EECS 1710
Programming for Digital Media

Lecture 14 :: Objects 2

YORK UNIVERSITÉ UNIVERSITY

Useful Types

Built-in to processing:

- PVector // 2D/3D vectors
- PShape // shape objects
- ArrayList, IntList, FloatList, StringList // dynamic arrays
- BufferedReader // file IO
- PImage // image data

Imported from libraries:

- Other types (need to import from library to use)
  - sound → AudioIn, AudioSample, SoundFile
  - video

PVector

Class Name

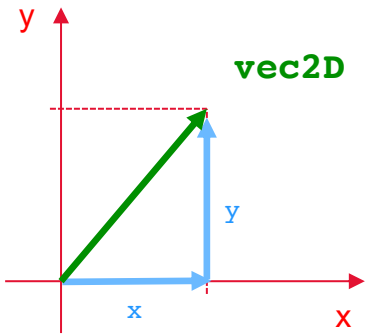
PVector

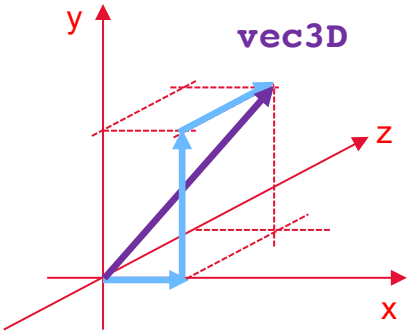
Description

A class to describe a two or three dimensional vector, specifically a Euclidean (also known as geometric) vector. A vector is an entity that has both magnitude and direction. The datatype, however, stores the components of the vector (x,y for 2D, and x,y,z for 3D). The magnitude and direction can be accessed via the methods `mag()` and `heading()`.

In many of the Processing examples, you will see `PVector` used to describe a position, velocity, or acceleration. For example, if you consider a rectangle moving across the screen, at any given instant it has a position (a vector that points from the origin to its location), a velocity (the rate at which the object's position changes per time unit, expressed as a vector), and acceleration (the rate at which the object's velocity changes per time unit, expressed as a vector). Since vectors represent groupings of values, we cannot simply use traditional addition/multiplication/etc. Instead, we'll need to do some "vector" math, which is made easy by the methods inside the `PVector` class.

PVector → constructors





Constructors

```
PVector()
PVector(x, y, z)
PVector(x, y)
```

Fields

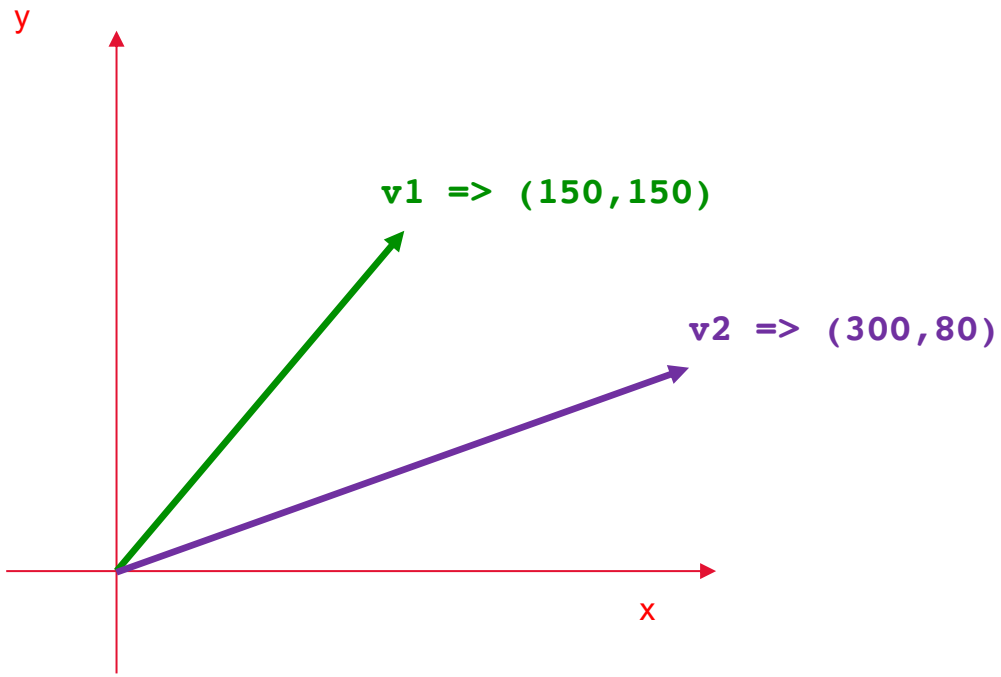
- `x` The x component of the vector
- `y` The y component of the vector
- `z` The z component of the vector

