CS15 2014 TShirts!

- Congratulations to Emily Kasbohm, whose T-Shirt design won!

- We will set up a Teespring campaign, and email a link to the class if you would like to order one

Programming Languages

and why you should love them

Problem 2.1

Design a program called rainfall that consumes a list of real numbers representing daily rainfall readings. The list may contain the number -999 indicating the end of the data of interest. Produce the average of the non-negative values in the list up to the first -999 (if it shows up). There may be negative numbers other than -999 in the list (representing faulty readings). Assume that there is at least one non-negative number before -999.

Example:
rainfall takes in an ArrayList containing(1, -2, 5, -999, 8) and returns 3

Solution A

```
public class Rainfall1 {
  public double rainfall (ArrayList<Double> data) {
     double total = 0;
    int days = 0 ;

    for (double d : data) {
        if (-999 == d) {
                return (total / days) ;
        }
        else if (d >= 0) {
                total = total + d ;
                days = days + 1 ;
        }
    }
    return (total / days) ;
  }
}
```

Solution C

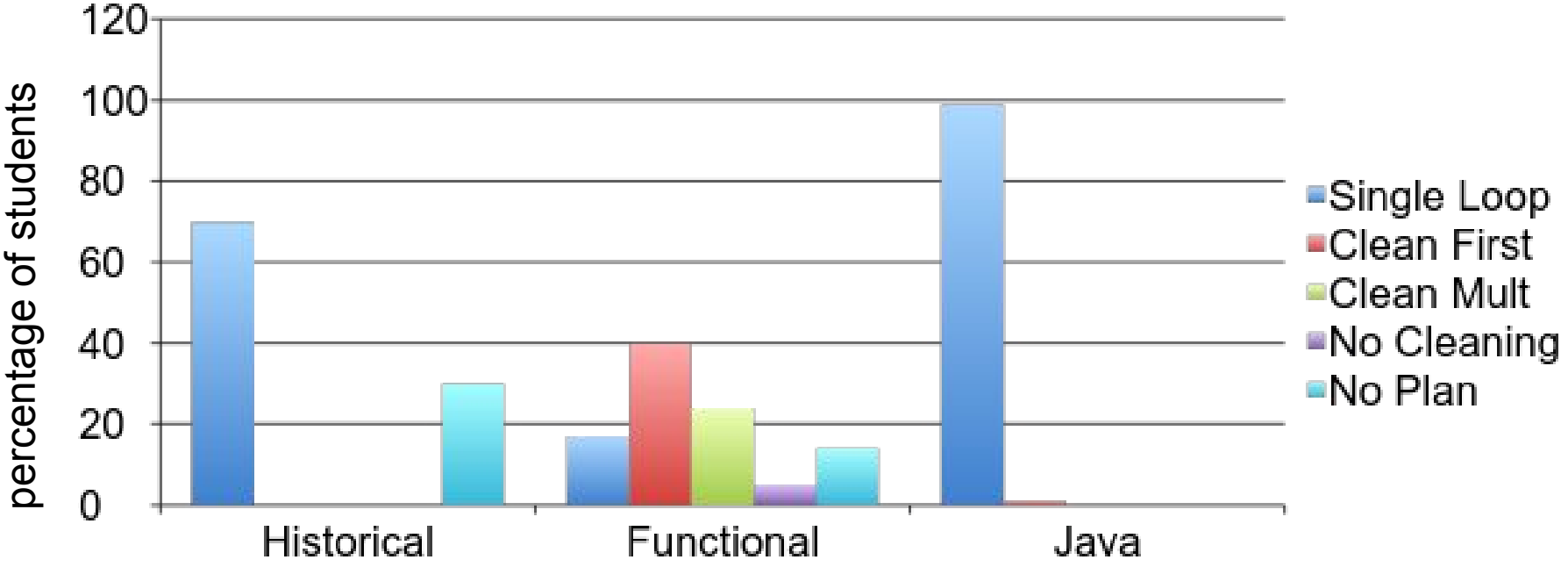
```
public class Rainfall3 {
  public ArrayList<Double> cleanData (ArrayList<Double> data){
   ArrayList<Double> cleanData = new ArrayList<Double>();

   for (double d : data) {
       if (-999 == d) {
             return(cleanData);
       }
       else if (d >= 0) {
             cleanData.add(d);
       }
   }
    return cleanData;
  }
```

```
public double sum(ArrayList<Double> data) {
    double total = 0;
    for (double d : data) {
        total = total + d;
    }
    return total;
}


public double rainfall (ArrayList<Double> data) {
    ArrayList<Double> relevant = cleanData(data);
    return(sum(relevant) / relevant.size());
  }
}
```

Stats breakdown:



Historical:	rough summary of data from literature since 1986 (all imperative programming)
Functional:	218 students over 5 courses, all had a 17-like curriculum
Java:	51 students from 1 course, had covered arrays but not lists

Why does this happen, and why do we care?

