CS 106A, Lecture 21
Classes

suggested reading:

Java Ch. 6

Plan for today

- Review: HashMaps
- HashMaps as Counters
- Classes
- Recap

Learning Goals

- Know how to define our own variable types

Plan for today

- Review: HashMaps
- HashMaps as Counters
- Classes
- Recap

Introducing… HashMaps!

- A variable type that represents a collection of unordered **key-value pairs**
- You access a value associated with each *key*
- Keys and values can be any type of **Object**
- Keys are unique
- Resizable – can add and remove pairs

HashMap Examples

- **Phone book:** name -> phone number
- **Search engine:** URL -> webpage
- **Dictionary**: word -> definition
- **Bank**: account # -> balance
- **Social Network**: name -> profile
- **Counter**: text -> # occurrences
- And many more…

Our First HashMap

```
HashMap<String, String> myHashMap = new HashMap<>();
```

Review: HashMap Operations

- *m*.put(*key*, *value*); Adds a key/value pair to the map.

```
m.put("Eric", "650-123-4567");
```

  - Replaces any previous value for that key.

- *m*.get(*key*) Returns the value paired with the given key.

```
String phoneNum = m.get("Jenny");  // "867-5309"
```

  - Returns null if the key is not found.

- **m.remove**(*key*); Removes the given key and its paired value.

```
m.remove("Annie");
```

  - Has no effect if the key is not in the map.

key	value
"Jenny"	→ "867-5309"
"Mehran"	→ "123-4567"
"Marty"	→ "685-2181"
"Chris"	→ "947-2176"

Data Structure Recap

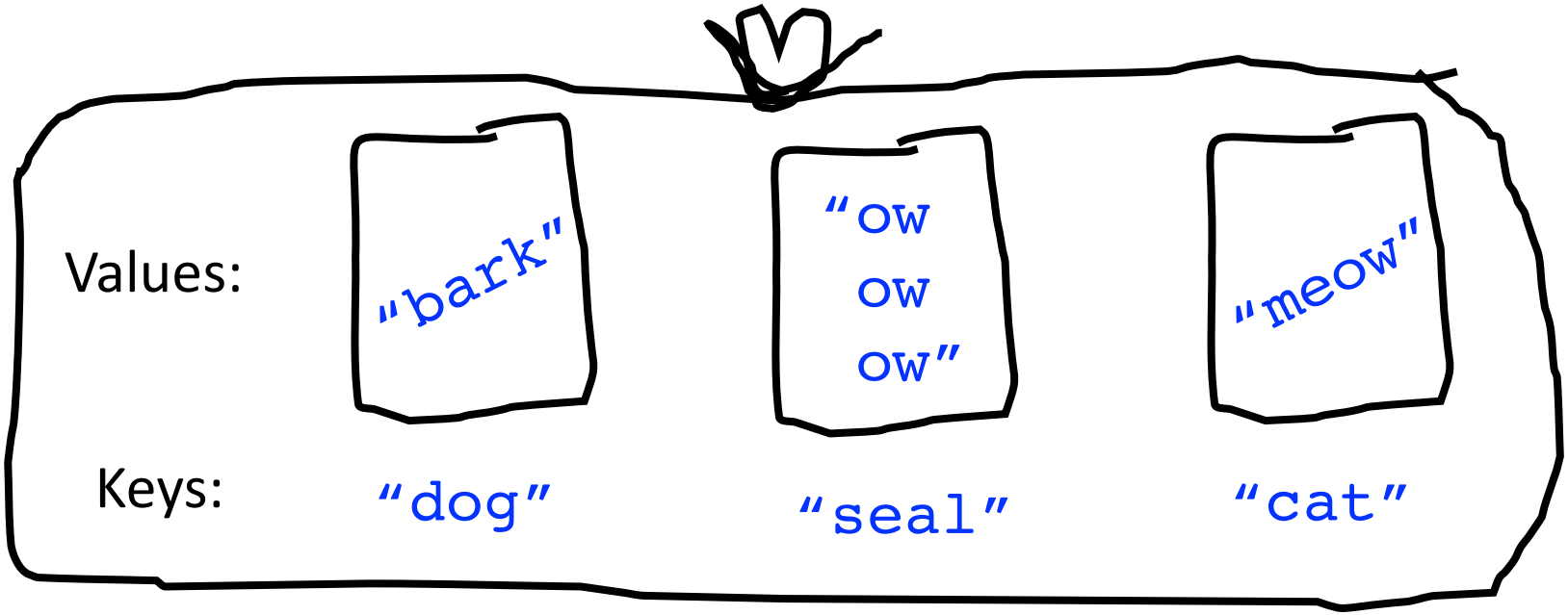
- ArrayLists are a variable type representing a list of items
- Unlike arrays, ArrayLists have:
  - The ability to resize dynamically
  - Useful methods you can call on them
- Unlike ArrayLists, arrays have:
  - The ability to store any type of item, not just Objects
- HashMaps are a variable type representing a key-value pairs
- Unlike arrays and ArrayLists, HashMaps:
  - Are not ordered
  - Store information associated with a key of any Object type