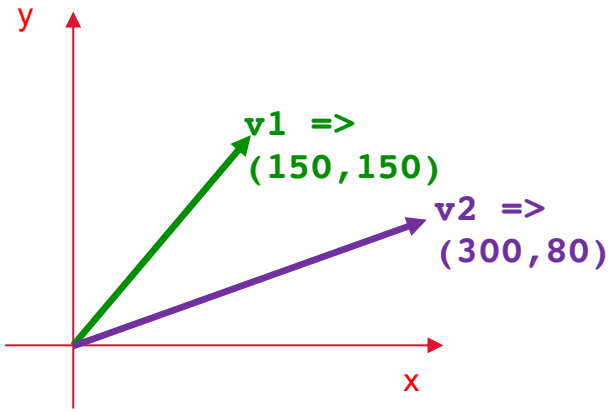
Parameters

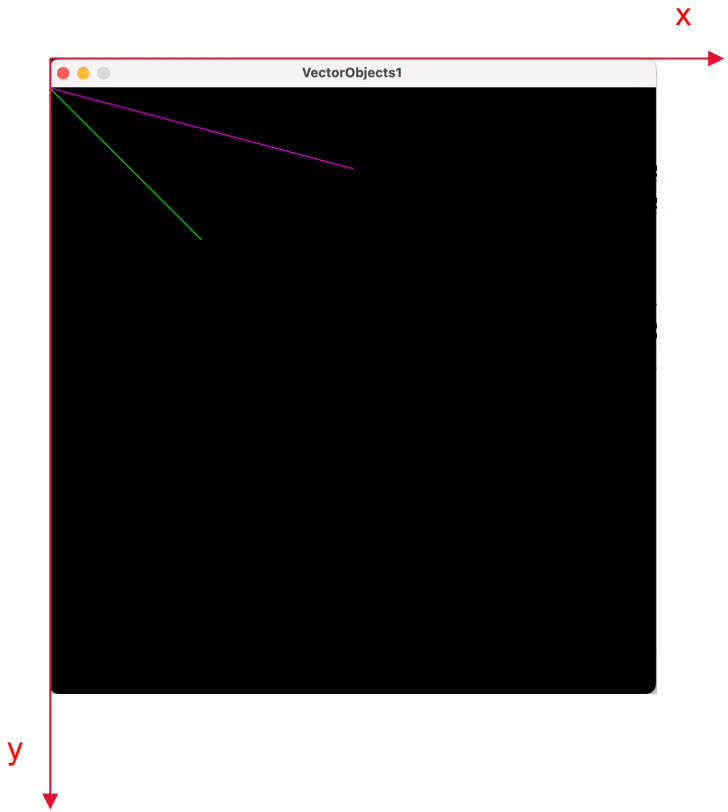
- **x** the x coordinate.
- **y** the y coordinate.
- **z** the z coordinate.

YORK UNIVERSITÉ UNIVERSITY

PVector → constructors








```
final int GREEN = color(0,255,0);
final int PURPLE = color(255,0,255);

PVector v1;
PVector v2;

void setup() {
  size(600, 600);
  background(0,0,0);

  v1 = new PVector(150, 150);
  v2 = new PVector(300, 80);









}
```

YORK UNIVERSITÉ UNIVERSITY

PVector → accessing fields?

Constructors

```
PVector()
PVector(x, y, z)
PVector(x, y)
```

Fields

x	The x component of the vector
y	The y component of the vector
z	The z component of the vector

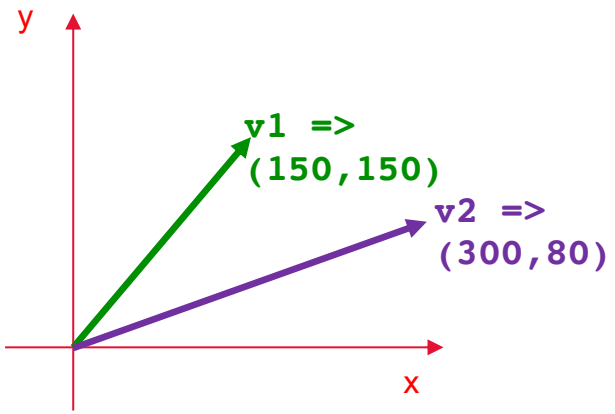
use dot syntax

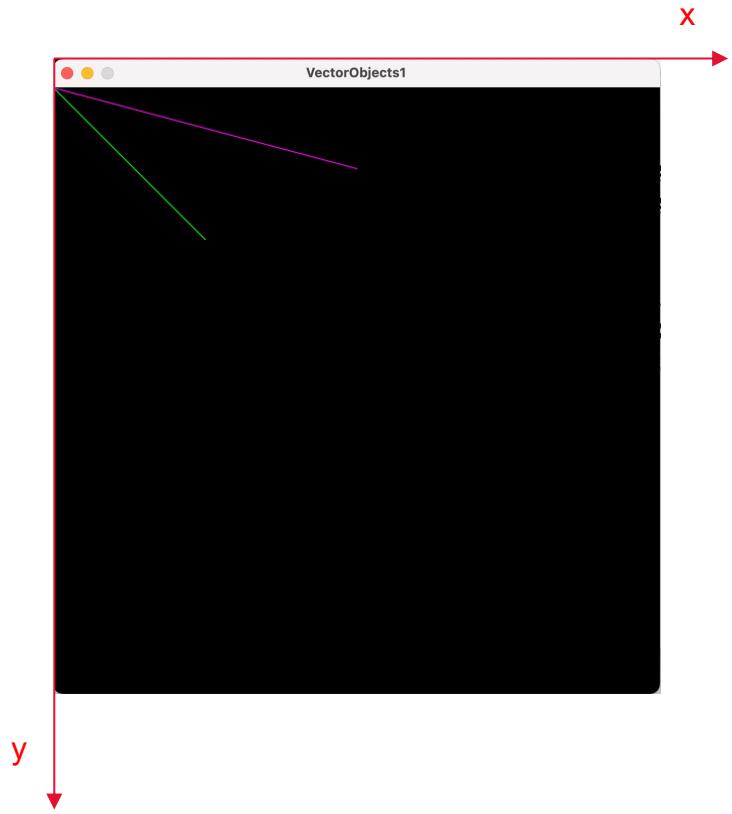
Parameters

x	the x coordinate.
y	the y coordinate.
z	the z coordinate.

- Accessing a field
  `reference.field`
- Invoking methods
  `reference.method(…)`

YORK UNIVERSITÉ UNIVERSITY






```
final int GREEN = color(0,255,0);
final int PURPLE = color(255,0,255);

PVector v1;
PVector v2;

void setup() {
  size(600, 600);
  background(0,0,0);

  v1 = new PVector(150, 150);
  v2 = new PVector(300, 80);

  stroke(GREEN);
  line(0, 0, v1.x, v1.y);
  stroke(PURPLE);
  line(0, 0, v2.x, v2.y);


}
```

YORK UNIVERSITÉ UNIVERSITY

PVector → methods (behaviours)

Constructors

```
PVector()
PVector(x, y, z)
PVector(x, y)
```

Fields

`x`	The x component of the vector
`y`	The y component of the vector
`z`	The z component of the vector

