Classes

Lecture 21

CS106A, Summer 2019

Sarai Gould && Laura Cruz-Albrecht

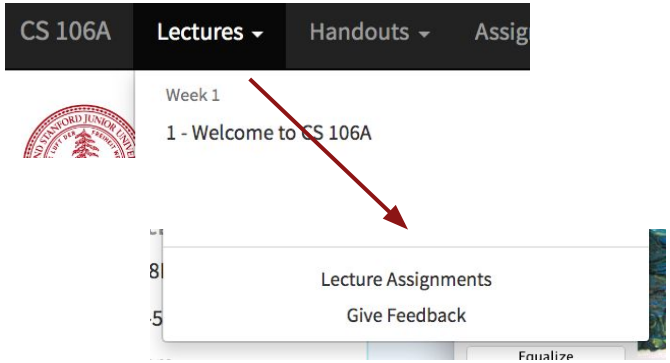
With inspiration from slides created by Keith Schwarz, Mehran Sahami, Eric Roberts, Stuart Reges, Chris Piech, Brahm Capoor, & others.

Announcements

- A Note on the Midterm

Announcements

- Lecture Feedback



Learning Goal for Today

Learn how to define our **own variable types**!

Plan for Today

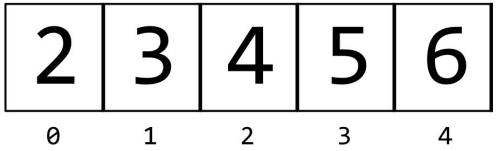
- Revew: Data Structures
- Classes
- Practice: Hedgehog Show
- Recap

Plan for Today

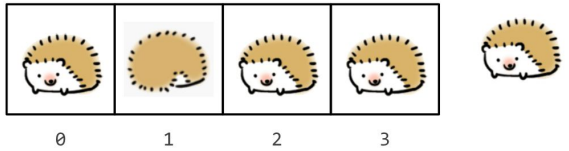
- Revew: Data Structures
- Classes
- Practice: Hedgehog Show
- Recap

Review: Data Structures

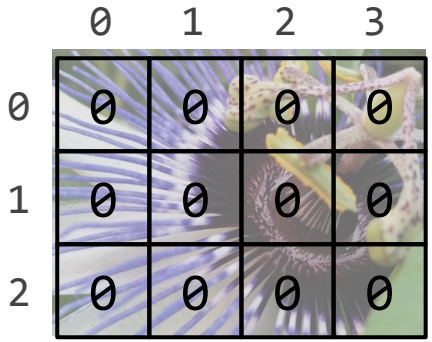
Arrays



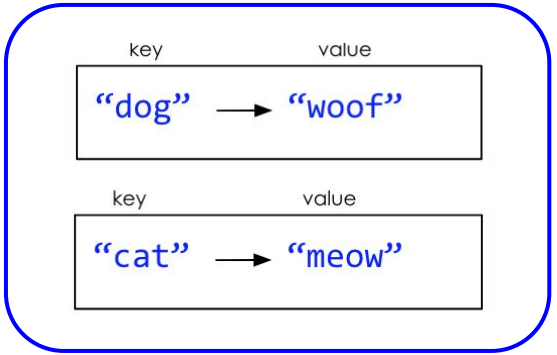
ArrayLists



2D Arrays



HashMaps

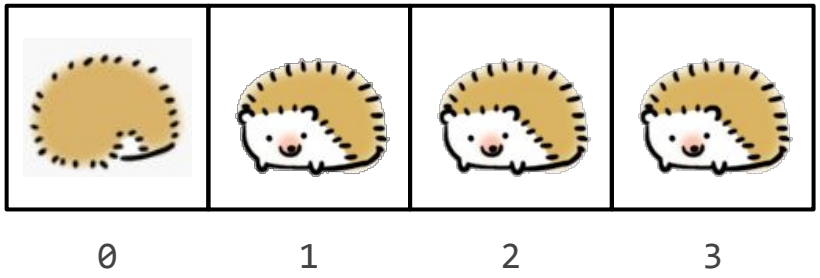


// An Aside

Fun fact: the collective noun for a group of hedgehogs is “an array of hedgehogs”.

// An Aside

Fun fact: the collective noun for a group of hedgehogs is "an **`array`** of hedgehogs".



Plan for Today

- Revew: Data Structures
- Classes
- Practice: Hedgehog Show
- Recap

Some *Large* Programs are in Java

Some *Large* Programs are in Java

How?