Plan for today

- Review: HashMaps
- HashMaps as Counters
- Classes
- Recap

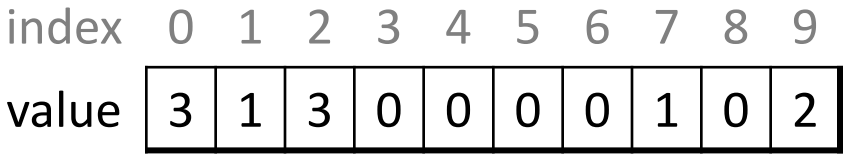
Iterating Over HashMaps

```
...
for (String key : map.keySet()) {
  String value = map.get(key);
  // do something with key/value pair...
}
// Keys occur in an unpredictable order!
```



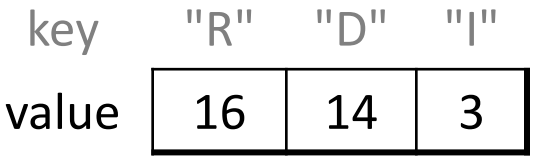
Maps and Tallying

- a map can be thought of as generalization of a tallying array
  - the "index" (key) doesn't have to be an `int`
  - count digits: `22092310907`




```
// (R)epublican, (D)emocrat, (I)ndependent
```

  - count votes: `"RDDDDDDRRRRRDDDDDDRDRRIRDRRIRDRRID"`



key		value
"R"	→	16
"D"	→	14
"I"	→	3

Practice: What's Trending?

- Social media can be used to monitor popular conversation topics.
- Write a program to count the frequency of **#hashtags** in tweets:
  - Read saved tweets from a large text file.
  - Report hashtags that occur at least 15 times.

- How can a **map** help us solve this problem?

Given these hashtags…

```
#stanford
#summer
#california
#stanford
```

We want to store...

```
"#stanford"   → 2
"#summer"     → 1
"#california" → 1
```

Plan for today

- Review: HashMaps
- HashMaps as Counters
- Classes
- Recap

Large Java Programs

There are some *large* programs written in Java!

Defining New Variable Types

Artist	Album	Song
- Albums - Awards	- Songs - Producer	- Length - Collaborators

What Is A Class?

A class defines a new variable type.

Why Is This Useful?

- **Classes** let you define new types of variables, which lets you decompose your program code across different files.

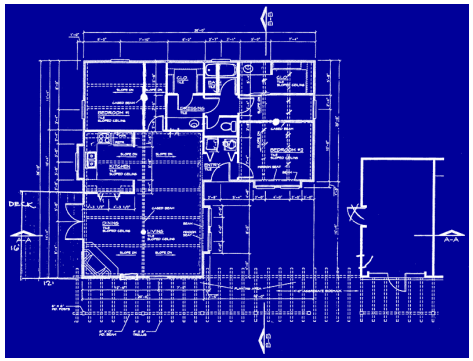
- Non-primitive variable types "hide" information. They let programmers do potentially complicated operations without having to understand *how* those operations are performed.
  - Example: The ArrayList class provides programmers a contract: "Give me a value to add, and it will end up at the end of the list." Behind the scenes, the ArrayList might have to make a new array, copy all the old elements into the new array, and then put the value in the first open slot in the new array.

Classes Are Like Blueprints

iPod blueprint (class)

state:
- current song
- volume
- battery life

behavior:
- power on/off
- change station/song
- change volume
- choose random song



constructs

iPod (variable) #1

state:
- song = "Follow Your Arrow"
- volume = 17
- battery life = 2.5 hrs

behavior:
- power on/off
- change station/song
- change volume
- choose random song

iPod (variable) #2

state:
- song = "Tightrope"
- volume = 9
- battery life = 3.41 hrs

behavior:
- power on/off
- change station/song
- change volume
- choose random song

iPod (variable) #3

state:
- song = "Burn"
- volume = 24
- battery life = 1.8 hrs

behavior:
- power on/off
- change station/song
- change volume
- choose random song

Creating A New Class

Let's define a new variable type called **BankAccount** that represents information about a single person's bank account.

