Lecture 10: Inheritance

CS 2110, Matt Eichhorn and Leah Perlmutter

September 24, 2025

Announcements

- A5 released, due Wednesday
- Prelim 1 on October 9 (in 2 weeks)
  - Fill out conflict survey
  - Practice exam coming next week
  - Make sure you’re on top of studying!
- DIS 5 this week
  - Fun topic, prep for A5
  - Importance of think-before-you-code activities (coding ability & exam prep)

Today's Learning Outcomes

1. Compare and contrast interfaces, abstract classes, and (concrete) classes.
2. Compare and contrast static types and dynamic types.
3. Explain the benefits of leveraging polymorphism in object-oriented code.
4. Describe the principle of dynamic dispatch and the compile-time reference rule.
5. Explain inheritance relationships and their benefits/drawbacks over interfaces.
6. Given a parent class, use inheritance to develop one or more child subclasses.
7. Determine the correct visibility modifier (public, protected, or private) for a given field or method and justify your choice.
8. Trace through the execution of a code sample that includes one or more of the following: inheritance, overridden methods, and super calls.

Recall: Checking and Savings Accounts

- Both
  - Name, balance, deposit, transfer, transactionReport
- Savings only
  - Earning interest
- Checking only
  - Monthly fee if balance too low

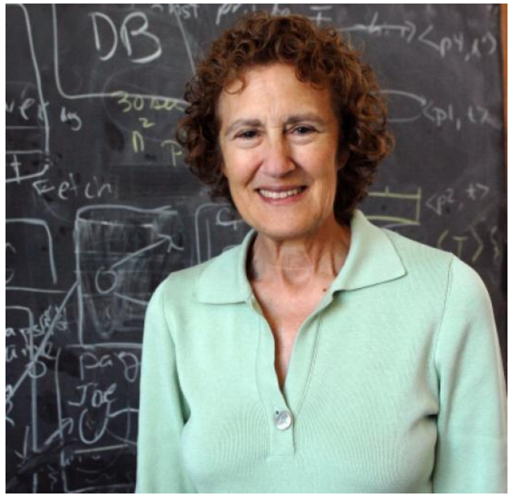
Recall: Substitutability

- Also known as: Liskov Substitution Principle (LSP)
    - next slide

The Liskov Substitution Principle

Let `P(x)` be a property provable about objects x of type T. Then `P(y)` should be true for objects y of type S where S is a subtype of T.

This means B is a subtype of A if *anywhere* you can use an A, you could also use a B.



-- Barbara Liskov

Subclassing

- Lets us reuse code in a subtype
- Account has lots of code that Savings and Checking could reuse!

Subclassing with extends

```java
/** Models an account in our personal finance app. */
public class Account {

}


/** Models a savings account in our personal finance app. */
public class SavingsAccount extends Account {

}
```

Subclassing and fields

```
public class Account {
  private String name;
  private int balance;
  StringBuilder transactions;
...}

public class SavingsAccount extends Account {
  private double rate;
  // name, balance, and transactions are inherited!
...}
```

Subclassing and methods

```
/** Models an account in our personal finance app. */
public class Account {
    // which visibility modifier?
    private  void resetTransactionLog() {...}
    public   String name() {...}
    public   int balance() {...}
    public   boolean transferFunds(Account receiving, int amount) {...}
    public   String transactionReport() {...}
  protected static String centsToString(int cents) {...}
}
```