Defining New Variable Types

Inbox Database

Email Sender

Login Manager

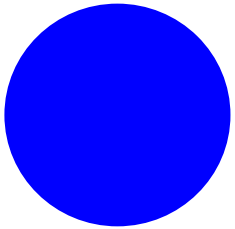
Email

User

Inbox

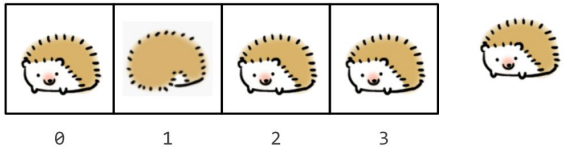
You've Been Using Variable Types

GOval



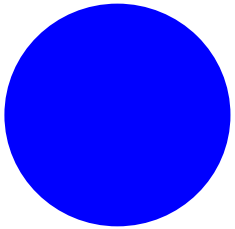
RandomGenerator

ArrayList



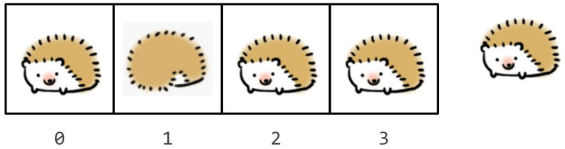
You've Been Using Variable Types

`GOval`



`RandomGenerator`

`ArrayList`



How can we make our *own* variable types?

A class defines a new variable type.

Why Is This Useful?

- A student registration system needs to store info about students, but Java has no `Student` variable type.

- A music synthesizer app might want to store information about different types of instruments, but Java has no `Instrument` variable type.

- An email program might have many emails that need to be stored, but Java has no `Email` type.

Why Is This Useful?

- However, Java does provide a feature for us to add new data types to the language: **classes**.
  - Writing a class defines a new variable type

- This lets you decompose your program across multiple files.

Why Is This Useful?

```
// We already have these variable types
GRect square = new GRect(100, 100);
String msg = “It’s almost August?!”;
ArrayList<String> list = new ArrayList<String>();

// Using classes, we can now make these variable types!
Student s = new Student();
Email email = new Email();
Hedgehog walnoot = new Hedgehog(“Walnoot”);
```

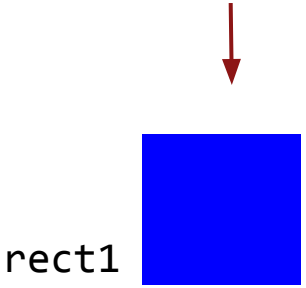
Classes and Objects

- Every object is an ***instance*** of a **class**.
- Each new variable is a new instance.

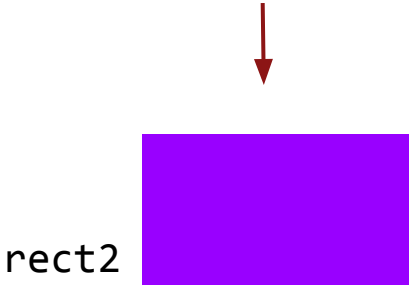
- The class determines:
  - What **state** each instance has.
  - What **behaviors** each instance has.
- Each instance determines:
  - The specific values for that state information.

Classes and Objects

One instance of the `GRect` class

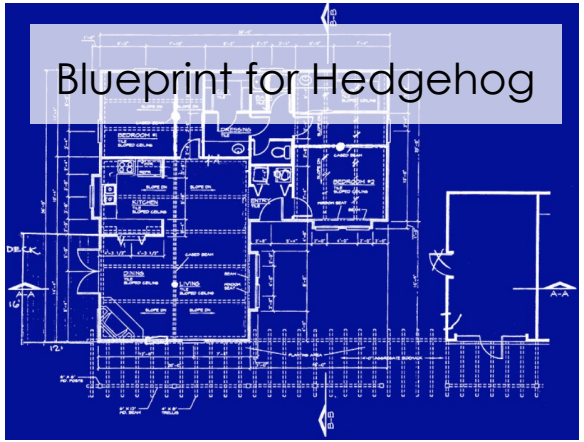


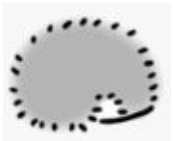
Another instance of the `GRect` class



Classes Are Like Blueprints

class: A template for a new type of variable.