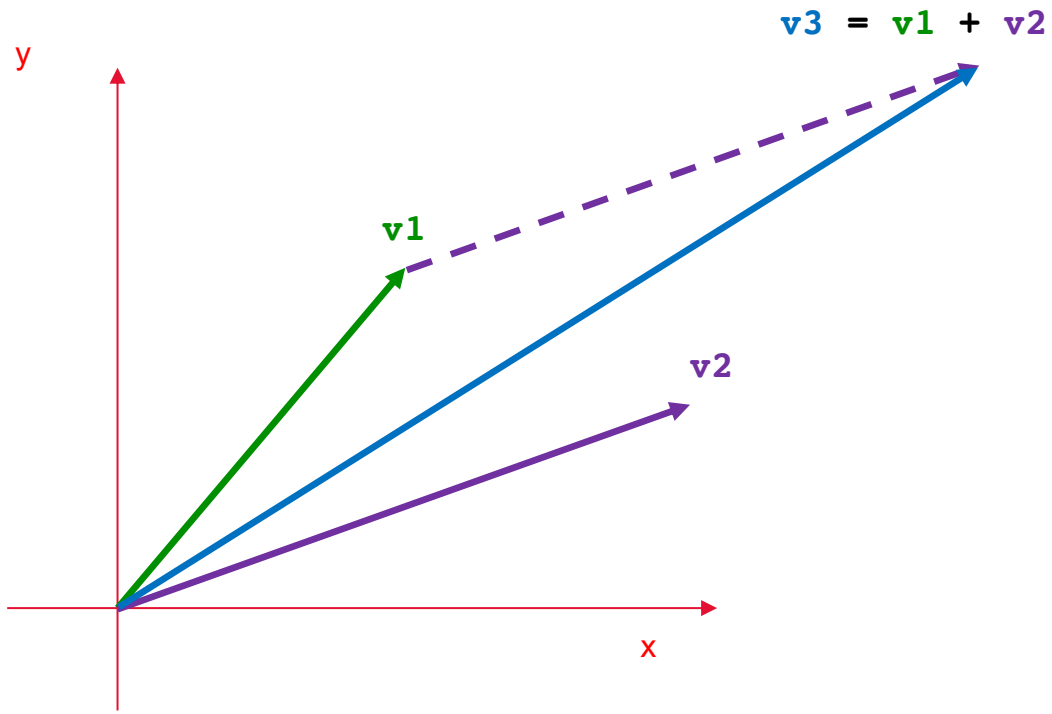
Parameters

x	the x coordinate.
y	the y coordinate.
z	the z coordinate.

Methods

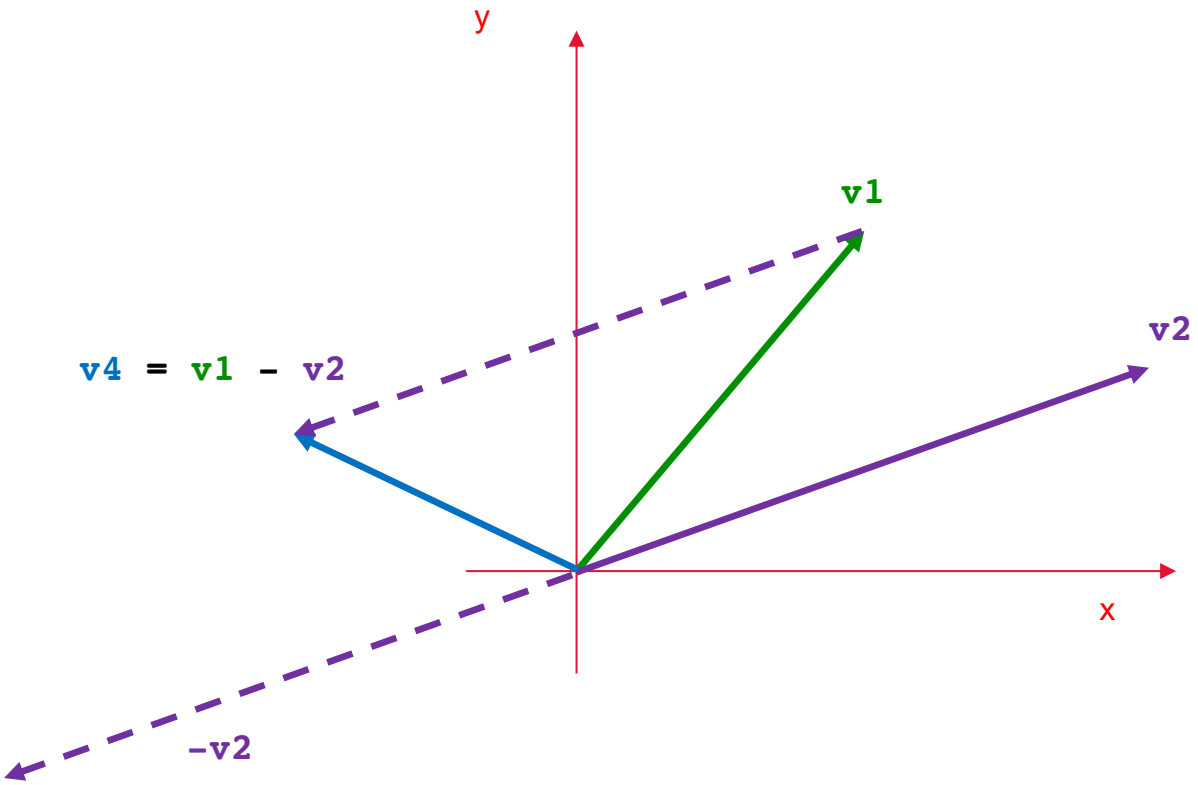
`set()`	Set the components of the vector
`random2D()`	Make a new 2D unit vector with a random direction
`random3D()`	Make a new 3D unit vector with a random direction
`fromAngle()`	Make a new 2D unit vector from an angle
`copy()`	Get a copy of the vector
`mag()`	Calculate the magnitude of the vector
`magSq()`	Calculate the magnitude of the vector, squared
`add()`	Adds x, y, and z components to a vector, one vector to another, or two independent vectors
`sub()`	Subtract x, y, and z components from a vector, one vector from another, or two independent vectors
`mult()`	Multiply a vector by a scalar
`div()`	Divide a vector by a scalar
`dist()`	Calculate the distance between two points
`dot()`	Calculate the dot product of two vectors
`cross()`	Calculate and return the cross product
`normalize()`	Normalize the vector to a length of 1
`limit()`	Limit the magnitude of the vector
`setMag()`	Set the magnitude of the vector
`heading()`	Calculate the angle of rotation for this vector
`rotate()`	Rotate the vector by an angle (2D only)
`lerp()`	Linear interpolate the vector to another vector
`angleBetween()`	Calculate and return the angle between two vectors
`array()`	Return a representation of the vector as a float array

Pvector → adding vectors (add)



