TODO: record lecture

Week03

Today is...
✨ No-Laptop Monday! ✨

- HW03 preview
- functions
- classes
- HW03 behind the scenes
- references

WARMUP
What is a bullet hell?

BONUS
Who wrote the game Everyday Shooter? What is she working on now?

BONUS
What does Touhou 7: Perfect Cherry Blossom's Extra Stage look like? Who wrote this?

but first, something nice

secret agent owl

secret agent owl

secret agent owl

secret agent owl

HW03 preview

functions

anatomy of a function

anatomy of a function (1/2)

```
ReturnType functionName(ArgumentOneType argumentOne, ...) {
    ...
}
```

- a **function** is a lil chunk of code you can call from elsewhere
  - a function takes any number of **arguments**
    - `... foo(int arg) { ... } // function foo takes an int`
  - a function with a non-void **return type** must **return** a value of that type
    - `int bar(...) { ... } // bar returns an int`
    - `void baz(...) { ... } // baz doesn't return anything`

anatomy of a function (2/2)

```
void drawLine(
        double a_x,
        double a_y,
        double b_x,
        double b_y,
        Color color) {
    ...
}
```

return

return (1/2)

- a **return** statement stops execution of a function and returns the program to where the function was called
  - some return statements return a value
    - `return 123;`
  - others do not
    - `return;`
      - ✨ this can be used to stop running a void-returning function in the middle

return (2/2)

- a function with a non-void **return type** must **return** a value of that type, regardless of the path taken through the function

```
Error: missing return statement

static boolean isPrime(int n) {
    if (n <= 1) { return false; }
    for (int i = 2; i <= SQRT(n); ++i) {
        if (n % i == 0) { return false; }
    }
}
```

return (2/2)

- a function with a non-void **return type** must **return** a value of that type, regardless of the path taken through the function

```
static boolean isPrime(int n) {
    if (n <= 1) { return false; }
    for (int i = 2; i <= SQRT(n); ++i) {
        if (n % i == 0) { return false; }
    }
    return true;
}
```

void

void

- **void** is a special return type meaning a function does not return a value
  - void functions often modify (the objects referenced by) their arguments
    - `static void inPlaceReverse(int[] array) { ... }`
      - `// no need to return a reference to array`
        `// (user of the function already has one)`
  - in Java, the **main method** is a void function (it doesn't return anything)
    - `public static void main(String[] arguments) { ... }`

the call stack

the call stack

- functions can call other functions
  - the resulting "stack" of function calls is called the **call stack**

```
class Main {

    static void snap() {
        crackle();
    }

    static void crackle() {
        pop();
    }

    static void pop() {
        return;
    }

    public static void main(String[] arguments) {
        snap();
    }
}
```

