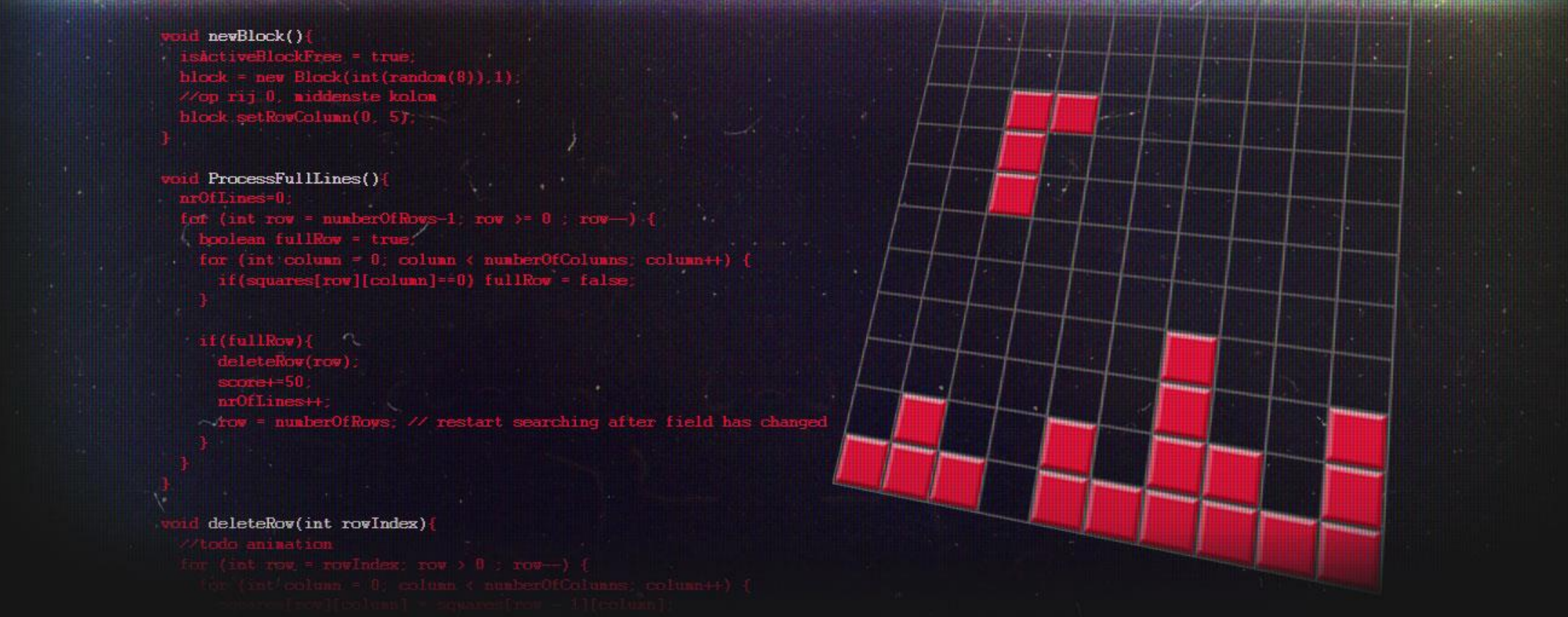


```
void newBlock(){
  isActiveBlockFree = true;
  block = new Block(int(random(8)),1);
  //op rij 0, middenste kolom
  block.setRowColumn(0, 5);
}

void ProcessFullLines(){
  nrOfLines=0;
  for (int row = numberOfRows-1; row >= 0 ; row--) {
    boolean fullRow = true;
    for (int column = 0; column < numberOfColumns; column++) {
      if(squares[row][column]==0) fullRow = false;
    }

    if(fullRow){
      deleteRow(row);
      score+=50;
      nrOfLines++;
      row = numberOfRows; // restart searching after field has changed
    }
  }
}

void deleteRow(int rowIndex){
  //todo animation
  for (int row = rowIndex; row > 0 ; row--) {
    for (int column = 0; column < numberOfColumns; column++) {
      squares[row][column] = squares[row - 1][column];
```

