also has some accessor methods for the name and age attributes.

BullFrog

```
public class BullFrog extends Frog
{

public BullFrog(String name, int age)
{
super(name, age);
}

// other code omitted -
// extends Frog in some way
}
```

BullFrog extends Frog. Notice that neither BullFrog nor Frog are Serializable.

Pond

```
import java.io.*;
public class Pond implements Serializable
{
    private BullFrog aFrog;
    private int pondDepth;

    public Pond( BullFrog aFrog, int depth )
    {
        this.aFrog = aFrog;
        pondDepth = depth;
    }

    public BullFrog getFrog()
    {
        return aFrog;
    }
```

We have a Pond class that contains a reference to a BullFrog object. Notice that the pond *is* serializable.

Pond, cont'd

```
public int getDepth()
    {
        return pondDepth;
    }
```


Creating Ponds and Frogs

```
public static void main( String[] args )
    {
   Pond littlePond = new Pond( new BullFrog( "Henry", 10 ), 4 );
```

We create a new Pond object, which contains a Duck object (named Henry, 10 years old) and has a depth of 4 meters.

Serializing Pond

```
public static void main( String[] args )
    {
   Pond littlePond = new Pond( new BullFrog( "Henry", 10 ), 4 );
   try
        {
            ObjectOutputStream out = new ObjectOutputStream(
                    new FileOutputStream( "pond.dat" ) );
            out.writeObject( littlePond );
            out.close();
        }
   catch( Exception ex )
        {
            ex.printStackTrace();
        }
```

Let's try to serialize the Pond. What will happen?

Serializing Pond

```
java.io.NotSerializableException: BullFrog
      at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1156)
      at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1509)
      at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1474)
      at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1392)
      at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1150)
      at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:326)
      at Pond.main(Pond.java:32)
```

Even though Pond is serializable, BullFrog is not. So we get an exception telling us as much. We have a couple of options for correcting this.

Serializing Pond

- We could mark BullFrog as `transient`
- Anything marked as transient will be skipped during serialization
- After we deserialize (bring the Pond object back to life) the BullFrog will be returned as `null`
- At that point we could initialize a new BullFrog object

Pond

```
import java.io.*;
public class Pond implements Serializable
{
    private transient BullFrog aFrog;
    private int pondDepth;

    . . .

}
```

So we go back and make BullFrog transient. We can go ahead and serialize the Pond object now. We won't get an exception. But what happens when we deserialize?

Deserializing

```
try
        {
        ObjectInputStream in = new ObjectInputStream( new
FileInputStream("pond.dat" ) );
        Pond myPond = (Pond) in.readObject();
        in.close();
        BullFrog myFrog = myPond.getFrog();
        System.out.println(myFrog == null);
        }
catch( Exception ex )
        {
        ex.printStackTrace();
        }
```


Deserializing

```
try
        {
        ObjectInputStream in = new ObjectInputStream( new
FileInputStream("pond.dat" ) );
        Pond myPond = (Pond) in.readObject();
        in.close();
        BullFrog myFrog = myPond.getFrog();
        System.out.println(myFrog == null);
        }
catch( Exception ex )
        {
        ex.printStackTrace();
        }
```

true

Deserializing

- Since BullFrog had a non-serializable superclass Frog, you may have expected that when Pond was deserialized, a new BullFrog object would be created and the superclass constructor would run, giving the BullFrog the name “kermit” and the age 1
- This doesn't happen, because any object that is skipped during serialization (marked as transient) is simply null after deserialization

Serializable BullFrogs

- But let's say BullFrog *was* Serializable and Frog was *not* Serializable

```
import java.io.Serializable;


public class BullFrog extends Frog implements Serializable
{
public BullFrog(String name, int age)
{
super(name, age);
}
// other code omitted – extends Frog in some way
}
```


Pond

```
import java.io.*;
public class Pond implements Serializable
{
    private BullFrog aFrog;
    private int pondDepth;

    . . .
}
```

We no longer have to mark BullFrog as transient. But what happens when we deserialize?

What gets printed?

```
try   {
    ObjectInputStream in = new ObjectInputStream( new
FileInputStream("pond.dat" ) );
    Pond myPond = (Pond) in.readObject();
    in.close();
    BullFrog myFrog = myPond.getFrog();
    System.out.println( "My frog's name is " +
myFrog.getName() );
    System.out.println( "He is " + myFrog.getAge() + " years
old" );
    System.out.println( "He lives in a pond that is "+
myPond.getDepth() + " feet deep" );
      }
 catch( Exception ex ){ex.printStackTrace();}
```

Deserialization

My frog's name is kermit

He is 1 years old

He lives in a pond that is 4 feet deep

- During deserialization, Java sees that BullFrog has a non-serializable superclass Frog, and runs the no-arg constructor for that superclass
- The BullFrog then ends up with the name Kermit and age of 1, even though previously it was named Henry and was 10 years old

Frog

- What if we make Frog serializable too?

```
public class Frog implements Serializable
```

- Now both Frog and BullFrog are Serializable
- Now what happens when we serialize, deserialize, and print out the attributes of the BullFrog?