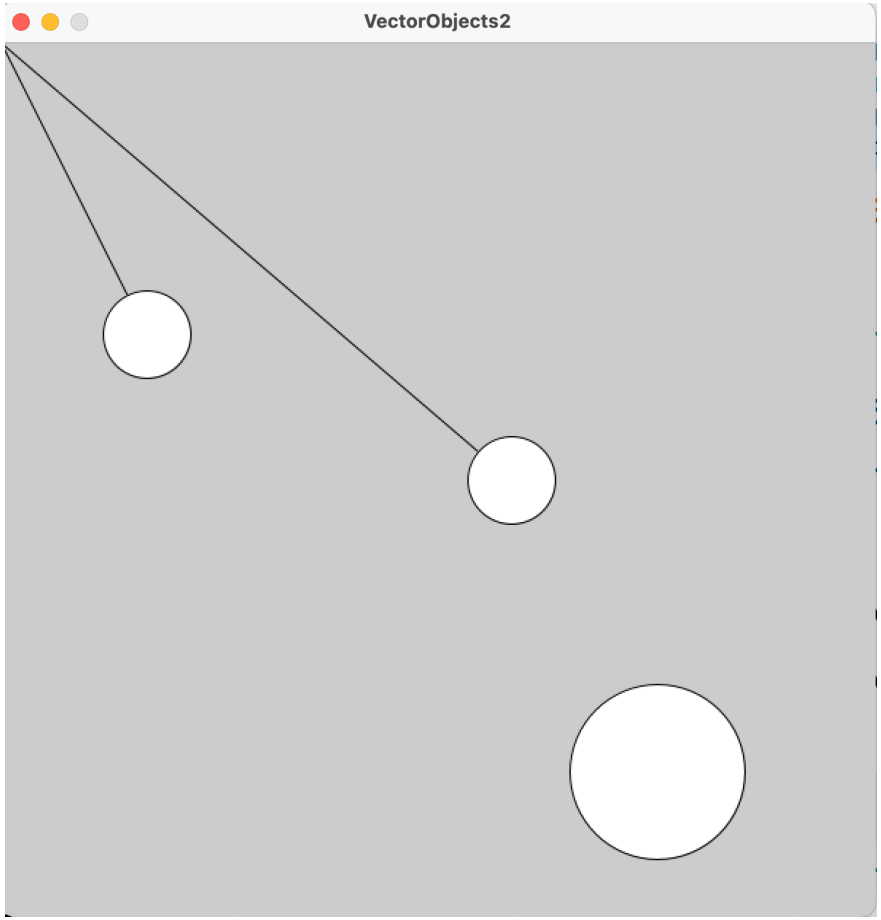

```
final int WHITE = color(255,255,255);
final int GREEN = color(0,255,0);
final int PURPLE = color(255,0,255);

PVector v1;
PVector v2;

void setup() {

  size(600, 600);
  background(0,0,0);

  v1 = new PVector(150, 150);
  v2 = new PVector(300, 80);

  stroke(GREEN);
  line(0, 0, v1.x, v1.y);
  stroke(PURPLE);
  line(0, 0, v2.x, v2.y);

  PVector v3 = v1.add(v2);
  stroke(WHITE);
  line(0,0,v3.x,v3.y);

}
```

PVector → subtracting vectors (sub)



Using PVector (to store ellipse sizes)

```
PVector v1, v2;  // declare

void setup() {
  noLoop();       // stop processing from continuously running draw
  v1 = new PVector(40, 20);        // instantiate
  v2 = new PVector(25, 50);        // instantiate
}

void draw() {

  ellipse(v1.x, v1.y, 12, 12);     // use vectors to define each ellipse
  ellipse(v2.x, v2.y, 12, 12);

  v2.add(v1);
  ellipse(v2.x, v2.y, 24, 24);     // create bigger ellipse from others

}
```

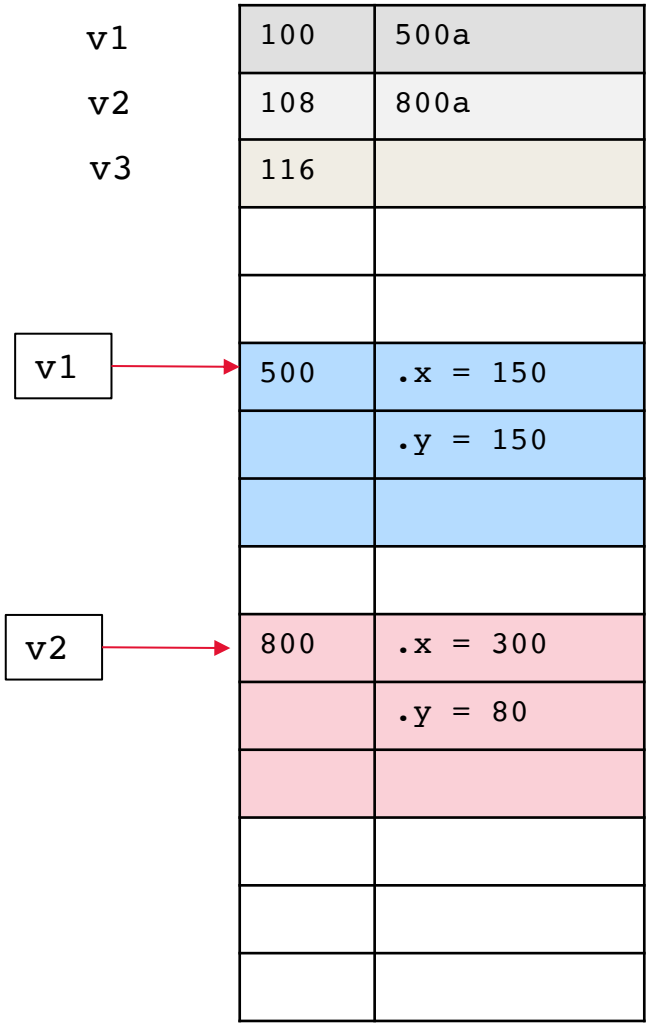


```
// SimpleProjectileMotionVectors.pde (from lecture 6) <= REVISITED

final float GRAVITY = 9.8f;    // 9.8 m/s (in the direction of ground +y)
float mag = sqrt(2*pow(50, 2));

PVector v0;                    // start position
PVector v1;                    // launch vector
float t  = 0f;

void setup() {
  size(800, 800);
  v0 = new PVector(100f, 400f);
  println(v0);
  v1 = new PVector(50f, -50f);
  println(v1);
}

void draw() {
  background(255, 255, 255);
  t+= 0.05;

  float x = v0.x + v1.x*t ;                          // calc. new x
  float y = v0.y + v1.y*t + 0.5*GRAVITY*pow(t, 2) ;  // calc. new y

  line(v0.x, v0.y, v0.x+v1.x, v0.y+v1.y);  // draw v1 at start point (launch vector)
  circle(x, y, 20);                        // draw object (circle) at new x,y
}

void mousePressed() {
  // reposition start time, use all same initial values
  t=0;
}
```

Don't have to use cos() & sin()

Objects Live and Die …

```
final int WHITE = color(255,255,255);
final int GREEN = color(0,255,0);
final int PURPLE = color(255,0,255);

PVector v1;
PVector v2;
PVector v3;

void setup() {

  size(600, 600);
  background(0,0,0);

  v1 = new PVector(150, 150);
  v2 = new PVector(300, 80);



}
```



v1 & v2 are alive… (i.e. instantiated)
v3?

