Lambda Notes for CS 2102

Remember filter and map from CS1101/1102:

;; filter: (X->Boolean) ListOfX -> ListOfX

the function argument (X-> Boolean) is a predicate. Filter applies the predicate to each item in the list to determine whether or not to keep that item in the resulting list

;; map: (X->Y) ListOfX -> ListOfY

the general function (X->Y) is applied to each item in ListOfX to transform it into an element of ListOfY

Passing functions as arguments in Racket is easy (functions are first-class objects). It can be considerably more difficult to do this in other languages. Java 1.8 is a major step in making this easy to do in Java. The Java libraries now support methods filter and map that can be used similarly to the way we used those functions in Racket.

Today, we'll

- learn how to define a lambda expression in Java
- learn how to create a predicate (such as could be used to filter a collection)
- learn how to create a general function (such as could be used to map a collection)

In lab next week, you'll use what you learned today to be able to write code that uses the Java filter and map methods.

What is a lambda expression?

A lambda expression is an anonymous function; that is, it's a method without an associated name. A lambda expression is a nameless function, written exactly in the place that it's needed (often as an argument to another function).

Basic syntax of lambda expressions

(parameters) -> expression

OR

(parameters) -> { statements; }

Examples of lambda expressions

```
(int x, int y) -> x + y
```

This can be thought of as

```
int namelessMethod(int x, int y){
      return x + y;
}
```

```
(x, y) -> x + y
```

This one's more interesting, because it demonstrates that lambda expressions *execute in the context of their appearance*. In other words, the same lambda expression can be used to produce different results based on different parameter types. In the above expression, if x and y are int's, the effect of the lambda expression is to produce the sum of x and y. If x and y are String's the effect of the lambda expression is to concatenate the two Strings.

A few more examples:

() -> 99 // an expression that always produces the value 99

x-> x * 2 // doubles a number; ()'s can be omitted for a single, inferred parameter

// takes a collection (such as a LinkedList), removes all the elements from the list, and returns
// the original size of the list
// Here you can see that parentheses can be omitted for a single (inferred-type) parameter
// This example can actually work on any class that provides the methods size() and clear() (that
// context thing again)
// Use {..}'s when the expression consists of multiple statements

aList -> { int s=aList.size(); aList.clear(); return s;}

Using lambda expressions (with Predicate)

Let's use a lambda expression to create a predicate.

```
LinkedList<String> greetings = new LinkedList<String>();
greetings.addFirst("hello");
greetings.addFirst("hola");
greetings.addFirst("bonjour");
greetings.addFirst("aloha");

Predicate<String> p = (s)->s.length() > 4;

// count the number of strings that have more than 4 characters
int count = 0;
for (String s:  greetings){
      if (p.test(s)){
            count++;
      }
}
```

Java has an interface called Predicate<T> in java.util.function that provides a method called test( ) (show Java API). test( ) returns true if the input argument satisfies the predicate, otherwise it returns false. Predicate is a functional interface (an interface that provides a single method). The type of a lambda expression is a functional interface type. So Predicate can be used as the assignment target for a lambda expression:

```
Predicate<String> p = (s)->s.length() > 4;
```

More usually, a Predicate will be a parameter to a method. When the method is called, an appropriate lambda expression can be passed as the Predicate argument:

```
// returns the number of strings that satisfy the given Predicate
public int countStringsThatSatisfy(LinkedList<String> aList, Predicate p){
 int count = 0;
 for (String s:  aList){
      if (p.test(s)){
         count++;
     }
 }
 return count;
}
```

Here are some calls to the method:

int numThatSatisfy;

// count the number of strings in myList that have more than four characters
p = (s)-> s.length() > 4;
numThatSatisfy = countStringsThatSatisfy (myList, p);

// count the number of occurrences of the string “hello” in myList
p = s -> s.equals(“hello”);
numThatSatisfy = countStringsThatSatisfy (myList, p);

// count the number of occurrences of strings in myList that start with the letter “h”
p = s->s.startsWith(“h”);
numThatSatisfy = countStringsThatSatisfy (myList, p);

Using lambda expressions (with Function)

Java has a functional interface called `Function<T, R>` in `java.util.function`, where T is the type of input to the function and R is the type of the result of the function. Function provides a method called `apply()`. apply() applies this function to the given argument.

- create a lambda expression that will map a string to the number of characters in the string. Assign your lambda expression to a variable f, where the type of f is Function.
- create an int variable called totalChars and initialize it to zero.
- write an enhanced for loop that will cycle through the LinkedList called greetings. Each time through the loop, use apply() to add the number of characters in each string to totalChars.

```
Function<String, Integer> f = (s)-> s.length();
int totalChars = 0;
for (String s:  greetings){

      totalChars += f.apply(s);

}
```

```
// count strings that satisfy the given criteria
int countStrings (LinkedList<String> strList, Predicate p){
  int numStrings = 0;

  for (String s:  strList){
    if ( p.test(s) ){
       numStrings++;
    }
  }

  return numStrings;
}



LinkedList<Dillo> transformDillos (LinkedList<Dillo> dilloList, Function f){
  LinkedList<Dillo> newDilloList = new LinkedList<Dillo>();

  for (Dillo d:  dilloList){
     newDilloList.add( (Dillo) (f.apply(d)));
  }

  return newDilloList;
}
```