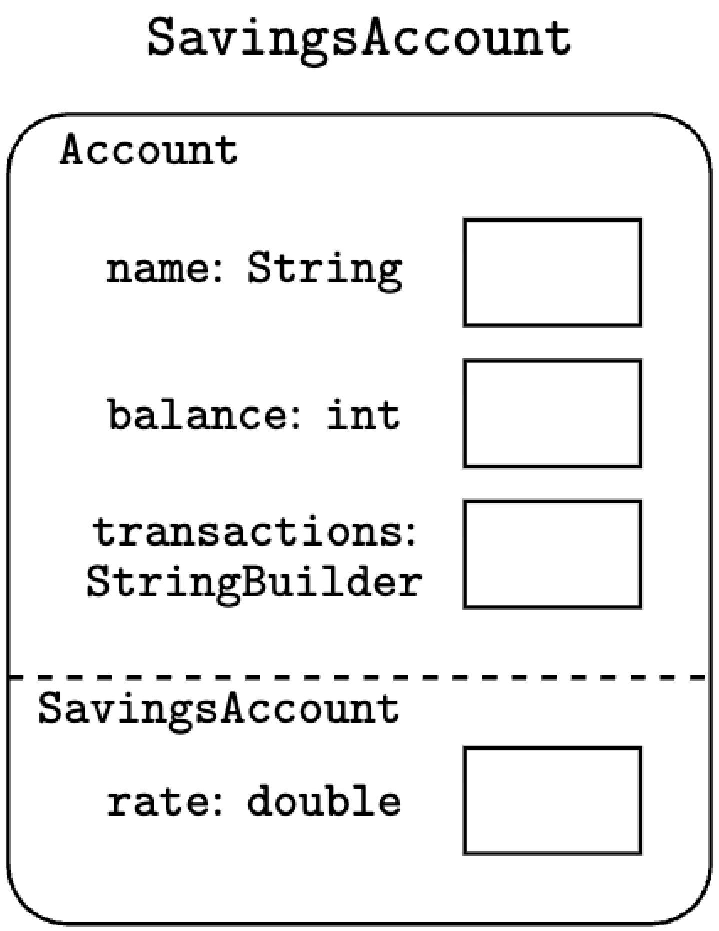
Specialization Interfaces

- specialization interface
  - fields and methods that are visible to child classes but not clients
  - contrast with *client interface,* which is public members of a class visible to client code
- **`protected`**
  - grants access to subclasses and not (usually) to the client
- Caution: avoid exposing representation to subclass
  - super class is responsible for managing rep invariant

Specialization Interfaces

```
/** Models an account in our personal finance app. */
public class Account {
    // which visibility modifier?
    private   void resetTransactionLog() {...}
    public    String name() {...}
    public    int balance() {...}
    public    boolean transferFunds(Account receiving, int amount) {...}
    public    String transactionReport() {...}
  protected static String centsToString(int cents) {...}
}
```

Object diagrams revisited



Reading fields with observer method

```
public class Account {
  private int balance;
  ...
}

public class SavingsAccount extends Account {
 private void accrueMonthlyInterest() {
    int interestAmount =
                  (int)(this.balance() * this.rate / (12 * 100));
    this.depositFunds(interestAmount, "Monthly interest @"
                       + this.rate + "%");
  }
}
```

Overriding methods

```
// Account
public String transactionReport() {
    this.transactions.append("Final Balance: ");
    this.transactions.append(centsToString(this.balance));
    this.transactions.append("\n");

    String report = this.transactions.toString();
    this.resetTransactionLog();
    return report;
}

// SavingsAccount
@Override
public String transactionReport() {
    this.processMonthlyFee();
    this.transactions.append("Final Balance: ");
    this.transactions.append(centsToString(this.balance));
    this.transactions.append("\n");

    String report = this.transactions.toString();
    this.resetTransactionLog();
    return report;
}
```