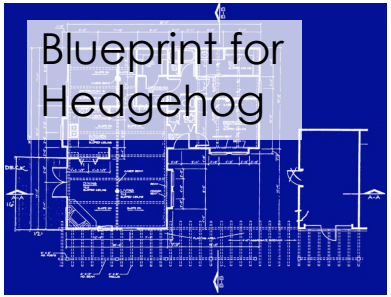


Classes Are Like Blueprints

Hedgehog Class (blueprint)

State: Has name
Has color
Has cuteness level

Behavior: Can eat
Can run*
Can curl up

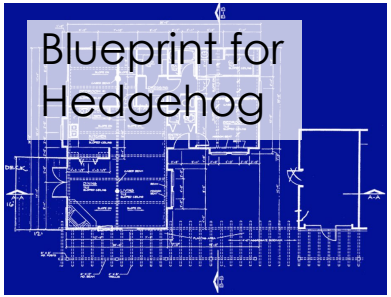


Classes Are Like Blueprints

Hedgehog Class (blueprint)

State: Has name
Has color
Has cuteness level

Behavior: Can eat
Can run*
Can curl up



Hedgehog #1 (variable)

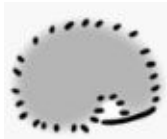
State: **name = "Walnoot"**
color = Brown
cuteness = 10 (Very cute)

Behavior: Can eat
Can run
Can curl up

Hedgehog #2 (variable)

State: **name = "Nutmeg"**
color = Snowflake
cuteness = 15 (VERY cute)

Behavior: Can eat
Can run
Can curl up

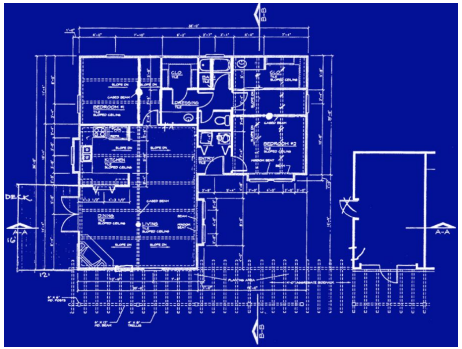


Hedgehog #3 (variable)

State: **name = "Ruffles"**
color = Beige
cuteness = 50 (speechless)

Behavior: Can eat
Can run
Can curl up

Classes Are Like Blueprints



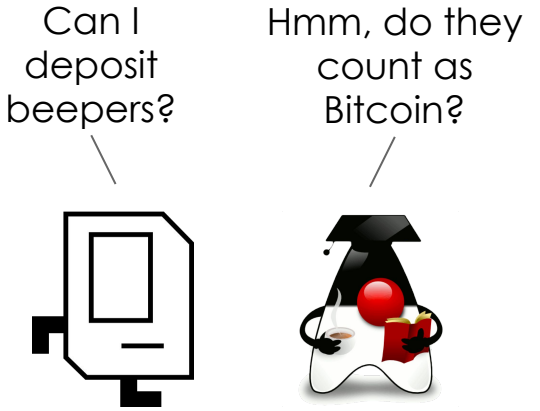
To design a new variable type, you must specify 3 things:

1. What subvariables make up this new variable type? (think: state)
2. How do you create a new variable of this type?
3. What methods can you call on a variable of this type? (think: behaviors)

What If...

What if we could write a program like this?

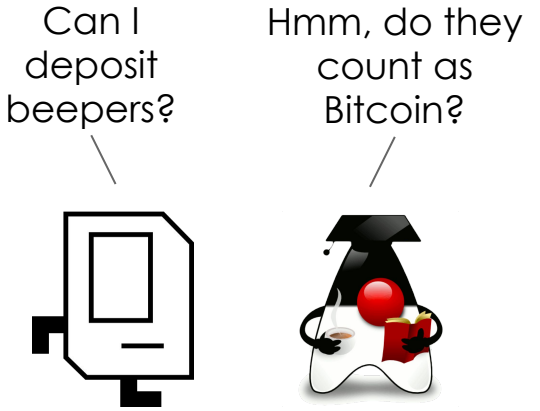
```
BankAccount dukeAccount = new BankAccount("Duke", 50);
dukeAccount.deposit(50);
println("Duke now has: $" + dukeAccount.getBalance());

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



What If…

What if we could write a program like this?

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

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



Creating a New Class

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

Creating a New Class

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

State

A `BankAccount`:

- Contains the name of account holder
- Contains the balance

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`:

State

- Contains the name of account holder
- Contains the balance

Behavior

- Can deposit money
- Can withdraw money

Creating a New Class