Birth of an object (happens at runtime)

Four steps

- Locate the class

```
(import PVector) => done automatically in Processing
```

- Declare a reference

```
PVector v1;
```

- Instantiate the class

```
new PVector(150, 150);
```

- Assign the reference

```
v1 = new PVector(150, 150);
```

Lets assume we are using PVector objects..

Declaring a PVector variable only creates a reference (not the object itself)

Usually combined

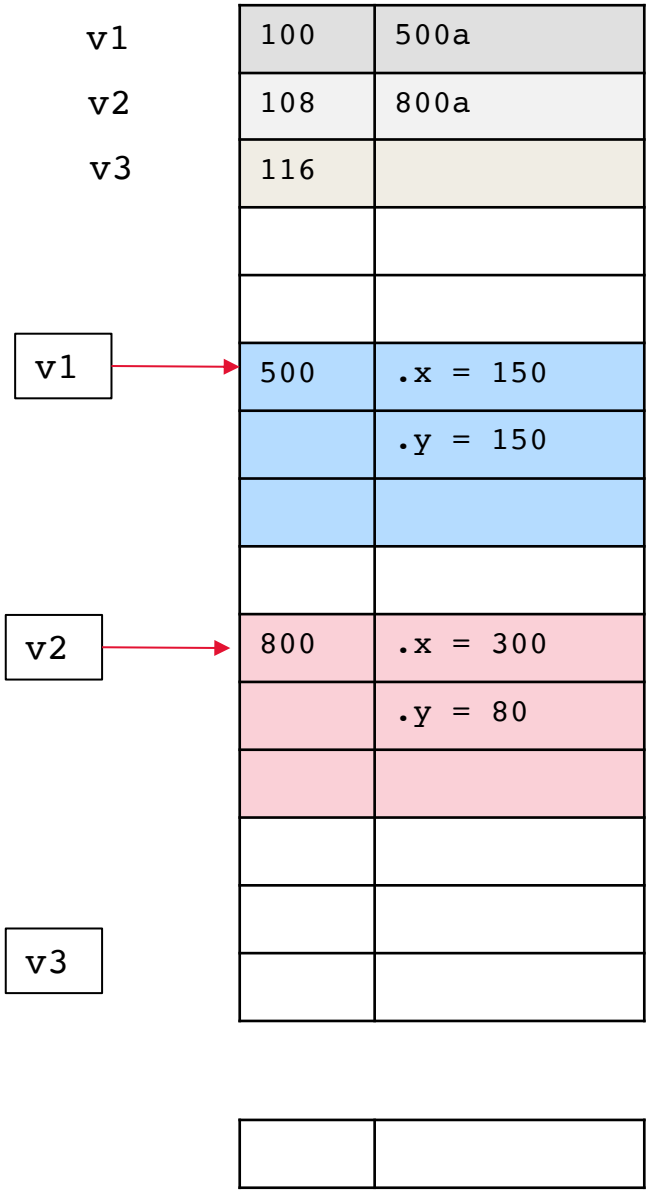
PVector objects have several fields: .x, .y, .z (2D => .z==0)

Summary:

- Variables of primitive types hold **values directly**
- Variables of reference types hold **addresses** of objects (not the objects themselves)

- A class may be **instantiated** using the ***new*** operator along with a ***constructor***
  - The object exists at **runtime only** (not compile time), and is allocated an available slot in memory at runtime
  - If a reference is never assigned an instantiated object, then the program will cause a compile time error

```
final int WHITE = color(255,255,255);
final int GREEN = color(0,255,0);
final int PURPLE = color(255,0,255);

PVector v1;
PVector v2;
PVector v3;

void setup() {

  size(600, 600);
  background(0,0,0);

  v1 = new PVector(150, 150);
  v2 = new PVector(300, 80);



}
```



Aliases

- Many variables can point at the same object:

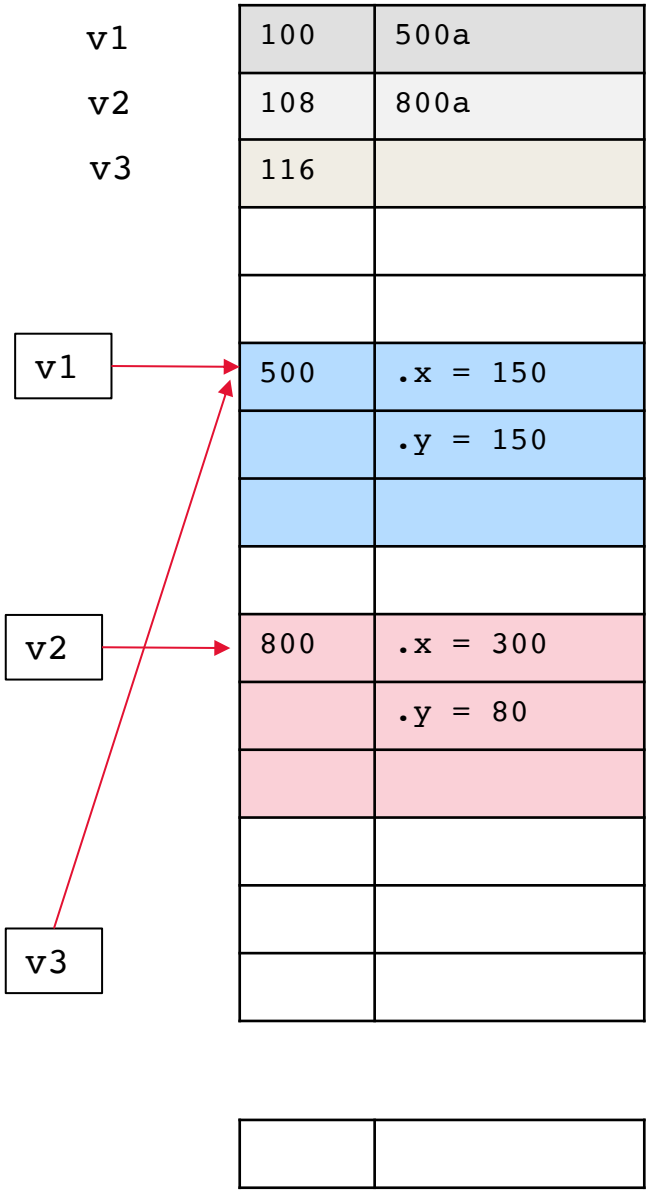
```
v1 = new PVector(150, 150);
v2 = new PVector(300, 80);

PVector v3 = v1.add(v2);   // v3 and v1 both
                           // reference same object
                           // v1 & v3 are ALIASES
```