My frog's name is Henry

He is 10 years old

He lives in a pond that is 4 feet deep

- Since the Frog class is now serializable, its constructor never runs and the BullFrog ends up with the same attributes it had before serialization

Frog and BullFrog

- By the way, what would happen if only Frog implemented Serializable and BullFrog did not?
- If we try to serialize Pond without making BullFrog also implement Serializable, will we get an exception again?

Frog and BullFrog

- No – BullFrog is still of type Serializable because it inherits from Frog

Subclassing

- If we didn't define Frog or BullFrog (someone else did) and neither are Serializable, is our only option to make them transient when we serialize a Pond object?
- No, we could also subclass BullFrog and make that subclass serializable
- Or we could keep them as transient and save critical attribute values and use those to create new objects after deserialization

In-Class Exercise I

```
public class Frog implements Serializable
{

private String name;
private transient int age;

. . .

}
```

Say we have Frog defined along these lines...

In-Class Exercise I

```
public class FrogSerialTester{
 public static void main(String[] args){
  Frog frank = new Frog("Frank", 3);
  Frog flo = new Frog("Flo", 2);
  Frog fran = new Frog("Fran", 5);
  try {
    ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream( "frogs.dat" ) );
    out.writeObject(frank);
    out.writeObject(flo);
    out.writeObject(fran);
    out.close();
       }
  catch( Exception ex ){ex.printStackTrace();}
 } }
```

Write the code to read these Frog objects back in, and print out the name and age of each Frog. Indicate what the output would be.

More Threads

```
public class MyRunnable implements Runnable {
public void run()
{
go();
}
public void go()
{
doMore();
}
public void doMore()
{
System.out.println("top o' the stack");
}
```

Remember this example? We had a class implementing Runnable, and the run() method just calls a couple other methods and something gets printed out.

More Threads

```
public class ThreadTester {


public static void main(String[] args)
{
Runnable threadJob = new MyRunner();
Thread myThread = new Thread(threadJob);
myThread.start();
```

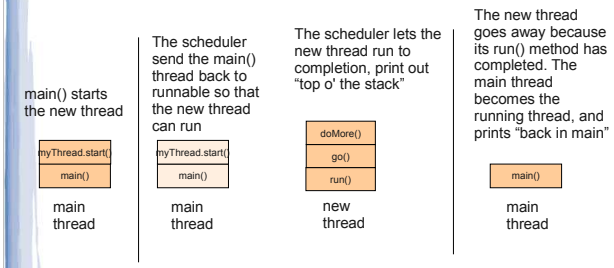
We then passed an instance of that Runnable job to a new Thread instance, and started the Thread. We also put a print statement here.

```
System.out.println("back in main");
}
}
```

We mentioned in the lecture that the order of the print statements will vary. Sometimes "top o' the stack" prints first and then "back in main" and sometimes the other way around.

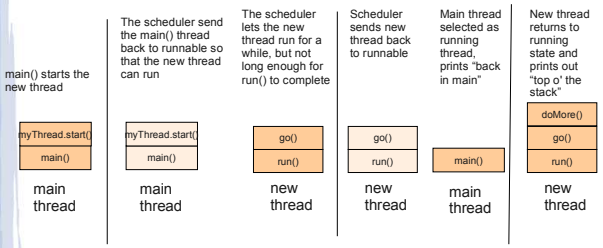
Why does it vary?

- Sometimes it runs like this:



Why does it vary?

- And sometimes it runs like this:



More Threads

```
public class ThreadTester {


public static void main(String[] args)
{
Runnable threadJob = new MyRunner();
Thread myThread = new Thread(threadJob);
myThread.start();
```

But you may have gone home and tried this and found that it always prints in the same order. So what's up?

```
System.out.println("back in main");
}
}
```

It depends on your scheduler (preemptive or non-preemptive). Your scheduler might allow the main thread always to run to completion before running the new thread. Or it might not.