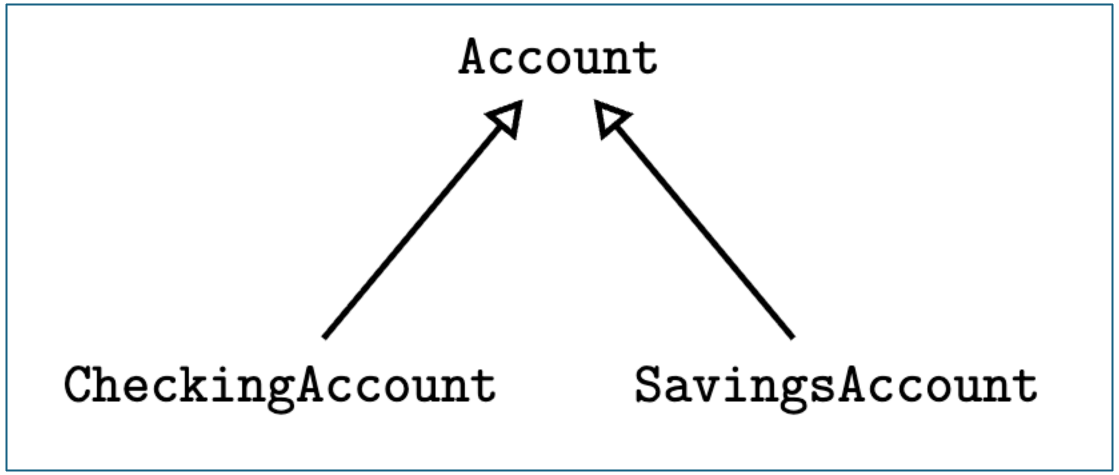
Overriding methods

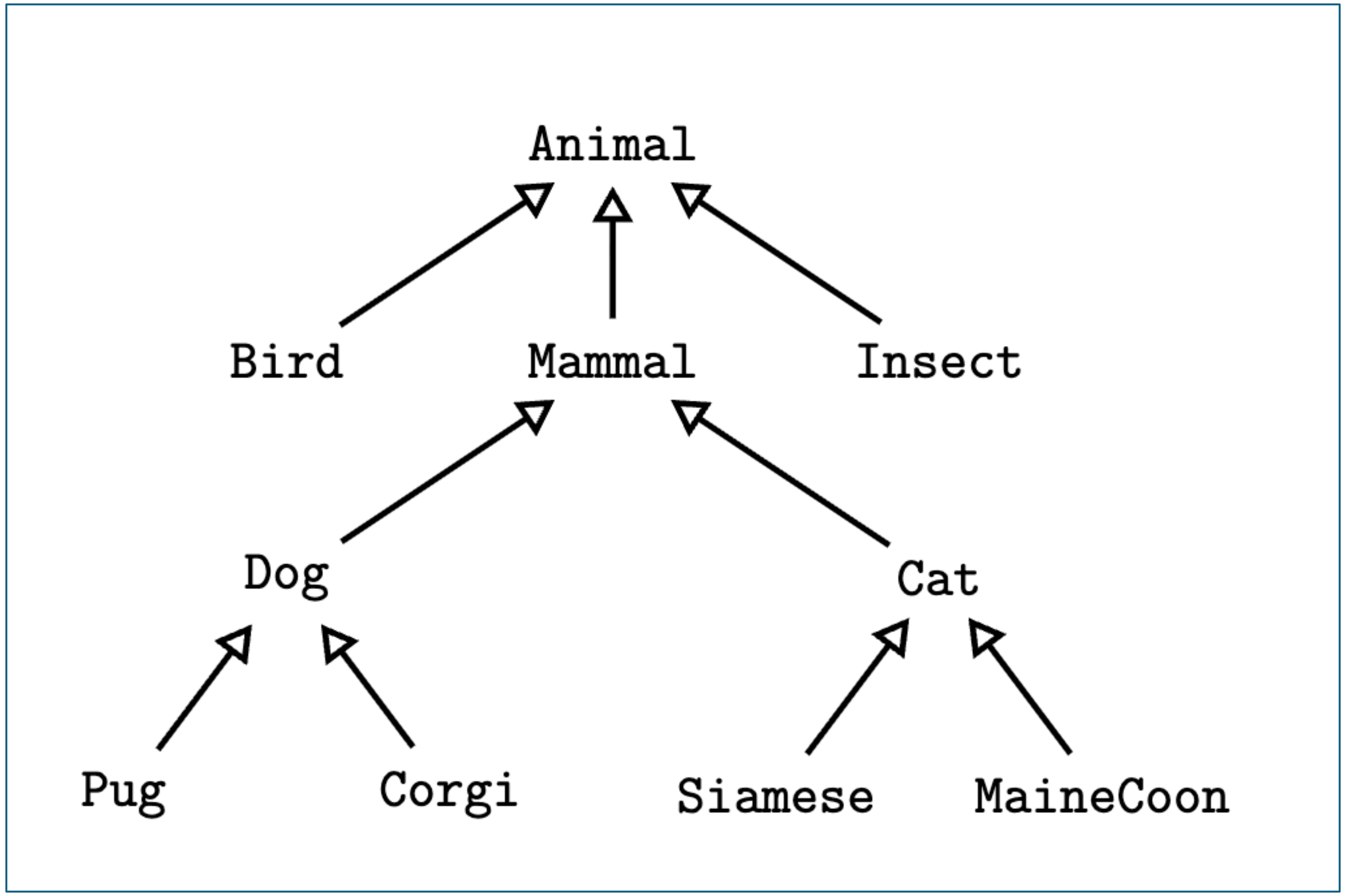
```
// Account
public String transactionReport() {
    this.transactions.append("Final Balance: ");
    this.transactions.append(centsToString(this.balance));
    this.transactions.append("\n");

    String report = this.transactions.toString();
    this.resetTransactionLog();
    return report;
}

// SavingsAccount
@Override
public String transactionReport() {
    this.processMonthlyFee();
    return super.transactionsReport();
}
```