Stack | Threads

Method	Line
Main.pop	12
Main.crackle	8
Main.snap	4
Main.main	16
sun.reflect.NativeMethodAccessorImpl.invoke0	-1
sun.reflect.NativeMethodAccessorImpl.invoke	62
sun.reflect.DelegatingMethodAccessorImpl.invoke	43
java.lang.reflect.Method.invoke	498
edu.rice.cs.drjava.model.compiler.JavacCompiler.runCommand	259
sun.reflect.NativeMethodAccessorImpl.invoke0	-1
sun.reflect.NativeMethodAccessorImpl.invoke	62
sun.reflect.DelegatingMethodAccessorImpl.invoke	43
java.lang.reflect.Method.invoke	498
edu.rice.cs.dynamicjava.symbol.JavaClass$JavaMethod.evalu...	362
edu.rice.cs.dynamicjava.interpreter.ExpressionEvaluator.han...	92
edu.rice.cs.dynamicjava.interpreter.ExpressionEvaluator.visit	84
koala.dynamicjava.tree.StaticMethodCall.acceptVisitor	121
edu.rice.cs.dynamicjava.interpreter.ExpressionEvaluator.value	38
edu.rice.cs.dynamicjava.interpreter.ExpressionEvaluator.value	37
edu.rice.cs.dynamicjava.interpreter.StatementEvaluator.visit	106
edu.rice.cs.dynamicjava.interpreter.StatementEvaluator.visit	29
koala.dynamicjava.tree.ExpressionStatement.acceptVisitor	101

```
sun.reflect.NativeMethodAccessorImpl.invoke0
sun.reflect.NativeMethodAccessorImpl.invoke
sun.reflect.DelegatingMethodAccessorImpl.invoke
java.lang.reflect.Method.invoke
edu.rice.cs.drjava.model.compiler.JavacCompiler.runCommand
sun.reflect.NativeMethodAccessorImpl.invoke0
sun.reflect.NativeMethodAccessorImpl.invoke
sun.reflect.DelegatingMethodAccessorImpl.invoke
java.lang.reflect.Method.invoke
edu.rice.cs.dynamicjava.symbol.JavaClass$JavaMethod.evalu...
edu.rice.cs.dynamicjava.interpreter.ExpressionEvaluator.han...
edu.rice.cs.dynamicjava.interpreter.ExpressionEvaluator.visit
koala.dynamicjava.tree.StaticMethodCall.acceptVisitor
edu.rice.cs.dynamicjava.interpreter.ExpressionEvaluator.value
edu.rice.cs.dynamicjava.interpreter.ExpressionEvaluator.value
edu.rice.cs.dynamicjava.interpreter.StatementEvaluator.visit
edu.rice.cs.dynamicjava.interpreter.StatementEvaluator.visit
koala.dynamicjava.tree.ExpressionStatement.acceptVisitor
```

[let's see what Eclipse does]

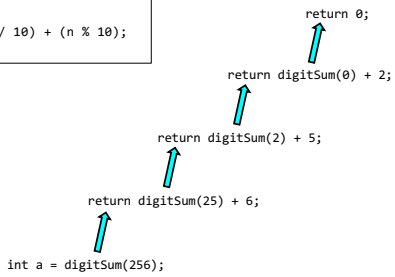
recursion

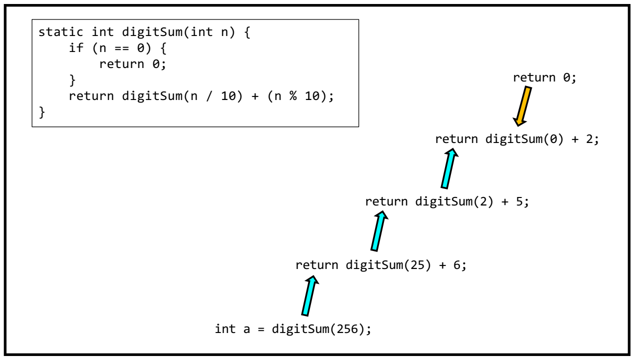
recursion (1/2)

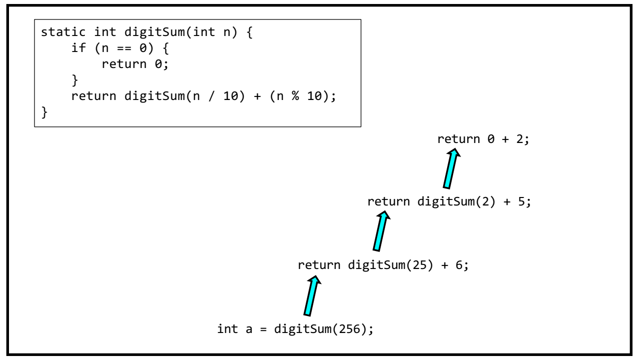
- a **recursive function** is a function that calls itself
  - each call must make progress towards a **base case** (when the function finally returns without calling itself)
    - ✨ when in doubt, try something like zero for your base case

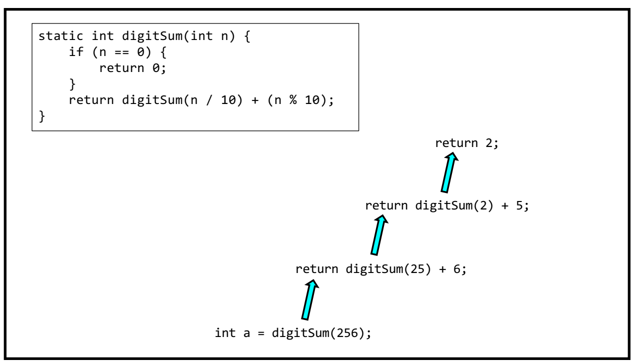
```
class Main extends Cow {
    static int digitSum(int n) {
        if (n == 0) {
            return 0;
        }
        return digitSum(n / 10) + (n % 10);
    }

    public static void main(String[] arguments) {
        PRINT(digitSum(256)); // 13
    }
}
```