More Threads

```
public class ThreadTester {


public static void main(String[] args)
{
Runnable threadJob = new MyRunner();
Thread myThread = new Thread(threadJob);
myThread.start();

System.out.println("back in main");
}
}
```

The important thing is that we can't assume what type of scheduler it is. It's very easy to make yourself think that the ordering is always going to be one way because that's always how it is on your computer. Don't make that assumption!

sleep()

```
public void go()
{
try {
Thread.sleep(2000);
}
catch(InterruptedException ex)
{
ex.printStackTrace();
}


doMore();
}
```

We found that making this change influences the order of the printing. The thread goes to sleep for a short while, and during that time it is waiting/blocked and other threads have a chance to be selected.

yield()

- Another option was to use yield()
- sleep() and yield() behave differently, though
- sleep(t) guarantees that the thread will not resume running for at least time t, even if there are no other threads to select
- yield() puts thread back in Runnable state, gives threads with equal priority a chance to run

yield()

```
public void go()
{
Thread.yield();
doMore();
}
```

join()

- If thread t calls r.join() on thread r, then t will be waiting/blocked until r finishes


```
public class MyRunnable
implements Runnable {

public void run(){

go();

}


public void go(){

Thread.yield();

doMore();

}


public void doMore(){

System.out.println("top o' the
stack");

}
}
```

```
public class ThreadTester {


public static void main(String[]
args)

{

Runnable threadJob = new
MyRunnable();

Thread myThread = new
Thread(threadJob);

myThread.start();

try {

myThread.join();

}

catch (Exception ex){}

System.out.println("back in main");

}
}
```

What gets printed?

In-Class Exercise II

```
public class MyRunnable
implements Runnable {


public void run()

{

System.out.println("in the new
thread");

}

}
```

How can we get it to print this?
first in main
in the new thread
second in main
List all the ways you can think of


```
public class ThreadTester {


public static void main(String[]
args){

Runnable threadJob = new
MyRunnable();

Thread myThread = new
Thread(threadJob);

myThread.start();


System.out.println("first in
main");

System.out.println("second in
main");

}

}
```


Implementing Associations

Learning Objectives:

- explain the similarites and differences between how associations map to object-oriented (Java) code
- write code that implements unidirectional, bidirectional, 1-1 and 1-many associations

Unidirectional one-to-one associations

- The simplest type of association to implement is a unidirectional one-to-one association between two classes
- With a unidirectional association, you can navigate from an object of one class to an object of the other class (as indicated by the direction of the arrow) but not vice-versa
- This kind of association is easily implemented using an attribute that holds a reference to an object of the other class

Unidirectional one-to-one cont'd

- Consider the following association:

- Implementation:

```
public class Watch
{
   private Display theDisplay;
   ...
}
```


Bidirectional one-to-one associations

- The following UML diagram indicates a bidirectional association
  - no arrows on any end of the association

- We must be able to navigate from an account to the corresponding customer and vice versa
- Implementation:
  - each class needs an attribute that holds a reference to an object of the other class
  - each class must have setter methods that allow the reference to the object of the other class be established

A Bad Implementation

```
public class Customer
{
  private Account theAccount;
  public void setAccount(Account account)
  {
    theAccount = account;
  }
  // etc.
}
public class Account
{
  private Customer theCustomer;
  public void setCustomer(Customer customer)
  {
    theCustomer = customer;
  }
  // etc.
}
```

How about this?

```
public class Customer
{
  private Account theAccount;
  public void setAccount(Account account)
  {
    theAccount = account;
    account.setCustomer(this)
  }
}
public class Account
{
  private Customer theCustomer;
  public void setCustomer(Customer customer)
  {
    theCustomer = customer;
    customer.setAccount(this)
  }
}
```

Infinite Recursion

- When we first saw this example a couple of weeks ago, we hadn't yet discussed recursion
- In the recursion lecture, we talked about *indirect* recursion
- This is an example of indirect recursion that results in an infinite recursion
- Why is it infinite? Which key component of recursion is missing here?

A Better Solution

```
public class Customer
{
  private Account theAccount;
  public void setAccount(Account account)
  {
    if (theAccount != account) {
      theAccount = account;
      account.setCustomer(this)
    }
  }
}
public class Account
{
  private Customer theCustomer;
  public void setCustomer(Customer customer)
  {
    if (theCustomer != customer) {
      theCustomer = customer;
      customer.setAccount(this)
    }
  }
}
```


Another 1-to-1 Association (Bikes and Riders)

```
public class Bicycle {

private Cyclist rider;

private String brand;


public Bicycle(String brand){

this.brand = brand;

}

public Cyclist getRider(){

return rider;

}

public String getBrand(){

return brand;

}
}
```

Bicycle, cont'd

```
public void setRider(Cyclist newRider)
{
???
}

}
```


Bicycle, cont'd

```
public void setRider(Cyclist newRider)
{
if (rider != newRider)
 {
  rider = newRider;
  newRider.setBike(this);
 }
}
```