invoke the parent class's method!

we can do this with constructors too, but the call to super MUST be on the first line

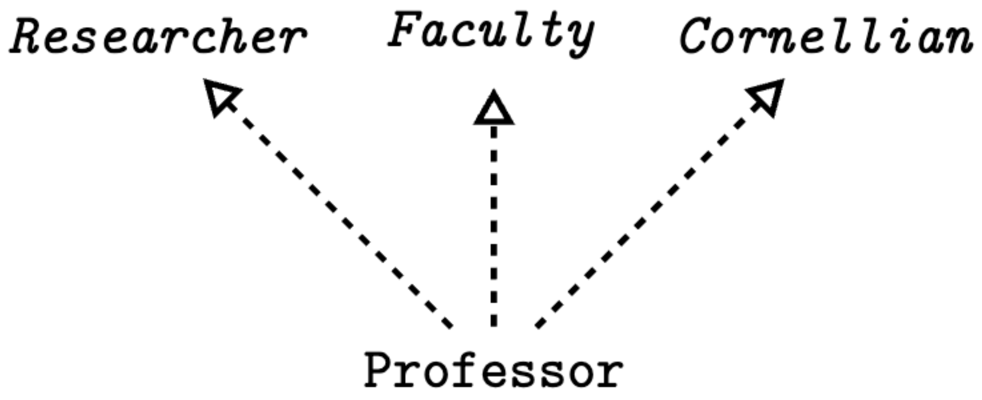
Type hierarchies





Single Inheritance in Java

- In Java, each class can extend only one superclass
- Use implementation inheritance sparingly!
- Prefer interface implementation



Abstract Classes

- middle ground between implementation inheritance and implementing an interface
- inheritance -- superclass has methods that we override
- interface -- super has method declarations that we implement
- abstract -- abstract super has both (!!)
- why?
  - super might want to leave space for the subclass to do something, but super doesn't know what that might be, it's up to the subclass
  - you might hear this called a "hook"
  - super makes a hook so that subclass can hang something on it

transactionReport(), revisited

```
public class Account {

  /**
   * Called once at the end of each month to return a `String` summarizing the
   *   account's initial balance that month, all transactions made during that
   *   month, and its final balance.
   * To maintain class invariant, subclasses that override transactionReport()
   *   must call super.transactionReport() within the body of the
   *   overriding method
   */
  public String transactionReport() {
      this.transactions.append("Final Balance: ");
      this.transactions.append(centsToString(this.balance));
      this.transactions.append("\n");

      String report = this.transactions.toString();
      this.resetTransactionLog();
      return report;
  }
...
}
```

Too much responsibility for subclass!