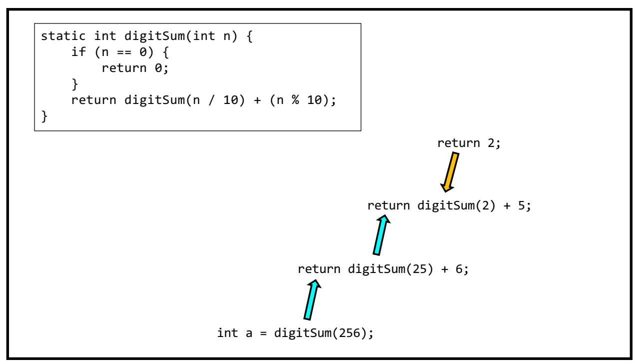
```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

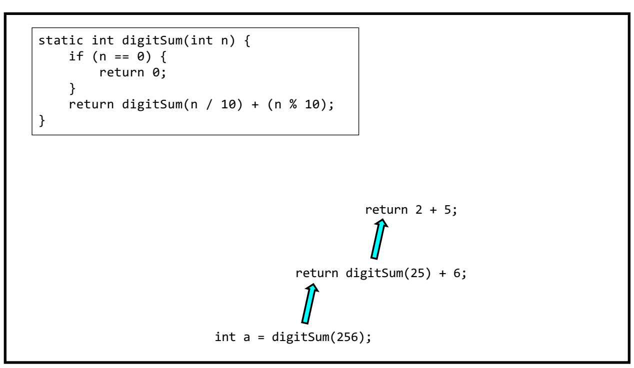


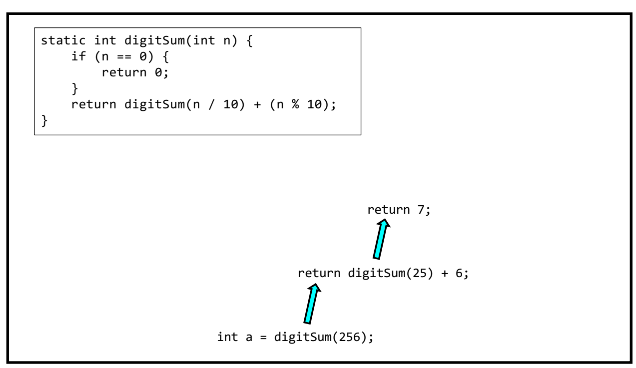

static int digitSum(int n) {
if (n == 0) {
return 0;
}
return digitSum(n / 10) + (n % 10);
}
return 0;
return digitSum(0) + 2;
return digitSum(2) + 5;
return digitSum(25) + 6;
int a = digitSum(256);


static int digitSum(int n) {
if (n == 0) {
return 0;
}
return digitSum(n / 10) + (n % 10);
}
return 0 + 2;
return digitSum(2) + 5;
return digitSum(25) + 6;
int a = digitSum(256);

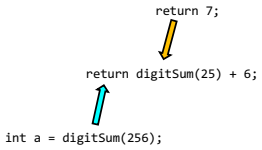

static int digitSum(int n) {
if (n == 0) {
return 0;
}
return digitSum(n / 10) + (n % 10);
}
return 2;
return digitSum(2) + 5;
return digitSum(25) + 6;
int a = digitSum(256);


static int digitSum(int n) {
if (n == 0) {
return 0;
}
return digitSum(n / 10) + (n % 10);
}
return 2;
return digitSum(2) + 5;
return digitSum(25) + 6;
int a = digitSum(256);

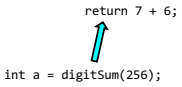

static int digitSum(int n) {
if (n == 0) {
return 0;
}
return digitSum(n / 10) + (n % 10);
}
return 2 + 5;
return digitSum(25) + 6;
int a = digitSum(256);


static int digitSum(int n) {
if (n == 0) {
return 0;
}
return digitSum(n / 10) + (n % 10);
}
return 7;
return digitSum(25) + 6;
int a = digitSum(256);