Cyclist

```
public class Cyclist {
private Bicycle bike;
private String name;

public Cyclist(String name){
 this.name = name;
}
public Bicycle getBike(){
 return bike;
}

public String getName(){
 return name;
}
```


Cyclist, cont'd

```
public void setBike(Bicycle newBike)
{
???
}

}
```

Cyclist, cont'd

```
public void setBike(Bicycle newBike)
{
if (bike != newBike)
 {
 bike = newBike;
 newBike.setRider(this);
 }
}

}
```


Cycle/Rider Tester

```
public class BikesAndRiders
{
public static void main(String[] args)
{
Cyclist rider1= new Cyclist("Steve");
Bicycle bike1 = new Bicycle("Giant");
rider1.setBike(bike1);
System.out.println(rider1.getName()+ " rides a
"+rider1.getBike().getBrand());
System.out.println(bike1.getBrand()+" is ridden by
"+bike1.getRider().getName());
}
}
```


Consistency

```
public class BikesAndRiders
{
public static void main(String[] args)
{
Cyclist rider1= new Cyclist("Steve");
Bicycle bike1 = new Bicycle("Giant");
rider1.setBike(bike1);
System.out.println(rider1.getName()+ " rides a
"+rider1.getBike().getBrand());
System.out.println(bike1.getBrand()+" is ridden by
"+bike1.getRider().getName());
}
}
```

Steve rides a Giant
Giant is ridden by Steve

Recursion Redux

```
public void setBike(Bicycle newBike)
{
if (bike != newBike)
 {
 bike = newBike;
 newBike.setRider(this);
 }
}

}
```

Even with the check we are doing, there is still a way to end up with infinite recursion. What if both setBike() and setRider() had their method calls first, i.e. switching these two lines...

```
public void setBike(Bicycle newBike)
{
if (bike != newBike)
 {
 newBike.setRider(this);
 bike = newBike;
  }
}
public void setRider(Cyclist newRider)
{
if (rider != newRider)
 {
  newRider.setBike(this);
  rider = newRider;
  }
}
```

Trace through the code – how do we end up with infinite recursion?

In-Class Exercise III

- Let's say that, against our better judgment, we defined these methods in the initial naïve way

```
public void setBike(Bicycle newBike)
{
 bike = newBike;
}
public void setRider(Cyclist newRider)
{
 rider = newRider;
 }
}
```


What would be our output here?

```
public class BikesAndRiders {
  public static void main(String[] args){
    Cyclist rider1= new Cyclist("Steve");
    Bicycle bike1 = new Bicycle("Giant");
    rider1.setBike(bike1);
    Cyclist rider2 = new Cyclist("Svein");
    Bicycle bike2 = new Bicycle("Cerevelo");
    bike2.setRider(rider2);
    bike1.setRider(rider2);
    rider1.setBike(bike2);
    System.out.println(rider1.getName()+ " rides a
"+rider1.getBike().getBrand());
    System.out.println(bike2.getBrand()+" is ridden by
"+bike2.getRider().getName());
 }
}
```


One-to-many Associations

- One-to-many associations can also be bidirectional:

or unidirectional:

depending on the needs of the application. In either case the "many" part of the association is realized using a collection of references.

One-to-many Associations cont'd

```
public class Customer
{
    private Set<Video> rentedVideos;

    public void addVideo(Video video)
    {
      …
    }
   // etc.
}
```

- The particular type of collection that is used will depend on the needs of the application.
- If ordering matters, we may use an `Array` or `List`.
- If an element appears in the collection only once, we may choose to use a `Set`, etc.

One-to-many Associations cont'd

- Assuming a bidirectional association between customer and Video, the implementation of the Video class would look something like:

```
public class Video
{
    private Customer rentee;

    public void setRentee(Customer c)
    {
      …
    }
}
```


One-to-many associations cont'd

- Again, we have to be careful to ensure consistency. Would this be ok?

```
public class Customer {
     //…
    public void addVideo(Video video)
    {
       rentedVideos.add(video);
       video.setRentee(this);
    }
}

public class Video {
     //…
    public void setRentee(Customer c)
    {
       rentee = c;
      rentee.addVideo(this);
    }
}
```


One-to-many associations cont'd

- Here's a better implementation …

```
public class Customer {
     //…
    public void addVideo(Video video)
    {
        if (rentedVideos.add(video)){
            video.setRentee(this);
        }
    }
}

public class Video {
     //…
    public void setRentee(Customer c)
    {
        if (rentee != c)
        {
            rentee = c;
            rentee.addVideo(this);
        }
    }
```

One-to-many, cont'd

- Maybe we decide that Cyclists and Bicycles are many-to-one...