- If the object is changed through `v1` the change will be seen by `v3`

```
** similarly if v3
changed, v1 will see v3
```

YORK UNIVERSITÉ UNIVERSITY

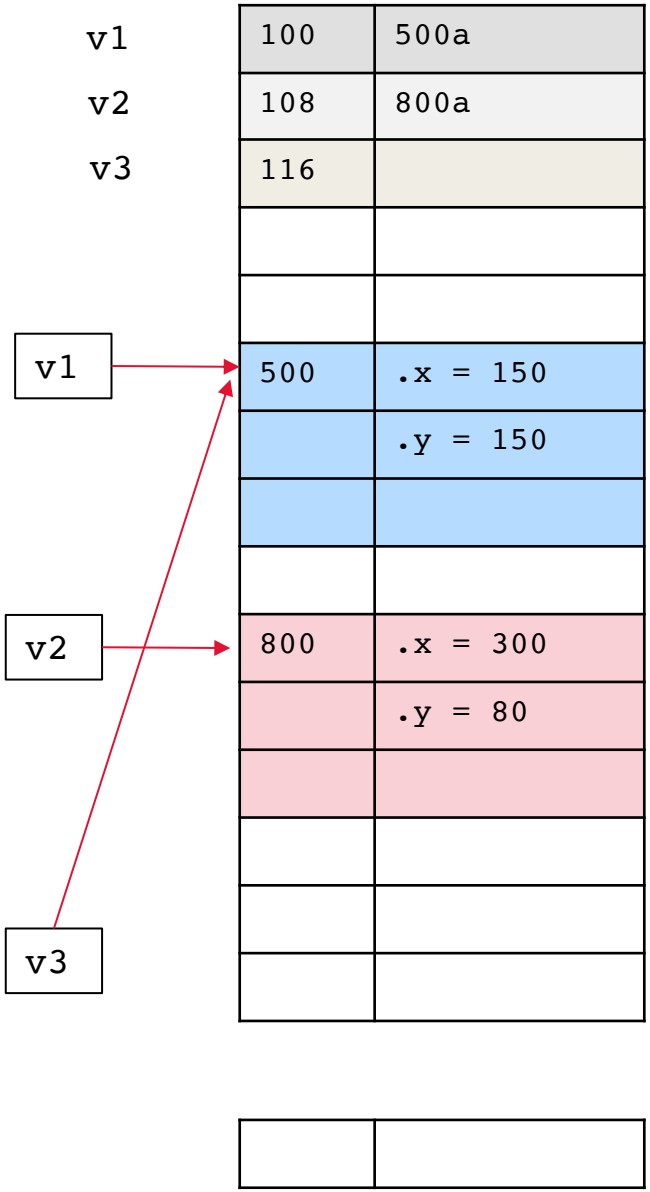
```
final int WHITE = color(255,255,255);
final int GREEN = color(0,255,0);
final int PURPLE = color(255,0,255);

PVector v1;
PVector v2;
PVector v3;

void setup() {

  size(600, 600);
  background(0,0,0);

  v1 = new PVector(150, 150);
  v2 = new PVector(300, 80);
  v3 = v1.add(v2);



}
```



What if an object suddenly has no reference to it?

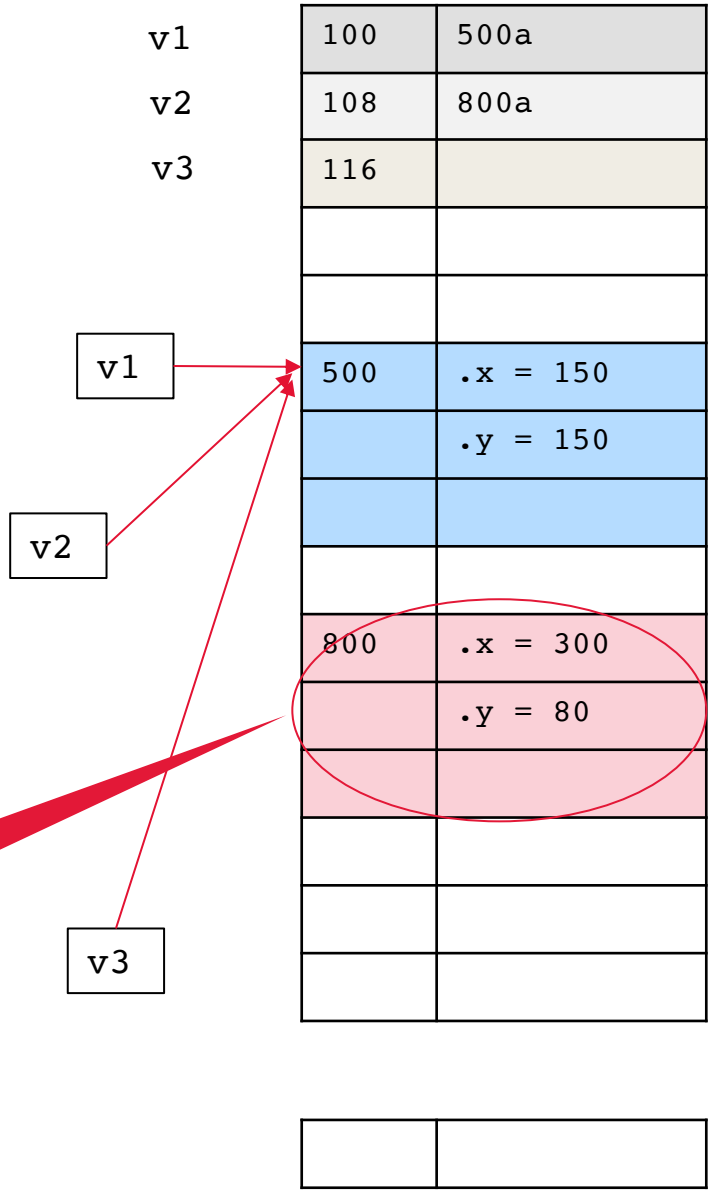
```
v1 = new PVector(150, 150);
v2 = new PVector(300, 80);
v3 = v1.add(v2);
v2 = v1;
```

```
final int WHITE = color(255,255,255);
final int GREEN = color(0,255,0);
final int PURPLE = color(255,0,255);

PVector v1;
PVector v2;
PVector v3;

void setup() {

  size(600, 600);
  background(0,0,0);

  v1 = new PVector(150, 150);
  v2 = new PVector(300, 80);
  v3 = v1.add(v2);



}
```