Superclass should be responsible for its own invariants!

`transactionReport()`, revisited

```
public abstract class Account {
  /**
   * Called once at the end of each month to return a `String` summarizing the
   *   account's initial balance that month, all transactions made during that
   *   month, and its final balance.
   */
  public String transactionReport() {
      this.closeOutMonth();
      this.transactions.append("Final Balance: ");
      this.transactions.append(centsToString(this.balance));
      this.transactions.append("\n");

      String report = this.transactions.toString();
      this.resetTransactionLog();
      return report;
  }
  protected abstract void closeOutMonth();
...
}
```

abstract class!

abstract method!

Dynamic Dispatch with subclassing

```java
public abstract class Account {
  public String transactionReport() {
    this.closeOutMonth();
    ...
  }
  protected abstract void closeOutMonth();
  ...
}

public class CheckingAccount extends Account {
  @Override
  protected void closeOutMonth() {
    ...
  }
...
}
```

```java
// Client code
Account checking =
  new CheckingAccount("Checking", 13000);
Account savings =
  new SavingsAccount("Savings", 230000, 3.0);
checking.transferFunds(savings, 10000);
System.out.println(checking.transactionReport());
```

Dynamic Dispatch with subclassing

```
public abstract class Account {
  public String transactionReport() {
    this.closeOutMonth();
    ...
  }
  protected abstract void closeOutMonth();
  ...
}

public class CheckingAccount extends Account {
  @Override
  protected void closeOutMonth() {
    ...
  }
...
}
```

```
// Client code
Account checking =
  new CheckingAccount("Checking", 13000);
Account savings =
  new SavingsAccount("Savings", 230000, 3.0);
checking.transferFunds(savings, 10000);
System.out.println(checking.transactionReport());
```

What happens when we try to call `checking.transactionReport()` in the client code above and `checkingTransactionReport()` tries to call `closeOutMonth()`?

A) It calls `Account's closeOutMonth()` which does nothing
B) It tries to call `Account's closeOutMonth()` which results in an error
C) It calls `CheckingAccount's closeOutMonth()`
D) Compile time error

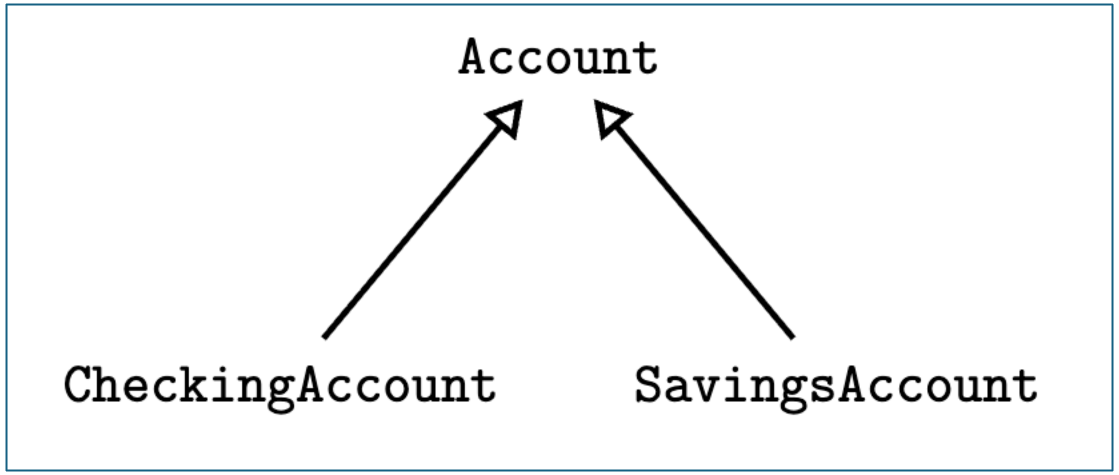
PollEv.com/leahp
text **leahp** to **22333**