```
public class <Classname> {

    // some awesome code

}
```

Creating a New Class

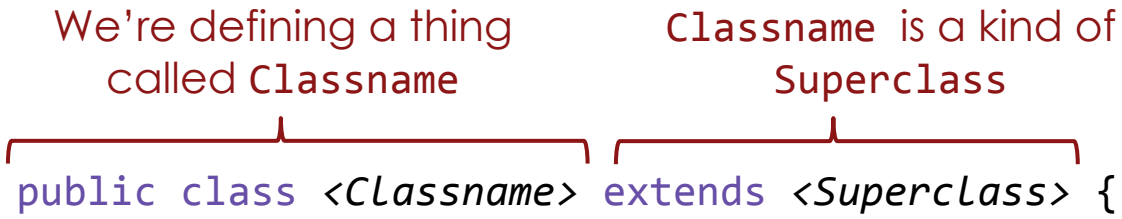
We're defining a thing called `Classname`

```
public class <Classname> {

    // some awesome code

}
```

Creating a New Class

```
public class <Classname> extends <Superclass> {

    // some awesome code

}
```

Creating a New Class



Creating a New Class


```
public class BankAccount {

    // some awesome code

}
```

Creating a New Class

BankAccount.java


```
public class BankAccount {

    // some awesome code

}
```

Creating a New Class

BankAccount.java

```
public class BankAccount {

    // some awesome code

}
```

Creating a New Class

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

Example: BankAccount

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 variable type? (state)**
   - These are its private instance variables

2. **How do you create a new variable of this type?**
   - Constructor

Constructors

```
GRect rect = new GRect();

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

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

Constructors

```
BankAccount dukeAccount = new BankAccount("Duke", 50);

BankAccount karelAccount = new BankAccount("Karel");
```

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 new BankAccount
    public BankAccount(String accountName, double startBalance) {
        this.name = accountName;
        this.balance = startBalance;
    }



}
```

Example: BankAccount

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

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

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

Constructors

- Initializes the state of new objects as they are created

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

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

Using Constructors

```
BankAccount dukeAccount = new BankAccount("Duke", 50);
```

dukeAccount

```
name = "Duke"
balance = 50

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

Using Constructors

```
BankAccount dukeAccount = new BankAccount("Duke", 50);
BankAccount karelAccount = new BankAccount("Karel");
```

dukeAccount

```
name = "Duke"
balance = 50

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

karelAccount

```
name = "Karel"
balance = 0

BankAccount(name) {
     this.name = name;
     this.balance = 0;
}
```

Using Constructors

```
BankAccount dukeAccount = new BankAccount("Duke", 50);
BankAccount karelAccount = new BankAccount("Karel");
```

dukeAccount

```
name = "Duke"
balance = 50

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

karelAccount

```
name = "Karel"
balance = 0

BankAccount(name) {
     this.name = name;
     this.balance = 0;
}
```

When you call a constructor (with **new**):

1. Java creates a new "instance" of that class
2. The constructor initializes the object's state (instance variables)
3. The newly created object is returned to your program

Creating a New Class

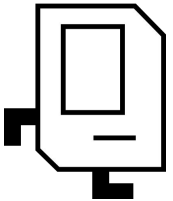
1. **What information is inside this variable type? (state)**
   - These are its private instance variables
2. **How do you create a new variable of this type?**
   - Constructor
3. **What can this new variable type do? (behaviors)**
   - These are its public methods

What If...

What if we could write a program like this?

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

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

I really hope my beepers count as $$...



Example: BankAccount

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

    // Step 2: how to create a new BankAccount (ommitted)

    // Step 3: the things a BankAccount can do
    public void deposit(double amount) {
        this.balance += amount;
    }
    public boolean withdraw(double amount) {
        if (this.balance >= amount) {
            this.balance -= amount;
            return true;
        }
        return false;
    }
}
```

Defining Methods in Classes

Methods defined in classes are called on an instance of that class.

dukeAccount

```
name = “Duke”
balance = 50

deposit(amount) {
     this.balance += amount;
}
```

karelAccount

```
name = “Karel”
balance = 0

deposit(amount) {
     this.balance += amount;
}
```

Defining Methods in Classes

Methods defined in classes are called on an instance of that class.