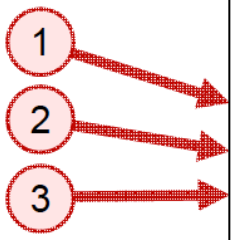
```
final int WHITE = color(255,255,255);
final int GREEN = color(0,255,0);
final int PURPLE = color(255,0,255);

PVector v1;
PVector v2;
PVector v3;

void setup() {

  size(600, 600);
  background(0,0,0);

  v1 = new PVector(150, 150);
  v2 = new PVector(300, 80);
  v3 = v1.add(v2);
  v2 = v1;


}
```

orphaned object (no reference to it – i.e. not accessible)



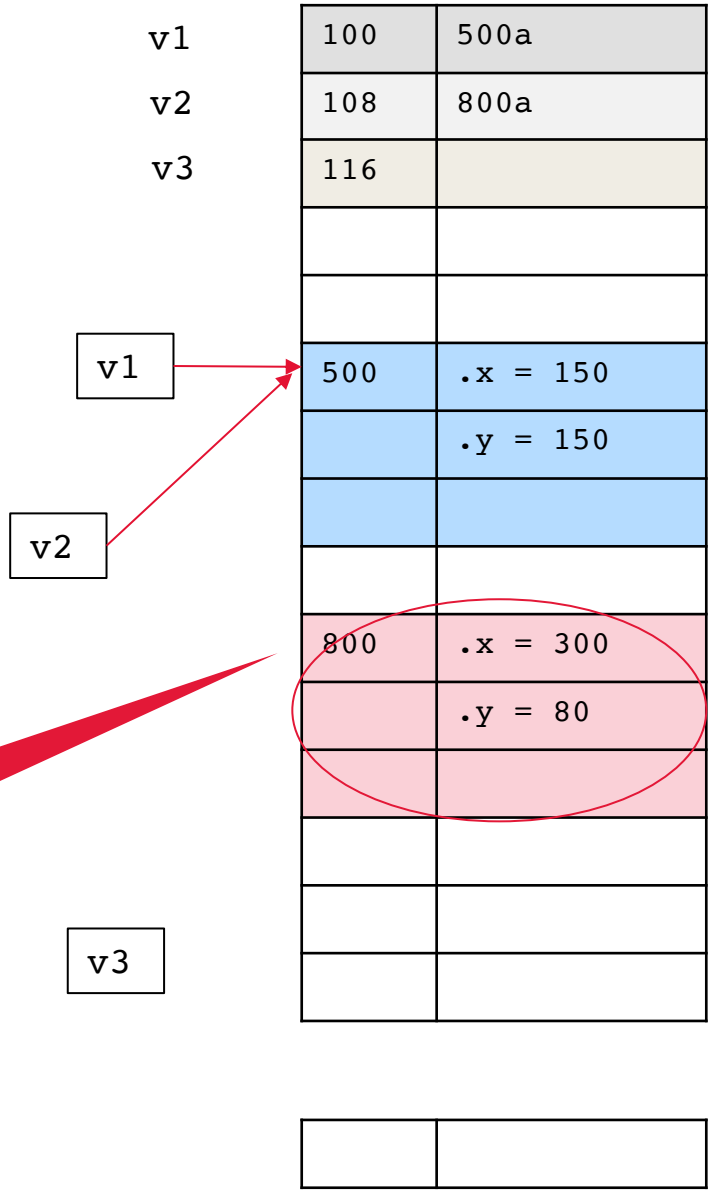
Death of an Object [1]

- The object-reference connection can be destroyed by
  1. No more references to the object (orphaned)
  2. Exiting the scope of the reference
  3. Setting the reference to **null**




```
v1 = new PVector(150, 150);
v2 = new PVector(300, 80);



v2 = v1;
{   PVector v3 = new Pvector(1,1); }
v1 = null;
```