- The language you learn to program in influences how you approach solving problems

Bold claim, Ardra…..

- Different languages make some things easier and more efficient to do
- Experts deciding the “right” way to program in a language take into account things like compiler optimizations and stack space
- Also considered when building a language: make efficient features easier for a programmer to use
  - Built in functions become part of problem solving toolkit

Example: Recursion vs. Iteration

- Most Java programmers' instincts are to use iteration, while functional programmers rely very heavily on recursion
- Recursion is slower in Java than in functional languages (like Racket, Haskell, Scala) which optimize recursion to not use more stack space
- Structure and syntax of a language makes some of its features more efficient and accessible, which changes how you use it.
- It might not even have all features!

- You probably don’t know all this stuff about stack frames and recursion optimization, and I don’t expect you to
- Most new programmers don’t!
- The people deciding what is “good” vs. “bad” design do though, which trickles down into standard coding styles and design patterns
- People building a language also make computationally cheap things easier to do, and expensive ones harder

Functional Programming

- Focus on computation, doesn't allow mutation
- What does that mean?

```
int a = 5
a = 6
```

- NOT ALLOWED – can't store or change state
- The whole idea of a method that doesn't return something seems crazy. What else can it do??

- Because you're only ever creating new information (not changing it), the language can optimize certain features
- Allows it to store data in memory more efficiently
- Creation of data structures becomes very cheap. Conscious choice because that's something they want to encourage!
- Extreme case of language design encouraging a certain style because it won't let you use mutation
- Can't even use loop counters in the same way!

How does this affect you?

- Let’s go back to the problem from the homework

Solution A

```
public class Rainfall1 {
  public double rainfall (ArrayList<Double> data) {
    double total = 0;
    int days = 0 ;

    for (double d : data) {
        if (-999 == d) {
                return (total / days) ;
        }
        else if (d >= 0) {
                total = total + d ;
                days = days + 1 ;
        }
    }
    return (total / days) ;
  }
}
```

Solution A

- Single loop
- If we've reached the end of the list return
- Otherwise, if it's good information, add it to the total and increment the counter

Solution C

```
public class Rainfall3 {
  public ArrayList<Double> cleanData (ArrayList<Double> data){
   ArrayList<Double> cleanData = new ArrayList<Double>();

   for (double d : data) {
       if (-999 == d) {
             return(cleanData);
       }
       else if (d >= 0) {
             cleanData.add(d);
       }
   }
    return cleanData;
  }
```

```
public double sum(ArrayList<Double> data) {
    double total = 0;
    for (double d : data) {
        total = total + d;
    }
    return total;
}


public double rainfall (ArrayList<Double> data) {
    ArrayList<Double> relevant = cleanData(data);
    return(sum(relevant) / relevant.size());
  }
}
```

- Three steps:
  - Cleanse data
  - Sum data
  - Average it
- This and Solution B take similar approaches

- Students with functional programming backgrounds tend to produce this kind of solution

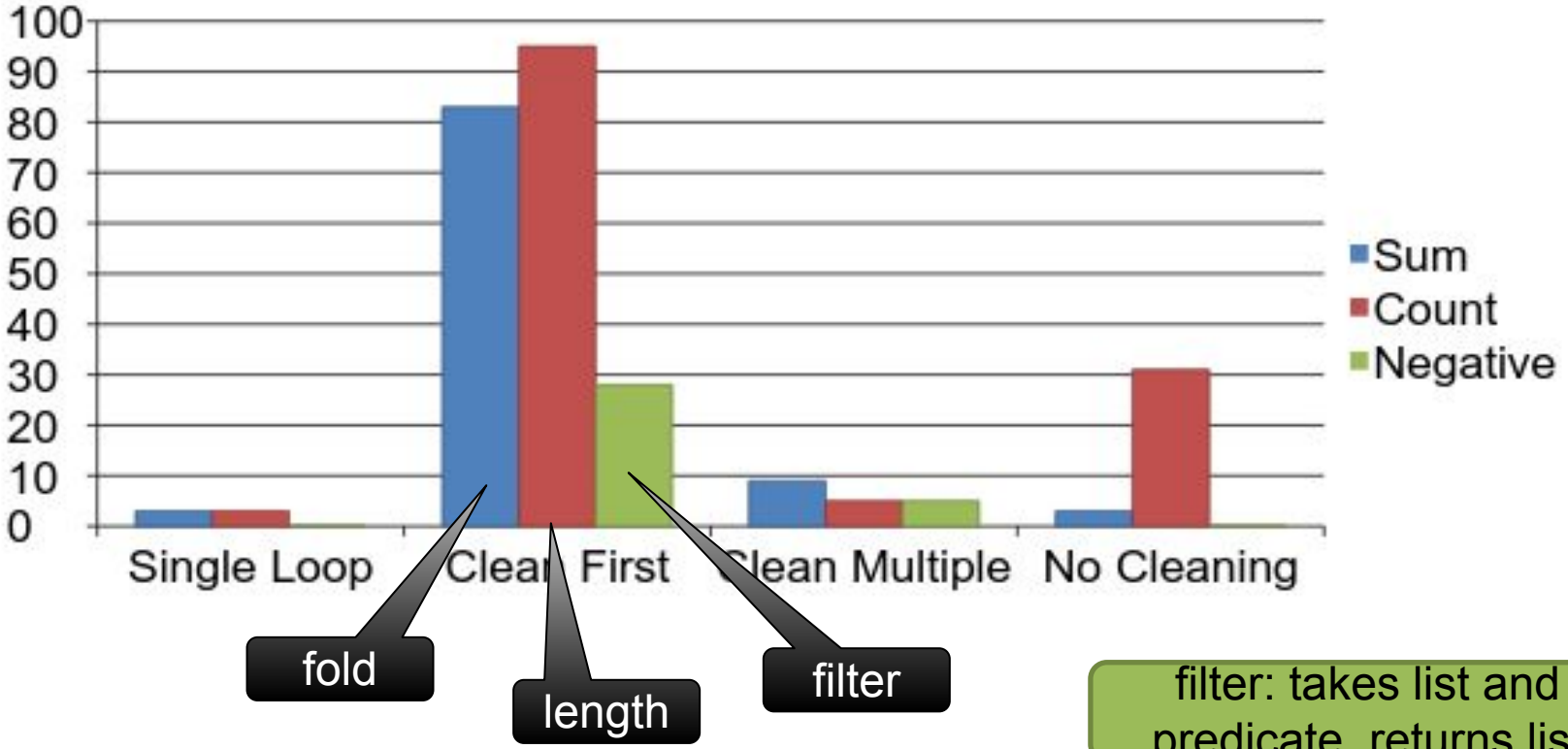
- Why??