```
dukeAccount.deposit(22);
```

dukeAccount

```
name = “Duke”
balance = 50

deposit(amount) {
     this.balance += amount;
}
```

karelAccount

```
name = “Karel”
balance = 0

deposit(amount) {
     this.balance += amount;
}
```

Defining Methods in Classes

Methods defined in classes are called on an instance of that class.

```
dukeAccount.deposit(22);
```

dukeAccount

```
name = “Duke”
balance = 50

deposit(amount) {
     this.balance += amount;
}
```

karelAccount

```
name = “Karel”
balance = 0

deposit(amount) {
     this.balance += amount;
}
```

Defining Methods in Classes

Methods defined in classes are called on an instance of that class.

```
dukeAccount.deposit(22);
```

dukeAccount

```
name = “Duke”
balance = 72

deposit(amount) {
     this.balance += amount;
}
```

karelAccount

```
name = “Karel”
balance = 0

deposit(amount) {
     this.balance += amount;
}
```

Defining Methods in Classes

Methods defined in classes are called on an instance of that class.

```
dukeAccount.deposit(22);
karelAccount.deposit(1.99);
```

dukeAccount

```
name = “Duke”
balance = 72

deposit(amount) {
     this.balance += amount;
}
```

karelAccount

```
name = “Karel”
balance = 0

deposit(amount) {
     this.balance += amount;
}
```

Defining Methods in Classes

Methods defined in classes are called on an instance of that class.

```
dukeAccount.deposit(22);
karelAccount.deposit(1.99);
```

dukeAccount

```
name = “Duke”
balance = 72

deposit(amount) {
     this.balance += amount;
}
```

karelAccount

```
name = “Karel”
balance = 0

deposit(amount) {
     this.balance += amount;
}
```

Defining Methods in Classes

Methods defined in classes are called on an instance of that class.

```
dukeAccount.deposit(22);
karelAccount.deposit(1.99);
```

dukeAccount

```
name = “Duke”
balance = 72

deposit(amount) {
     this.balance += amount;
}
```

karelAccount

```
name = “Karel”
balance = 1.99

deposit(amount) {
     this.balance += amount;
}
```

Defining Methods in Classes

Methods defined in classes are called on an instance of that class.

```
dukeAccount.deposit(22);
karelAccount.deposit(1.99);
```

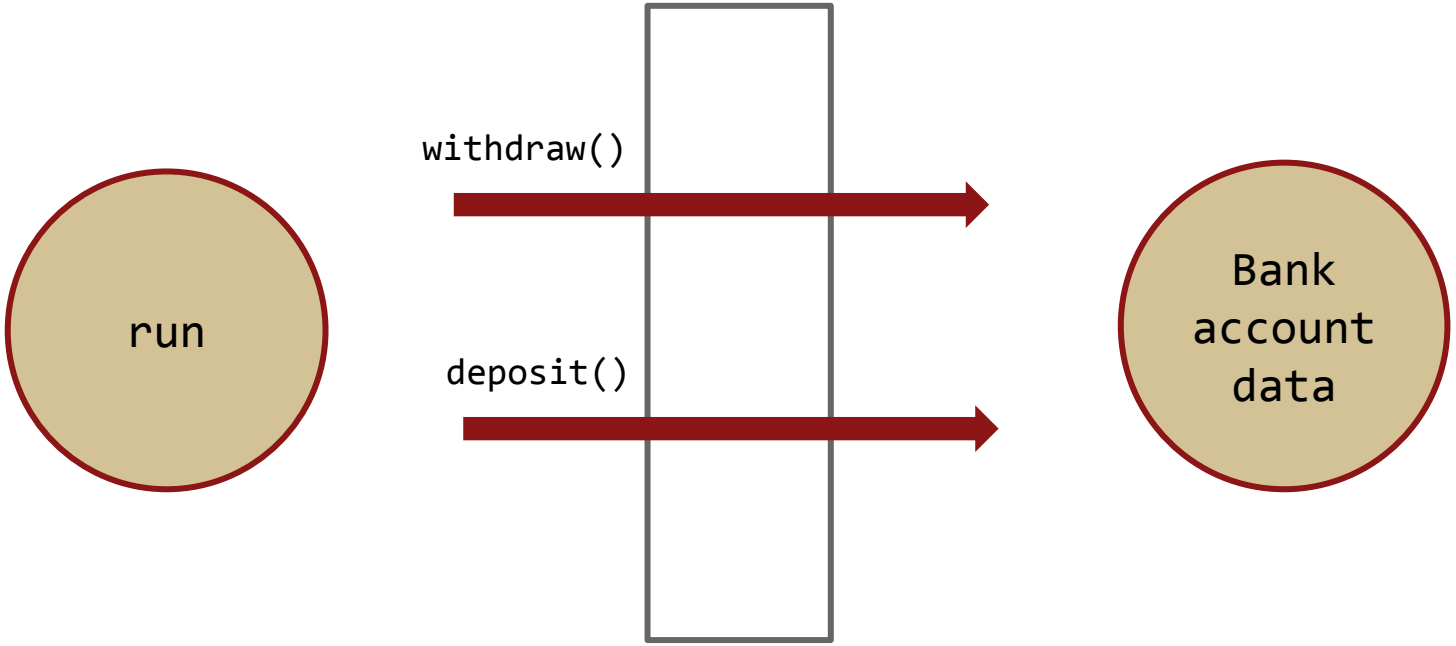
dukeAccount

