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(Vector2 pointA, Vector2 pointB, Vector3 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;`

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 <= Math.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 <= Math.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 reverseArrayInPlace(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
    - `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
```

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

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

    public static void main(String[] arguments) {
        System.out.println(digitSum(256)); // 13
    }
}
```

```java
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);
```

```java
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);
```

```java
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);
}
```

```
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);
}
```

```
return 7 + 6;
int a = digitSum(256);
```

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

```
return 13;
```

```
int a = digitSum(256);
```

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

```
return 13;
```

```
int a = digitSum(256);
```

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

```
int a = 13;
```

---

fin.

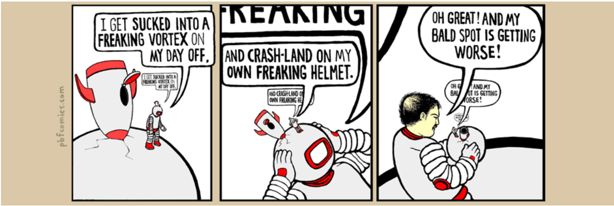
---

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)
      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 Vector2 {
    // instance variables
    double x;
    double y;

    // constructor
    Vector2(double x, double y) { ... }

    // instance methods
    double length() { ... }
    ...
}
```

dot

dot

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

```
Vector2 a = new Vector2();
a.x = 3.0;
a.y = 4.0;
System.out.println(a.length()); // 5.0
```

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"

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 }`
  - `Vector2 a = new Vector2(3.0, 4.0); // (3.0, 4.0)`
  - **you don't need `new` to create a new string**
    - `String string = "strings are their own thing";`
  - **`new` doesn't actually return the object it created; it returns *a reference to the object it created***

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
    - `Vector2 a = new Vector2(3.0, 4.0); // (3.0, 4.0)`
    - `Vector2 b = new Vector2(); // (0.0, 0.0)`

constructors (2/2)

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

```
Vector2 a = new Vector2(3.0, 4.0); // a: (3.0, 4.0)
```

```
Vector2 a = new Vector2(); // a: (0.0, 0.0)
a.x = 3.0; // a: (3.0, 0.0)
a.y = 4.0; // a: (3.0, 4.0)
```

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

```
class Vector2 {
    double x;
    double y;

    Vector2(double x, double y) {
        this.x = x;
        this.y = y;
    }

    Vector2 plus(Vector2 other) {
        return new Vector2(this.x + other.x, this.y + other.y);
    }
    ...
}
```

this (2/2)

- **this** is especially useful inside a constructor (elsewhere it's usually optional)

```
class Vector2 {
    double x;
    double y;

    Vector2(double x, double y) {
        this.x = x;
        this.y = y;
    }

    Vector2 plus(Vector2 other) {
        return new Vector2(x + other.x, y + other.y);
    }
    ...
}
```

Java memory model (references to objects)

references and null

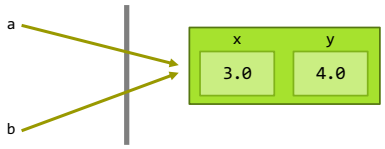
references (1/2)

- in Java, we deal with **references to objects**
  - `int[] foo; // foo is a reference to an int array`
  - `String bar; // bar is a reference to a String object`
  - `Vector2 a; // a is a reference to a Vector2 object`

references (2/2)

- ☠ **you can (unintentionally) make multiple reference to one object**

```
Vector2 a = new Vector2(3.0, 4.0);
Vector2 b = a;
// a and b are references to the same Vector2 object!
```



null

- a **null** reference doesn't refer to any object
  - `Vector2[] array = new Vector2[3]; // { null, null, null }`
  - `Vector2 a = null;`

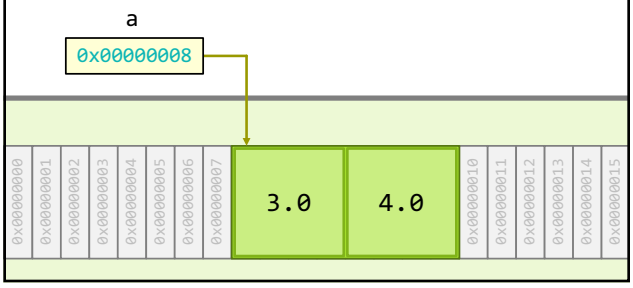
the heap

a reference is a memory address
a memory address is an integer

the heap

- objects made using new are stored "on **the heap**"
  - **the heap** is a huuuuuge chunk of memory
    - objects have an **address** in that memory (that says where they live)

```
Vector2 a = new Vector2(3.0, 4.0);
```



references (1/2)

- **= assigns** the **value** on the right-hand side to the variable on the left-hand side

```
int a = 3;
int b = a;
```

a	b
3	3

references (1/2)

- **= assigns** the **value** on the right-hand side to the variable on the left-hand side

```
int a = 3;
int b = a;
a = 7;
```

a	b
7	3

references (2/2)

- ☠️ **new doesn't actually return the object it created; it returns *a reference to the object it created***
  - a **reference** is an object's location in memory
    - this reference is a **number** (it acts like an `int`)!