PROGRAMMING FOR ARTISTS II

CLASSES & ARRAYS

SUMMARY: CREATING A CLASS

```
class Circle {

   //data members
   private int xPos, yPos;

   //constructor
   Circle(int xPos, int yPos){
       this.xPos = xPos;
       this.yPos = yPos;
   }

   //member methods
   void display(int circleRadius){
      ellipse(xPos, yPos, circleRadius*2, circleRadius*2);
   }
   //getter method
   int getXPosition(){
       return xPos;
   }
   //setter method
   void setYPosition(int y){
       this.yPos = y;
   }
}
```

CLASS KEYWORD + CLASS NAME

DATA MEMBERS

CONSTRUCTOR

METHODS

SUMMARY: CREATING OBJECTS

STEP 1: DECLARE
Choose a **meaningful** variable **name** for the object.
Use the **class name** for the **data type**

```
Circle myCircle;
```

STEP 2: INITIALIZE → IN SETUP() !
The **constructor** will be called automatically when a new object is created with ***new***

```
myCircle = new Circle(15, 30);
```

Arguments for x and y

STEP 3: USE
Properties and **methods** of an object can be **called**, using the **reference variable**

```
myCircle.display(50);
```

Argument for circleRadius

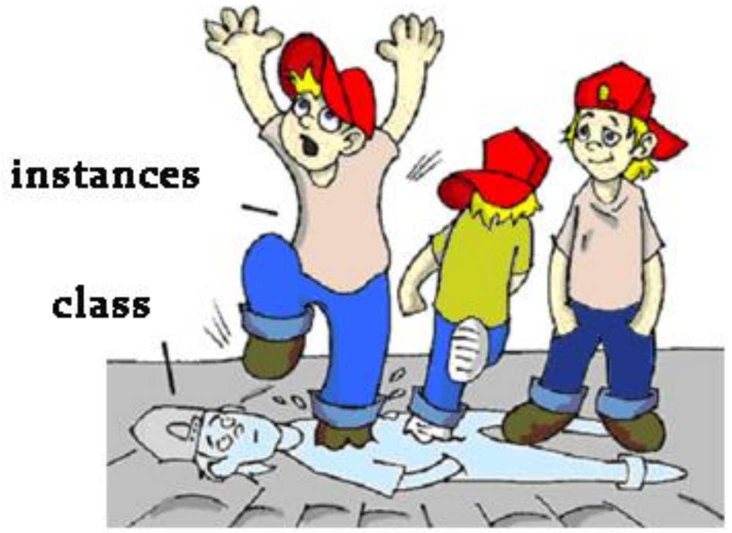
CONSTRUCTOR OVERLOADING

➢ **more than one constructor with different parameters list**

- different number of parameters
- different data types

WHY?
Initialize an object in different ways

HOW?
Two types:
- Default constructor
- Parameterized constructor



CONSTRUCTOR OVERLOADING

```
Student defaultStudent, StudentObj;

void setup(){

  defaultStudent = new Student();

  println("Student Name is: " + defaultStudent.getStuName());
  println("Student Age is: " + defaultStudent.getStuAge());
  println("Student ID is: " + defaultStudent.getStuID());

  //This object creation would call the parameterized constructor
  StudentObj = new Student(555, "Bart Banks", 25);

  println("Student Name is: " + StudentObj.getStuName());
  println("Student Age is: " + StudentObj.getStuAge());
  println("Student ID is: " + StudentObj.getStuID());

}
```

```
Student Name is: New Student
Student Age is: 18
Student ID is: 0
Student Name is: Bart Banks
Student Age is: 25
Student ID is: 555
```

```
class Student {
   private int studentID, studentAge;
   private String studentName;

  //Default constructor
   Student() {
       this.studentID = 0;
       this.studentName = "New Student";
       this.studentAge = 18;
   }

  //Parameterized constructor
   Student(int id, String name, int age) {
       this.studentID = id;
       this.studentName = name;
       this.studentAge = age;
   }

   String getStuName(){
     return studentName;
   }

   int getStuAge(){
     return studentAge;
   }

   int getStuID(){
     return studentID;
   }
}
```

USING THE THIS KEYWORD

USE?

this = reference to the current object

REASON?

to avoid naming conflicts in the method/constructor of your instance/object.

```
class Circle {
   private int xPos, yPos;

   //constructor
   Circle(int xPos, int yPos) {
      this.xPos = xPos;
      this.yPos = yPos;
   }
}
```

Inside the constructor, **xPos** is a **local** copy of the constructor's first argument.