Cyclist

```
public class Cyclist
{
private Set<Bicycle> bikes;
private String name;
// . . .
public void addBike(Bicycle newBike)
 {
 if (bikes.add(newBike))
  {
   newBike.setRider(this);
  }
 }
// . . .
}
```

A Cyclist now has a Set of bikes rather than a single bike

Bicycle

```
public class Bicycle {

private Cyclist rider;
private String brand;
// . . .

public void setRider(Cyclist newRider)
{
if (rider != newRider)
{
rider = newRider;
newRider.addBike(this);

}
}
```

A Bicycle is associated with one Cyclist.

Notice that rider has been set to newRider, but the Bicycle object hasn't been removed from the previous rider's bike Set. We should probably remove it.

Bicycle

```
public class Bicycle {

private Cyclist rider;
private String brand;
// . . .

public void setRider(Cyclist newRider)
{
if (rider != newRider)
{
if (rider != null) {rider.removeBike(this);}
rider = newRider;
newRider.addBike(this);

}
}
```

We will add a removeBike() method to the Cyclist class.

Cyclist, Updated

```
public class Cyclist {
// . . .
public void removeBike(Bicycle bike)
{
if (bikes.contains(bike))
{
bikes.remove(bike);
}
}
// . . .
}
```



```
Cyclist rider1= new Cyclist("Steve");
Bicycle bike1 = new Bicycle("Giant");
Bicycle bike2 = new Bicycle("Cerevelo");

rider1.addBike(bike1);
rider1.addBike(bike2);

Cyclist rider2 = new Cyclist("Svein");
rider2.addBike(bike2);

for (Bicycle b: rider1.getBikes()){
System.out.println(rider1.getName()+ " rides a "+b.getBrand());
}
for (Bicycle b2: rider2.getBikes()){
System.out.println(rider2.getName()+" rides a "+b2.getBrand());
}
```



```
Cyclist rider1= new Cyclist("Steve");
Bicycle bike1 = new Bicycle("Giant");
Bicycle bike2 = new Bicycle("Cerevelo");

rider1.addBike(bike1);
rider1.addBike(bike2);

Cyclist rider2 = new Cyclist("Svein");
rider2.addBike(bike2);
```

These getter methods return Sets of bikes.

```
for (Bicycle b: rider1.getBikes()){
System.out.println(rider1.getName()+ " rides a "+b.getBrand());
}
for (Bicycle b2: rider2.getBikes()){
System.out.println(rider2.getName()+" rides a "+b2.getBrand());
}
```



```
Cyclist rider1= new Cyclist("Steve");
Bicycle bike1 = new Bicycle("Giant");
Bicycle bike2 = new Bicycle("Cerevelo");

rider1.addBike(bike1);
rider1.addBike(bike2);

Cyclist rider2 = new Cyclist("Svein");
rider2.addBike(bike2);

for (Bicycle b: rider1.getBikes()){
System.out.println(rider1.getName()+ " rides a "+b.getBrand());
}
for (Bicycle b2: rider2.getBikes()){
System.out.println(rider2.getName()+" rides a "+b2.getBrand());
}
```

What's our output?

Many-to Many Associations

- Consider the following many-to-many association between `SalesRep` and `Customer`:

- One way of implementing it is for both classes to maintain collections of references to instances of the other class.
- Again, operations need to be added that preserve consistency between the two collections of references.

Aggregations and Compositions

- The implementation of an aggregation does not differ from an association
- The implementation of a composition should ensure that when the whole is deleted, the parts are also deleted
  - In Java, we need to make sure that when the whole is deleted, there are no references to its parts, so the parts are garbage collected.

Learning Goals Review

Learning Objectives:

- explain the similarites and differences between how associations map to object-oriented (Java) code
- write code that implements unidirectional, bidirectional, 1-1 and 1-many associations

Course Review

- When you complete this course, you will be able to:
  - move from personal software development methodologies to professional standards and practices
    - design software following standard principles and formalisms
    - create programs that interact with their environment (files etc.) and human users according to standard professional norms
    - develop effective software testing skills
  - given an API, write code that conforms to the API to perform a given task
  - identify and evaluate trade-offs in design and implementation decisions for systems of an intermediate size
  - read and write programs in Java using advanced features
    - collections, exceptions, etc.
  - extend your mental model of computation from that developed in CPSC111
    - recursion, concurrency, etc.
  - work with an existing codebase, including reading and understanding given code, and augment its functionality [in assignments]