Dynamic Dispatch with subclassing

```
public abstract class Account {
  public String transactionReport() {
    this.closeOutMonth();
    ...
  }
  protected abstract void closeOutMonth();
  ...
}

public class CheckingAccount extends Account {
  @Override
  protected void closeOutMonth() {
    ...
  }
...
}
```

```
// Client code
Account checking =
   new CheckingAccount("Checking", 13000);
Account savings =
   new SavingsAccount("Savings", 230000, 3.0);
checking.transferFunds(savings, 10000);
System.out.println(checking.transactionReport());
```

Bottom Up Rule!



Dynamic Dispatch with subclassing

```java
public abstract class Account {
  public String transactionReport() {
    this.closeOutMonth();
    ...
  }
  protected abstract void closeOutMonth();
...
}

public class CheckingAccount extends Account {
  @Override
  protected void closeOutMonth() {
    ...
  }
...
}
```

```java
// Client code
Account checking =
  new CheckingAccount("Checking", 13000);
Account savings =
  new SavingsAccount("Savings", 230000, 3.0);
checking.transferFunds(savings, 10000);
System.out.println(checking.transactionReport());
```

What happens when we try to call `checking.transactionReport()` in the client code above and `checkingTransactionReport()` tries to call `closeOutMonth()`?

A) It calls `Account's closeOutMonth()` which does nothing
B) It tries to call `Account's closeOutMonth()` which results in an error
C) It calls `CheckingAccount's closeOutMonth()`
D) Compile time error

PollEv.com/leahp
text **leahp** to **22333**

Class Design Case Study: Storing Names as Data

Storing People's Names as Data (Social Implications)

- There are social implications of using data to represent reality!
- Names are a kind of data commonly stored in many different data structures
- Caution: potentially more questions than answers here!