```
name = “Duke”
balance = 72

deposit(amount) {
     this.balance += amount;
}
```

karelAccount

```
name = “Karel”
balance = 1.99

deposit(amount) {
     this.balance += amount;
}
```

Wall of Abstraction



Wall of Abstraction

Adding Privacy

```
private double balance;
```

- **encapsulation**: Hiding implementation details of an object from its clients.
  - Encapsulation provides *abstraction*. Separates external view (behavior) from internal view (state).
  - Encapsulation protects the integrity of an object's data.

- A class's instance variables should always be private.
  - No code outside the class can directly access/change it.

Adding Privacy

```
// BankAccountProgram.java
BankAccount dukeAccount = new BankAccount("Duke", 50);
dukeAccount.deposit(22);
println("Duke has $" + dukeAccount.balance);    // doesn't work
```

Only accessible inside BankAccount.java

Getters & Setters

```
// BankAccountProgram.java
BankAccount dukeAccount = new BankAccount(“Duke”, 50);
dukeAccount.deposit(22);
println(“Duke has $” + dukeAccount.getBalance());   // but this does!
```

Getters & Setters

- To allow the client to reference private instance variables, we define public methods in the class that
  - **set** an instance variable's value ("getters"), and
  - **get** (return) an instance variable's value ("setters")

- Getters and setters prevent instance variables from being tampered with.

Example: BankAccount

```
public class BankAccount {
     private String name;
     private double balance;
     …
     // “setter”
     public void setName(String newName) {
          if (newName.length() > 0) {
               this.name = newName;
          }
     }

}
```

Example: BankAccount

```
public class BankAccount {
     private String name;
     private double balance;
     …
     // “setter”
     public void setName(String newName) {
          if (newName.length() > 0) {
               this.name = newName;
          }
     }
     // “getters”
     public String getName() {
          return this.name;
     }
     public double getBalance() {
          return this.balance;
     }
}
```

Getters & Setters

```
// BankAccountProgram.java
BankAccount dukeAccount = new BankAccount(“Duke”, 50);

dukeAccount.setName(“Duke J.”);
String name = dukeAccount.getName();
double balance = dukeAccount.getBalance();

println(name + “ has $” + balance);    // “Duke J. has $50”
```

One Special Method...

How can we do this?

```
BankAccount ba = new BankAccount(...);
println(ba);  // ba isn't a String!
```

One Special Method...

How can we do this?

```
BankAccount ba = new BankAccount(...);
println(ba);  // ba isn't a String!
```

We define a **`toString()`** method
(inside the class file)

```
public String toString() {
    return this.name
           + " has $" + this.balance;
}
```

One Special Method...

How can we do this?

```
BankAccount ba = new BankAccount(...);
println(ba);  // ba isn't a String!
```

We define a **`toString()`** method
(inside the class file)