A **BankAccount**:

- contains the name of account holder
- contains the balance
- can deposit money
- can withdraw money

What if…

What if we could write a program like this:

```
BankAccount colinAccount = new BankAccount("Colin", 50);
colinAccount.deposit(50);
println("Colin now has: $" + colinAccount.getBalance());

BankAccount annieAccount = new BankAccount("Annie");
annieAccount.deposit(50);
boolean success = annieAccount.withdraw(10);
if (success) {
      println("Annie withdrew $10.");
}
println(annieAccount);
```

Creating A New Class

1. **What information is inside this new variable type?**
   - These are its private instance variables.

Example: BankAccount

```
// In file BankAccount.java
public class BankAccount {
     // Step 1: the data inside a BankAccount
     private String name;
     private double balance;
}
```

Each BankAccount object has its *own copy* of all instance variables.

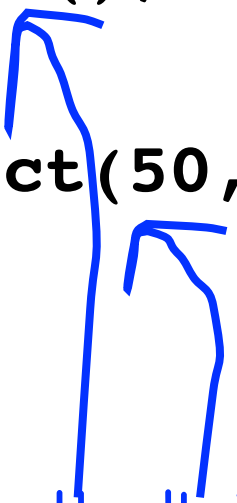
Creating A New Class

1. **What information is inside this new variable type?**
   - These are its instance variables.
2. **How do you create a variable of this type?**
   - This is the constructor.

Constructors

```
GRect rect = new GRect();

GRect rect2 = new GRect(50, 50);
```



This is calling a special method! The GRect **constructor**.

Constructors

```
BankAccount ba1 = new BankAccount("Colin", 50);

BankAccount ba2 = new BankAccount("Annie");
```

The constructor is executed when a new object is created.

Example: BankAccount

```
public class BankAccount {
  // Step 1: the data inside a BankAccount
  private String name;
  private double balance;

  // Step 2: how to create a BankAccount
  public BankAccount(String accountName, double startBalance) {
       name = accountName;
       balance = startBalance;
  }

  public BankAccount(String accountName) {
       name = accountName;
       balance = 0;
  }
}
```

Constructors

- **constructor**: Initializes the state of new objects as they are created.

```
public ClassName(parameters) {
    statements;
}
```

  - The constructor runs when the client says `new ClassName(...);`
  - no return type is specified; it "returns" the new object being created
  - If a class has no constructor, Java gives it a *default constructor* with no parameters that sets all fields to default values like `0` or `null`.

Using Constructors

```
BankAccount ba1 =
    new BankAccount("Marty");
```

ba1

```
name    = "Marty"
balance = 0.0

BankAccount(nm, bal) {
    name = nm;
    balance = bal;
}
```

```
BankAccount ba2 =
    new BankAccount("Mehran", 900000.00);
```

ba2

```
name    = "Mehran"
balance = 900000.00

BankAccount(nm, bal) {
    name = nm;
    balance = bal;
}
```

- When you call a constructor (with **new**):
  - Java creates a new "instance" of that class.
  - The constructor initializes the object's state (instance variables).
  - The newly created object is returned to your program.

Creating A New Class

1. **What information is inside this new variable type?**
   - These are its instance variables.
2. **How do you create a variable of this type?**
   - This is the constructor.
3. **What can this new variable type do?**
   - These are its public methods.

What if…

What if we could write a program like this:

```
BankAccount colinAccount = new BankAccount("Colin", 50);
colinAccount.deposit(50);
println("Colin now has: $" + colinAccount.getBalance());

BankAccount annieAccount = new BankAccount("Annie");
annieAccount.deposit(50);
boolean success = annieAccount.withdraw(10);
if (success) {
      println("Annie withdrew $10.");
}
println(annieAccount);
```