- Languages have built in functions
  - filter() which could be used to cleanse
  - sum() to add up list

- These built in functions are like toolkits
  - Once they're in your mind you're more likely to use them to solve problems
  - It's like the language is suggesting that you use them by making it really easy to do so
  - Start thinking of kinds of problems in terms of their solutions
    - This is just a cleansing problem!

Solution Composition

% of students using built-in/higher-order functions



filter: takes list and predicate, returns list

Comparing Solutions

- First solution has less code, depends on two counters, and does a lot of computation inside one for-loop

- Third solution has more code, takes up more space by taking up a second data structure, and runs through the data twice

- So the third one is worse, right?

- MAYBE NOT
- If programming functionally, creating new data structures is very cheap
- Easier to debug: can isolate problems to individual methods and test that they work
- Cleanly broken into 3 conceptual parts
  - Good for understanding, good for reuse!
  - 3 small tasks instead of one big one
- More extensible: if you ever want to use this data again, you would still have to clean it and could reuse your cleanseData() function. But in the first you'd have to write entirely new code!

- If we wanted to write it the second way but not use more space, we could use cleanseData to remove bad input instead of creating a new list
- Return type?
- Would functional programmers do this?
- No!!!
- Relies on mutation, mistakes easier to make and you catch them later
  - If cleanseData() returns an empty list you know you've got a problem, but if it deletes the wrong things the bug is harder to track down

- When methods change the state of the program you have no guarantee about the your variables after the method call
  - It could have deleted everything and you'd have no idea!
- Without mutation, it doesn't matter how a method does something as long as it has right return value
- Separates "what" from "how" – good encapsulation!
- So, use mutation if you really really need it, but not as a default
- Makes sense when you're manipulating huge matrices of data (like for graphics), but often it doesn't

Takeaway:

- Another way to approach problems: prioritizing the extensibility and clarity of your design instead of just pure space/number of times you look at the data
- Keep in mind that the most efficient solution may depend on the language you're working in – there's a reason background affects what you think is best!
- Different approaches can inform your thought process and help you develop better solutions
  - Some problems are best solved with one kind of solution, and a language can lend itself to multiple paradigms

- Mutation vs. none is just one of many choices programming languages make
- Can also not have types!
  - May seem like a small thing, but fundamentally changes your language
  - Inheritance? Polymorphism?
  - Say goodbye!
- Every programming language makes hundreds of choices about which features to include and how to implement them.
  - And then the choices interact with each other
  - It’s very intricate!!

- Which is why programming can be hard
- Every language is going to do things differently, but you should be able to pick up new ones fairly quickly because you've learned *how* to learn a language
- Once you understand the high level choices made in a language and the school of thought it's in, reading writing and understanding it gets way easier
  - Which is why you should learn more about programming language theory!!

Closing:

- Learning different languages gives you whole new ways to look at problems, and whole new toolkits for tackling them!
- You've spent this whole semester learning our style of programming, but it's only one way of looking at problems.
- Learning more languages and understanding their paradigms and how they're put together will make you a much better programmer.
  - And it's really freakin cool!!