```
public String toString() {
    return this.name
           + " has $" + this.balance;
}
```

And now this works!

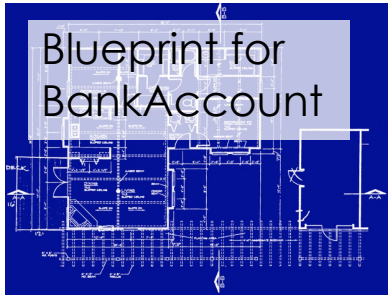
```
BankAccount ba = new BankAccount(...);
println(ba);  // prints "Duke has $50"
```

Classes Are Like Blueprints

BankAccount Class (blueprint)

State: Has name
Has balance

Behavior: Can deposit
Can withdraw



BankAccount #1 (variable)

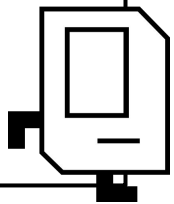
State: **name = "Duke"**
balance = 50

Behavior: Can deposit
Can withdraw

BankAccount #1 (variable)

State: **name = "Karel"**
balance = 1.99

Behavior: Can deposit
Can withdraw



Making a Class ~ 3 Ingredients

1. Define the **variables** each instance stores (state)
2. Define the **constructor** used to make a new instance
3. Define the **methods** you can call on an instance (behaviors)

* all class methods and constructors have access to a `this` reference

Example: BankAccount

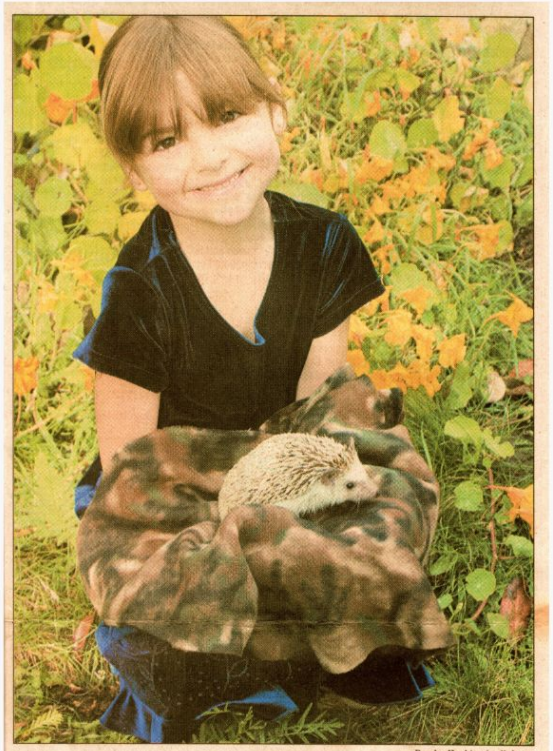
Plan for Today

- Revew: Data Structures
- Classes
- Practice: Hedgehog Show
- Recap

Hedgehog Show

True story: in a town in Washington, there is an annual Hedgehog Show! (ask Sarai)

Hedgehog Show



Bronlea Hawkins /staff photos

Rachel Griffin, 6, holds one of the Pointer sisters, a pair of three-year-old African Hedgehogs owner Carroll Meek adopted. An avid needlepointer, Meek named the sisters PetitPointe and GrossePointe.

It's a hedgehog day

A Hedgehog Gathering III will open at 10 a.m. today, Saturday, Oct. 8, and run until 5 p.m. at the Depot Arts Center and Gallery, 611 R Avenue in Anacortes. Admission is $5 for adults and $3 for seniors and children. Highlights of the show will be an discussion of hedgehog judging standards, as well as an auction and shopping for hedgehog merchandise. A portion of the show proceeds will benefit the Anacortes Community Theater, Depot Arts Center and Gallery, and Hedgehogs Northwest. Visit the Hedgehogs Northwest Web site at www.hhnw.hedgehogcentral.com.

The gathering is like a dog or cat show — but with hedgehogs instead. They are judged in three categories: attitude, health and shape.

Hedgehog Show

- Let's help keep track of hedgehogs at the Hedgehog Show!
- To do that, we'll need a new variable type, **Hedgehog**.
- How would you design a **Hedgehog** variable type?

Hedgehog Show

- Let's help keep track of hedgehogs at the Hedgehog Show!
- To do that, we'll need a new variable type, **Hedgehog**.
- How would you design a **Hedgehog** variable type?

The gathering is like a dog or cat show — but with hedgehogs instead. They are judged in three categories: attitude, health and shape.

- What **state/properties** (instance variables) and **behaviors** (methods) should it have?

Plan for Today

- Revew: Data Structures
- Classes
- Practice: Hedgehog Show
- Recap

A class …

A class defines a new variable type.

Making a Class ~ 3 Ingredients

1. Define the **variables** each instance stores (state)
2. Define the **constructor** used to make a new instance
3. Define the **methods** you can call on an instance (behaviors)

* all class methods and constructors have access to a `this` reference

Plan for Today

- Revew: Data Structures
- Classes
- Practice: Hedgehog Show
- Recap

Next time: Classes Practice