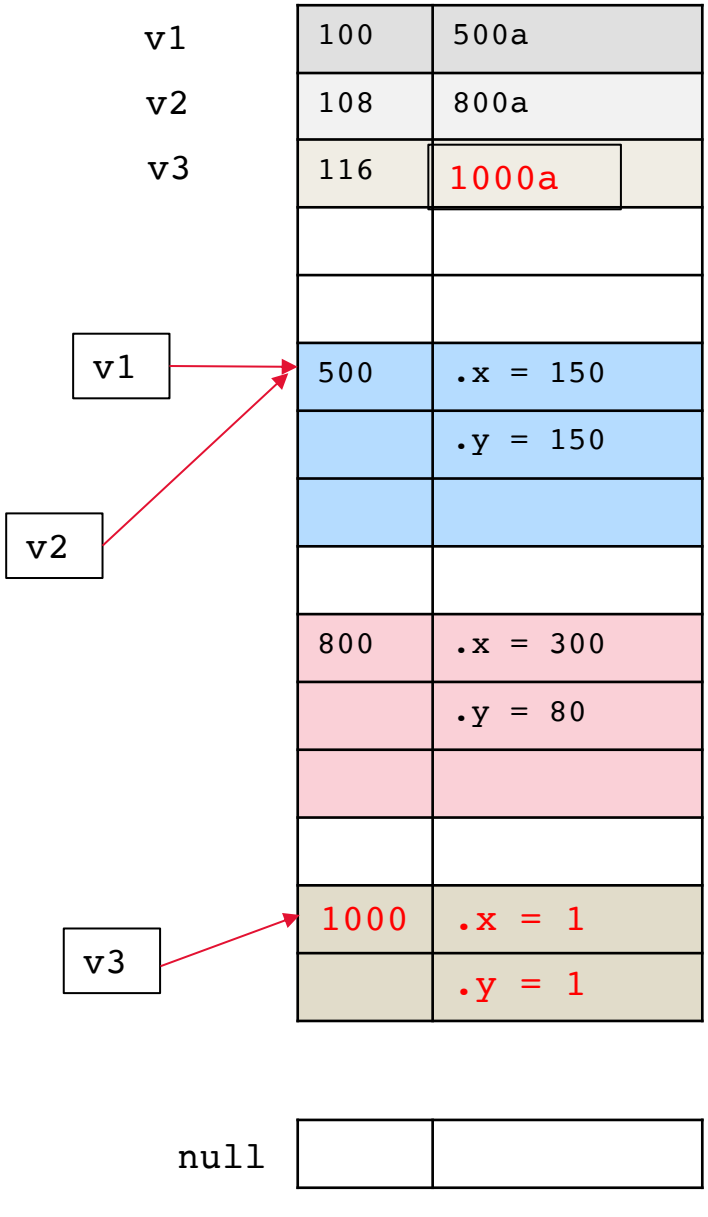
```
final int WHITE = color(255,255,255);
final int GREEN = color(0,255,0);
final int PURPLE = color(255,0,255);

PVector v1;
PVector v2;
PVector v3;

void setup() {

  size(600, 600);
  background(0,0,0);

  v1 = new PVector(150, 150);
  v2 = new PVector(300, 80);
  v3 = v1.add(v2);
  v2 = v1;


}
```



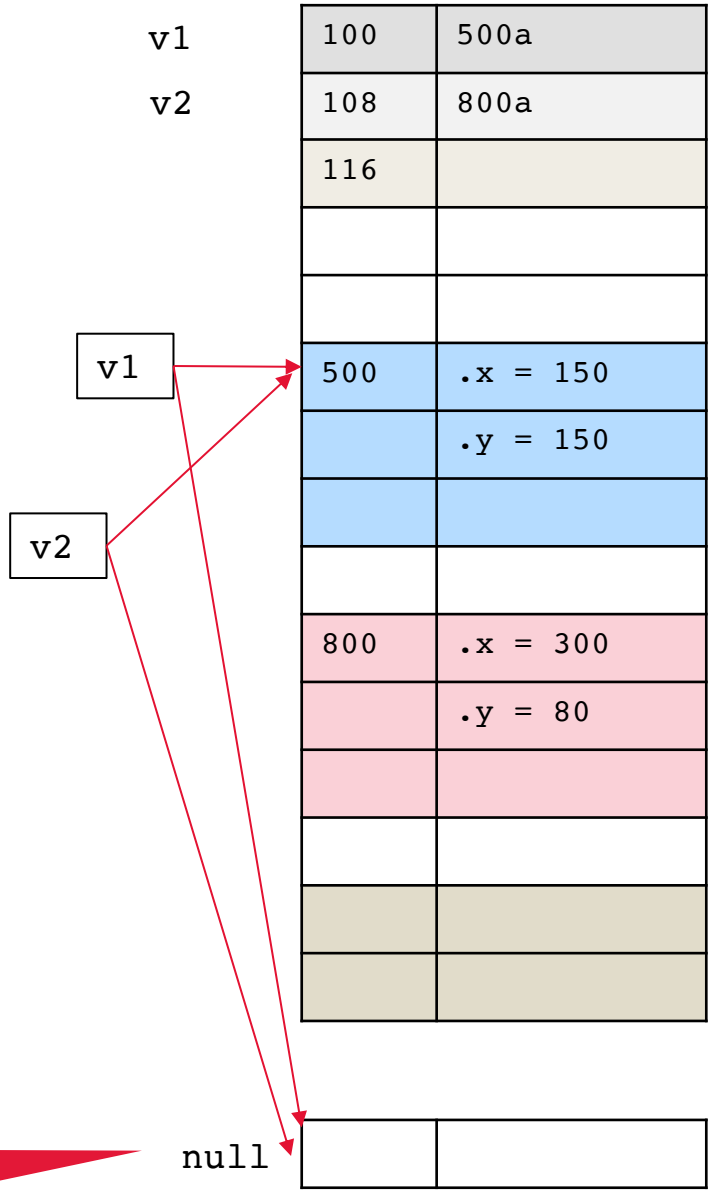
(1) orphaned object
(no more references to it –
i.e. not accessible)

```
final int WHITE = color(255,255,255);
final int GREEN = color(0,255,0);
final int PURPLE = color(255,0,255);

PVector v1;
PVector v2;
// PVector v3;

void setup() {

  size(600, 600);
  background(0,0,0);

  v1 = new PVector(150, 150);
  v2 = new PVector(300, 80);
  // v3 = v1.add(v2);
  v2 = v1;

  {
    PVector v3 = new PVector(1,1);
  }


}
```

(2) v3 went out of scope
(no longer exists)



```
final int WHITE = color(255,255,255);
final int GREEN = color(0,255,0);
final int PURPLE = color(255,0,255);

PVector v1;
PVector v2;
// PVector v3;

void setup() {

  size(600, 600);
  background(0,0,0);

  v1 = new PVector(150, 150);
  v2 = new PVector(300, 80);
  // v3 = v1.add(v2);
  v2 = v1;

  {
    PVector v3 = new PVector(1,1);
  }

  v1 = null;
  v2 = null;

}
```



(3) v1 & v2 set to null

Death of an object [2]

- What is **`null`**?
  - A special address that refers to “no object”
  - Any **reference** type may be assigned `null`
  - May test/output the value of `null`, but not access any object methods/fields (as there are none)
    - `null` is a literal (just like `true` and `false`) whose type is compatible with any non-primitive type.
    - It is OK to print a `null` reference
    - It is *not* OK to invoke methods on a null reference
  - Attempting to access a field/method of an object reference that is currently pointing at **`null`** will cause an exception (resulting in a program crash!)
    - We will look at this more next week