Use Case: Storing Names in a Hospital Patient Database

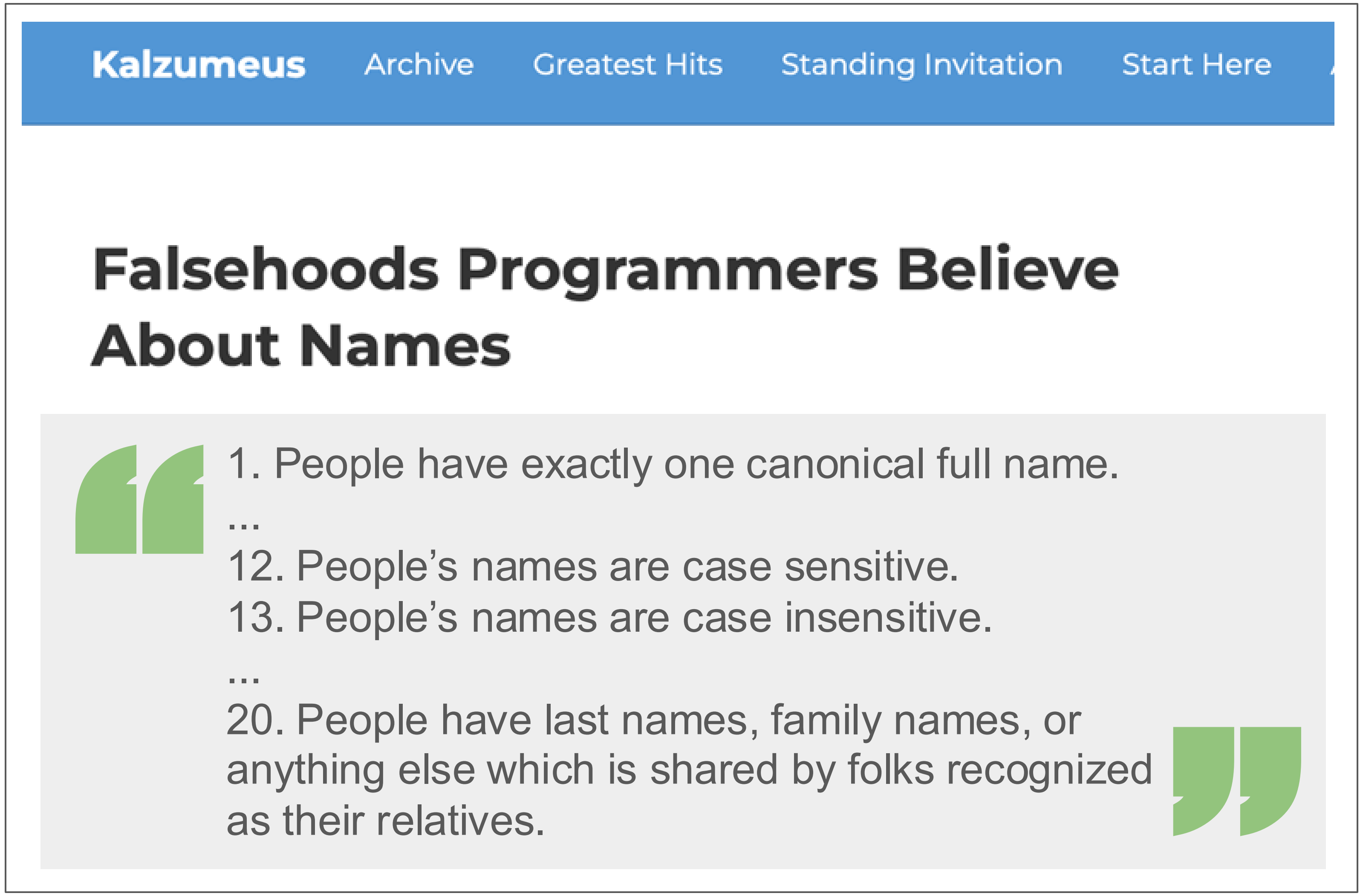
As a patient experience specialist, I want to make sure that patients are treated in a respectful way. This includes having hospital staff address patients respectfully. The hospital database will store patient names in 3 parts:

- Honorific (Ms, Mr, etc.) -- dropdown
- First Name -- free text
- Last Name -- free text

Then hospital staff will be instructed to address patients using [Honorific] [Last Name], for example "Ms. Perlmutter," because that is more respectful than simply using their first name, for example, "Leah."

What assumptions are made in this use case?