```
int a = 3;
int b = a;
```

a	b
3	3

```
Foo a = new Foo();
Foo b = a;
```

a	b
0x7aae6227	0x7aae6227

references (2/2)

- ☠️ **new doesn't actually return the object it created; it returns *a reference to the object it created***
  - a **reference** is an object's location in memory
    - this reference is a **number** (it acts like an `int`)!

```
int a = 3;
int b = a;
```

a	b
3	3

```
Foo a = new Foo();
Foo b = a;
a = null;
```

a	b
null	0x7aae6227

references (3/2)

- i told a small lie last time we talked
  - `System.out.println(array);` does NOT actually print `array`'s memory address
    - rather, it prints a number called `array`'s "hash code"
      - (the hash code is typically calculated *using the memory address*)
        - 🤷
  - the important point is that a **reference** is a **memory address**
    - and a **memory address** is an **integer** (often written in hex) 😀 👍

static

static variables and static methods

instance variables vs. static variables

- an **instance variable** is part of an instance of a class
- a **static variable** (class variable) is part of the class itself
  - there is only one, period. (it's a global variable that lives "on the class")

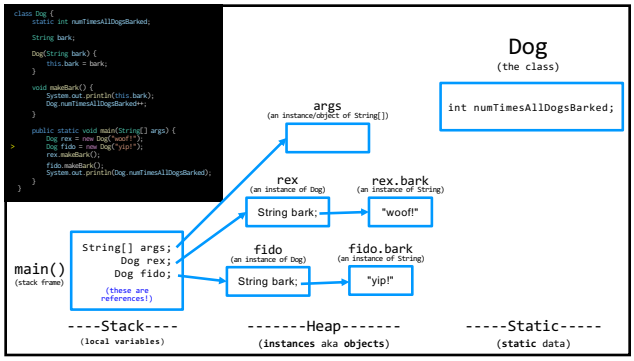
```
class Vector3 {
    double x;
    double y;
    double z;
    static Vector3 red = new Vector3(1.0, 0.0, 0.0);
    ...
}

// drawLine(a, b, Vector3.red);
```

instance methods vs. static methods

- an **instance method** must be called on an instance (object) of a class
- a **static method** (class method) can be called on the class itself
  - ✅ there is no `this` in a static method

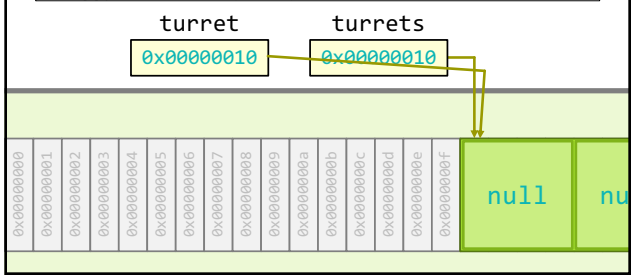
```
class Vector2 {
    double length() { ... }; // (non-static method)
    static double distanceBetween(Vector2 a, Vector2 b) { ... }
    ...
}

// a.length();
// Vector2.distanceBetween(a, b);
```

example

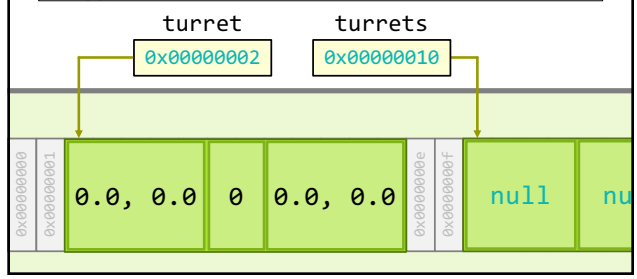


example

```
turrets = new Turret[100];
for (int i = 0; i < turrets.length; i++) { // consider iteration i <- 0
    Turret turret = turrets[i];
    turret = new Turret();
}
turrets[0].position = new Vector2(40.0, 0.0);
```