```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```




```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```




```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```


```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```


```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

int a = 13;

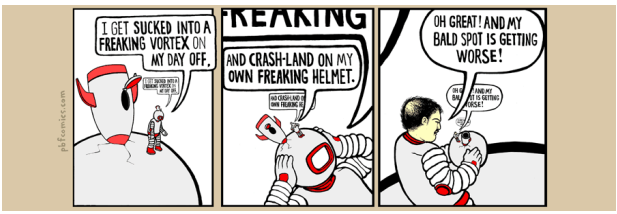
fin.

recursion (2/2)

- a **stack overflow** error happens when functions call too many functions without returning; these errors are usually caused by broken recursive code

```
class Main {
    static void stackOverflow() {
        stackOverflow();
    }

    public static void main(String[] arguments) {
        stackOverflow();
    }
}
```

```
java.lang.StackOverflowError
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
```



classes

anatomy of a class

anatomy of a class (1/2)

```
class ClassName {
    VariableOneType variableOne;
    ...

    FunctionOneReturnType functionOneName(...) { ... }
    ...
}
```

- a **class** is (a blueprint for) a lil chunk of data that you can make elsewhere
  - a class may have any number of **variables** (fields)
    - `int foo; // objects of this class have an int called foo`
  - a class may have any number of **functions** (methods)
    - `int bar() { ... } // objects of class have function bar`

anatomy of a class (2/2)

```
class Thing {
    // instance variables
    double x;
    double y;

    // instance methods
    void draw() { ... }
    ...
}
```

dot

dot

- the **dot** operator is used to access an object's variables and functions

```
Thing thing = new Thing();
thing.x = 3.0;
thing.y = 4.0;
thing.draw();
```

terminology

class vs. object (instance of a class)

- a **class** is NOT the same thing as an **object**
  - a class is "a blueprint for making objects"
  - we can make an **instance of a class** (an **object**) using the **new** keyword
    - this is called "instantiating the class"
    - `Thing thing = new Thing();`

[off the record note on OOP (Object Oriented Programming) terminology]

new and constructors

new

- the **new** keyword create a new instance of a class and calls its appropriate **constructor**
  - `int[] array = new int[5]; // { 0, 0, 0, 0, 0 }`
  - `Color color = new Color(1.0, 0.0, 0.0); // (1.0, 0.0, 0.0)`
  - ☠️ **you don't need `new` to create a new string**
    - `String string = "strings are their own thing";`
  - ☠️ **you don't need `new` to create a new array when using {} syntax**
    - `int[] array = { 1, 2, 3 };`
  - ☠️ **`new` doesn't actually return the *object* it created; it returns *a reference to the object***

constructors (1/2)

- a **constructor** is called when an object is created
  - if the class does not have a constructor, then the **default constructor** must be called, which takes no arguments and sets all variables to zero
    - `Color color = new Color(); // (0.0, 0.0, 0.0)`

constructors (2/2)

- a (non-default) **constructor** is never necessary, but is often convenient

```
Color color = new Color(1.0, 1.0, 1.0); // (r=1.0, g=1.0, b=1.0)
```

```
Color color = new Color(); // (0.0, 0.0, 0.0)
color.r = 1.0;             // (1.0, 0.0, 0.0)
color.g = 1.0;             // (1.0, 1.0, 0.0)
color.b = 1.0;             // (1.0, 1.0, 1.0)
```

this

in Python, `this` is `self`

this (1/2)

- **this** is a reference to the instance of the class whose function we're inside of
  - ✨ especially useful inside a constructor

```
class Color {
    double r;
    double g;
    double b;

    void shade() {
        this.r /= 2;
        this.g /= 2;
        this.b /= 2;
    }

    Color(double r, double g, double b) { // constructor
```