Storing Names as Data

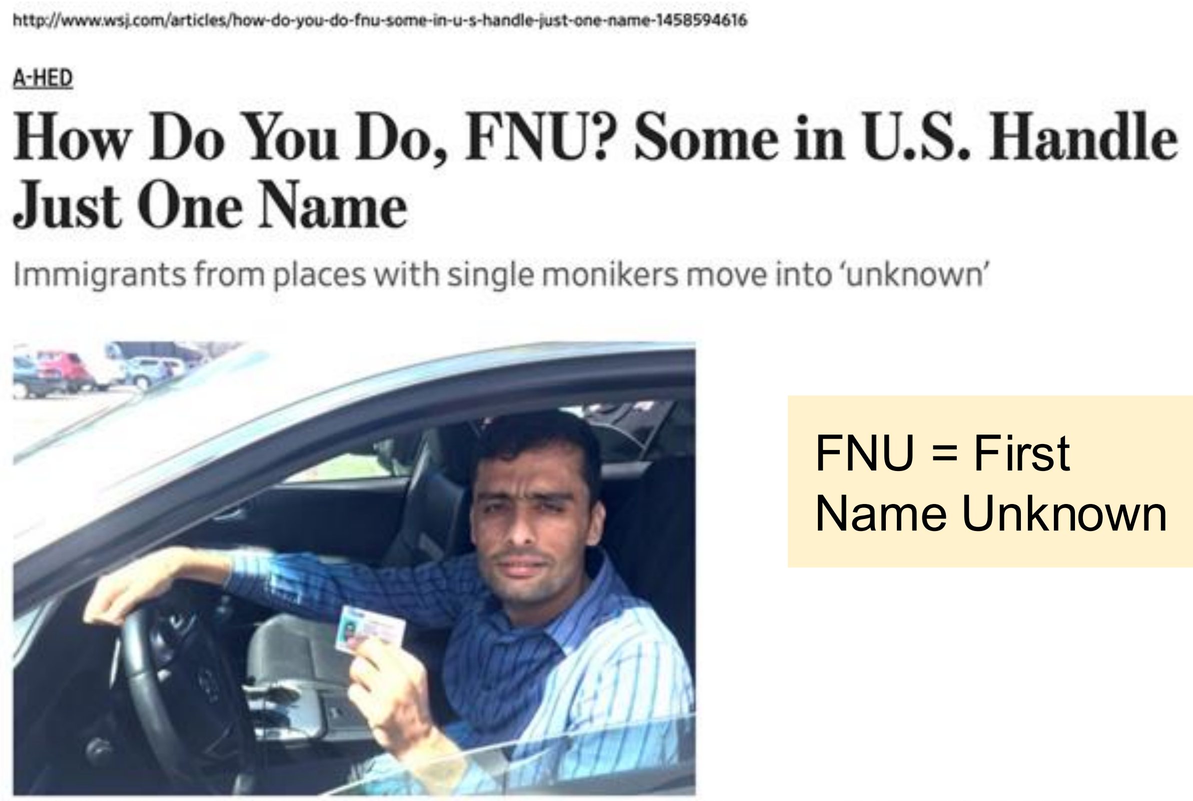


Falsehoods Programmers Believe About Names

1. People have exactly one canonical full name.
...
12. People’s names are case sensitive.
13. People’s names are case insensitive.
...
20. People have last names, family names, or anything else which is shared by folks recognized as their relatives.

Just one name = Mononym

Source: https://www.kalzumeus.com/2010/06/17/falsehoods-programmers-believe-about-names/



http://www.wsj.com/articles/how-do-you-do-fnu-some-in-u-s-handle-just-one-name-1458594616

A-HED

How Do You Do, FNU? Some in U.S. Handle Just One Name

Immigrants from places with single monikers move into 'unknown'

FNU = First Name Unknown

“After vetting and interviews, he received a visa at the U.S. embassy in Kabul identifying him as “FNU Naqibullah.” ... he also became FNU Naqibullah on his driver’s license, Social Security card and other identification. “Everywhere I go, they are calling me FNU," says Naqibullah.”

VICE

The Uniquely Indonesian Pains of Having Only One Name

Think your life is difficult? Try going through life with only a first name.

By Alice .

June 6, 2017, 2:23am Share Tweet Snap

“Perhaps the cure to all these "issues" is applying to the district court for a last name, ... But what last name should I pick? My ancestors' familial name, Huang? My parents' chosen name, Wijaya?”

Almost all data is an approximation of reality

Metacognition

- Take 1 minute to write down a brief summary of what you have learned today

closing announcements to follow...

Announcements

- A5 released, due Wednesday
- Prelim 1 on October 9 (in 2 weeks)
  - Fill out conflict survey
  - Practice exam coming next week
  - Make sure you're on top of studying!
- DIS 5 this week
  - Fun topic, prep for A5
  - Importance of think-before-you-code activities (coding ability & exam prep)