Example: BankAccount

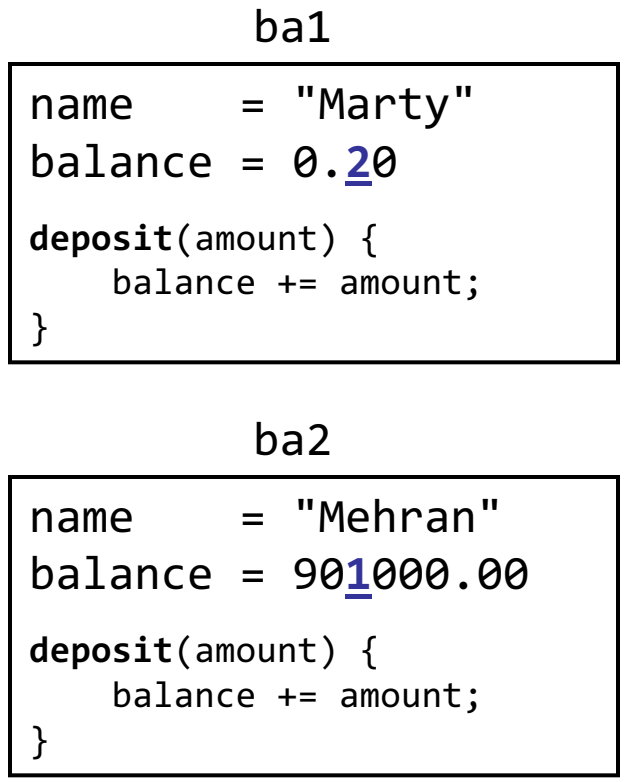
```
public class BankAccount {
	// Step 1: the data inside a BankAccount
	private String name;
	private double balance;
	// Step 2: how to create a BankAccount (omitted)
	// Step 3: the things a BankAccount can do
	public void deposit(double amount) {
		balance += amount;
	}
	public boolean withdraw(double amount) {
		if (balance >= amount) {
			balance -= amount;
			return true;
		}
		return false;
	}
}
```

Defining Methods In Classes

Methods defined in classes can be called **on an instance of that class**.

When one of these methods executes, it can reference **that object's copy** of instance variables.

```
ba1.deposit(0.20);
ba2.deposit(1000.00);
```



This means calling one of these methods on different objects will give *different results, reached via the same process*.

Getters and Setters

Instance variables in a class should *always be private*. This is so only the object itself can modify them, and no-one else.

To allow the client to reference them, we define public methods in the class that **set** an instance variable's value and **get** (return) an instance variable's value. These are commonly known as **getters** and **setters**.

```
account.setName("Colin");
String accountName = account.getName();
```

Getters and setters prevent instance variables from being tampered with.

Example: BankAccount

```
public class BankAccount {
        private String name;
        private double balance;

        ...
        public void setName(String newName) {
                if (newName.length() > 0) {
                        name = newName;
                }
        }

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

Printing Variables

- By default, Java doesn't know how to print objects.

```
BankAccount ba1 = new BankAccount("Marty", 1.25);
println("ba1 is " + ba1); // ba1 is BankAccount@9e8c34

// better, but cumbersome to write
println("ba1 is " + ba1.getName() + " with $"
        + ba1.getBalance()); // ba1 is Marty with $1.25


// desired behavior
println("ba1 is " + ba1);   // ba1 is Marty with $1.25
```

The toString Method

A special method in a class that tells Java how to convert an object into a string.

```
BankAccount ba1 = new BankAccount("Marty", 1.25);
println("ba1 is " + ba1);

// the above code is really calling the following:
println("ba1 is " + ba1.toString());
```

- Every class has a `toString`, even if it isn't in your code.
  - Default: class's name @ object's memory address (base 16)

```
BankAccount@9e8c34
```

The toString Method

```
public String toString() {
    code that returns a String
    representing this object;
}
```

– Method name, return, and parameters must match exactly.

– Example:

```
// Returns a String representing this account.
public String toString() {
    return name + " has $" + balance;
}
```

The “this” Keyword

`this`: Refers to the object on which a method is currently being called

```
BankAccount ba1 = new BankAccount();
ba1.deposit(5);

// in BankAccount.java
public void deposit(double amount) {
      // for code above, “this” -> ba1
      ...
}
```

Using "this"

Sometimes we want to name parameters the same as instance variables.

```
public class BankAccount {
    private double balance;
    private String name;
    ...

    public void setName(String newName) {
        name = newName;
    }
}
```

– Here, the parameter to `setName` is named `newName` to be distinct from the object's field `name` .

Using “this”

```
public class BankAccount {
    private double balance;
    private String name;
    ...

    public void setName(String name) {
        name = name;
    }
}
```

Using "this"

We can use "this" to specify which one is the instance variable and which one is the local variable.

```
public class BankAccount {
    private double balance;
    private String name;
    ...

    public void setName(String name) {
        this.name = name;
    }
}
```

Plan for today

- Review: HashMaps
- HashMaps as Counters
- Classes
- Recap

What Is A Class?

A class defines a new variable type.

Creating A New Class

1. **What information is inside this new variable type?**
   - These are its instance variables.
2. **How do you create a variable of this type?**
   - This is the constructor.
3. **What can this new variable type do?**
   - These are its public methods.

Recap

- Review: HashMaps
- HashMaps as Counters
- Classes
- Recap

Next time: classes practice + inheritance