```
turrets = new Turret[100];
for (int i = 0; i < turrets.length; i++) { // consider iteration i <- 0
    Turret turret = turrets[i];
    turret = new Turret();
}
turrets[0].position = new Vector2(40.0, 0.0);
```



stack trace

- a **stack trace** is the state of the call stack (all the functions that got called and haven't returned yet), usually printed when your program crashes

```
java.lang.NullPointerException
    at HW03.setup(HW03.java:70)
    at App.paintComponent(Low.java:397)
    at javax.swing.JComponent.paint(JComponent.java:1056)
    at javax.swing.JComponent.paintChildren(JComponent.java:889)
    at javax.swing.JComponent.paint(JComponent.java:1065)
    at javax.swing.JComponent.paintChildren(JComponent.java:889)
    at javax.swing.JComponent.paint(JComponent.java:1065)
    at javax.swing.JLayeredPane.paint(JLayeredPane.java:586)
    at javax.swing.JComponent.paintChildren(JComponent.java:889)
    at javax.swing.JComponent.paintToOffscreen(JComponent.java:5217)
    at javax.swing.RepaintManager$PaintManager.paintDoubleBuffered(RepaintManager.java:1579)
    at javax.swing.RepaintManager$PaintManager.paint(RepaintManager.java:1502)
    at javax.swing.RepaintManager.paint(RepaintManager.java:1272)
    at javax.swing.JComponent.paint(JComponent.java:1062)
    at java.awt.GraphicsCallback$PaintCallback.run(GraphicsCallback.java:39)
    at sun.awt.SunGraphicsCallback.runOneComponent(SunGraphicsCallback.java:79)
    at sun.awt.SunGraphicsCallback.runComponents(SunGraphicsCallback.java:116)
    at java.awt.Container.paint(Container.java:1978)
    at java.awt.Window.paint(Window.java:3905)
    at javax.swing.RepaintManager$4.run(RepaintManager.java:842)
    at javax.swing.RepaintManager$4.run(RepaintManager.java:814)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.security.ProtectionDomain$JavaSecurityAccessImpl.doIntersectionPrivilege(ProtectionDomain.java:76)
    at javax.swing.RepaintManager.paintDirtyRegions(RepaintManager.java:814)
    at javax.swing.RepaintManager.paintDirtyRegions(RepaintManager.java:789)
    at javax.swing.RepaintManager.prePaintDirtyRegions(RepaintManager.java:738)
```

stack trace

- a **stack trace** is the state of the call stack (all the functions that got called and haven't returned yet), usually printed when your program crashes

```
java.lang.NullPointerException
    at HW03.setup(HW03.java:70)
```

how did the starter code come to be?

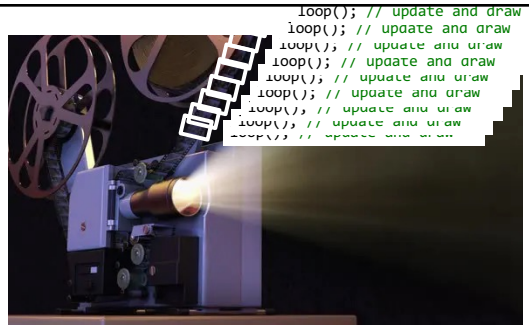
run the starter code

delete the starter code

get familiar with App

`setup()` runs once at the beginning
`loop()` runs over and over, once per **frame**

during a **frame** (like a frame of a movie) we **update** and **draw** the world



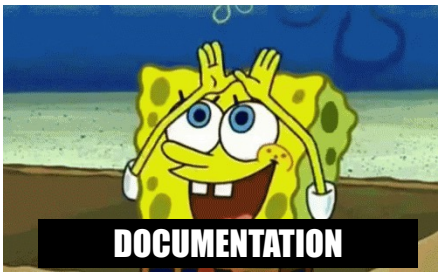
draw a circle for the player

get something (anything) on the screen ☺

move the player around

the player's position must "persist across frames"
the simplest approach is to make `playerPosition` an instance variable of the `HW03` class

we will initialize it in `setup()` and update it in `loop()`
keyboard input is explained in the ✨Documentation✨



make Player a class

optional, but seems reasonable in Java
(would have been okay to skip straight to this step; i tend not to)

make a Turret class

keep it simple!--make it similar to Player

write usage code for how we'll create and update the bullets

since it's Week-03, we'll store a bunch of "slots" for bullets in a huuuge array
i'll call currently unused slots "not alive" ("dead")
to fire a bullet, find the first empty slot in the bullets array and make that bullet `alive` 🔬