To refer to the Circle field xPos, the constructor must use the keyword **this**.

USING THE THIS KEYWORD WITH A CONSTRUCTOR

Explicit constructor invocation:

→ call another constructor in the same class

The Rectangle class contains a set of constructors:

- the **no-argument constructor** creates a 1x1 rectangle at coordinates (0,0).
- the **two-argument constructor** calls the four-argument constructor, passing in the width and height but always using the 0,0 coordinates.

As before, the compiler determines which constructor to call, based on the number and the type of arguments.

The invocation of another constructor must be the first line in the constructor!

```
class Rectangle {
   private int x, y;
   private int width, height;
   private int speed=1;

   Rectangle() {
      this(0, 0, 1, 1);
   }
   Rectangle(int width, int height, int speed) {
      this(0, 0, width, height);
      this.speed=speed;
   }
   Rectangle(int x, int y, int width, int height) {
      this.x = x;
      this.y = y;
      this.width = width;
      this.height = height;
   }
   ...
}
```

ARRAYS

STEP 1: DECLARE

```
int[] circles = new int[10];
```

STEP 2: INITIALIZE

```
for(int i=0; i < circles.length ; i++){
    circles[i] = random(10,500);
}
```

STEP 3: USE THE VALUES IN THE ARRAY

```
for(int i=0; i < circles.length; i++){
    ellipse(circles[i], circles[i], 30,30);
}
```

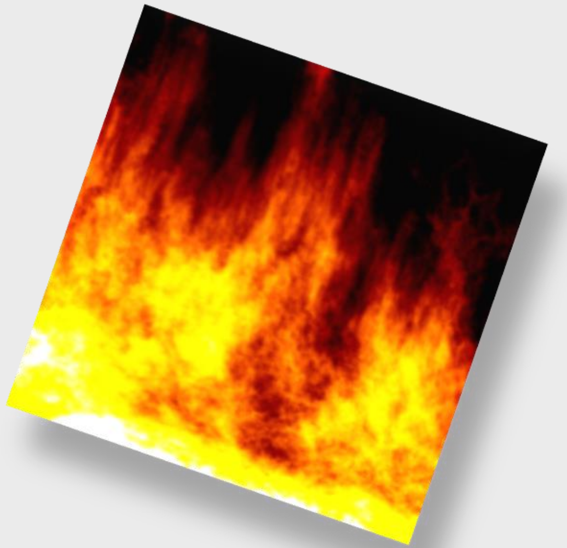
THE PIMAGE OBJECT ARRAY

```
int numFrames = 120, frameIndex=0;
PImage[] imageArr = new PImage[numFrames];
String name;

void setup(){
  size(250,250);
  imageMode(CENTER);

  //fill the array with images
  for(int i=1; i <= imageArr.length; i++){
    if(i<10) name = "Fire00" + i + ".bmp";
    else if(i<100) name = "Fire0" + i + ".bmp";
    else name = "Fire" + i + ".bmp";

    imageArr[i-1] = loadImage(name);
  }
}

void draw(){
  background(0);
  image(imageArr[frameIndex], width/2, height/2);
  frameIndex++;
  frameIndex%=numFrames;
}
```

Properties and predefined methods of the PImage object:

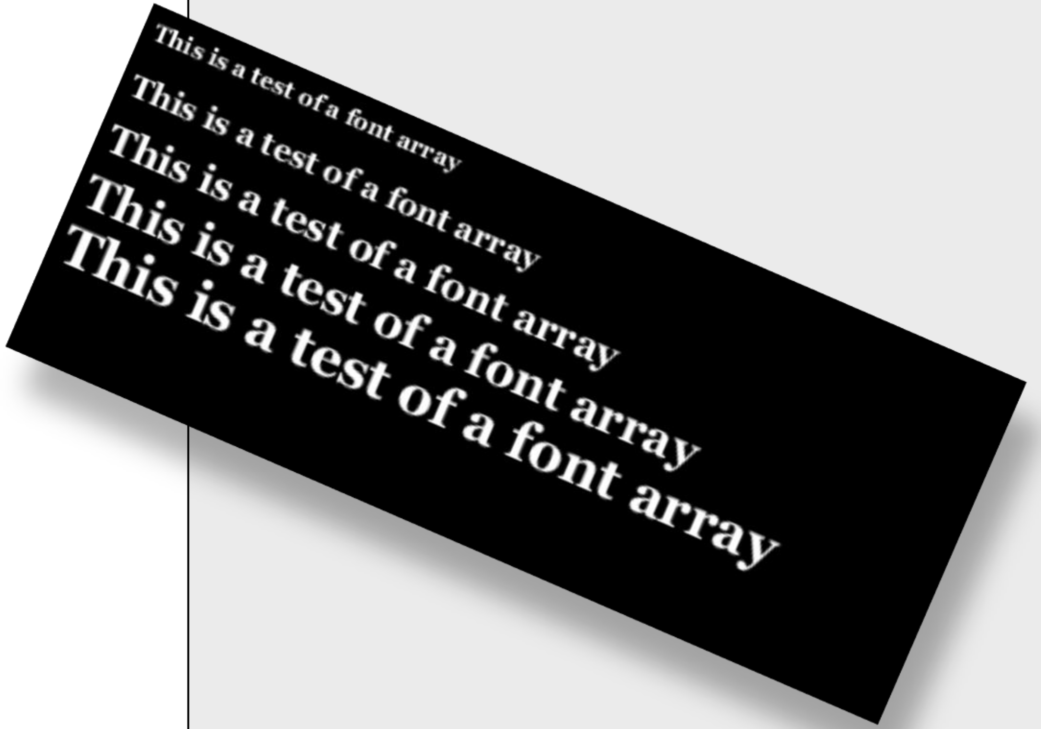
imageArr[0].width;
imageArr[1].height;

imageArr[2].get(...);
imageArr[2].set(...);
imageArr[2].filter(...);



THE PFONT OBJECT

```
int numFonts = 5;
PFont[] fontsArr = new PFont[numFonts];

void setup() {
  size(500,200);
  for (int i=0; i < fontsArr.length; i++) {
    fontsArr[i] = createFont("Georgia", 12 + i * 4);
  }
}

void draw() {
  background(0);

  int count=0;

  //With enhanced loop
  for (PFont myFont : fontsArr) {
    textFont(myFont);
    text("This is a test of a font array", 10, 20 + count * 30);
    count++;
  }
}
```



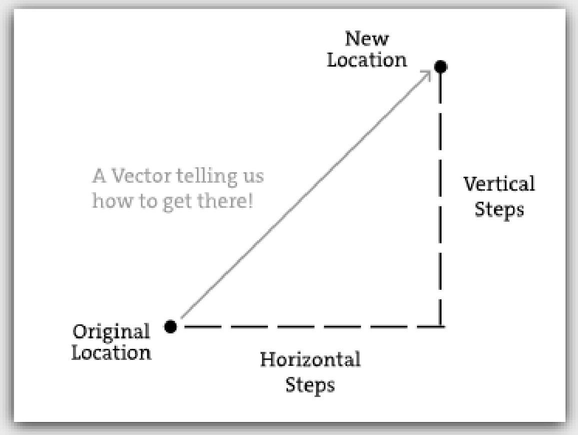
```
This is a test of a font array
This is a test of a font array
This is a test of a font array
This is a test of a font array
This is a test of a font array
```

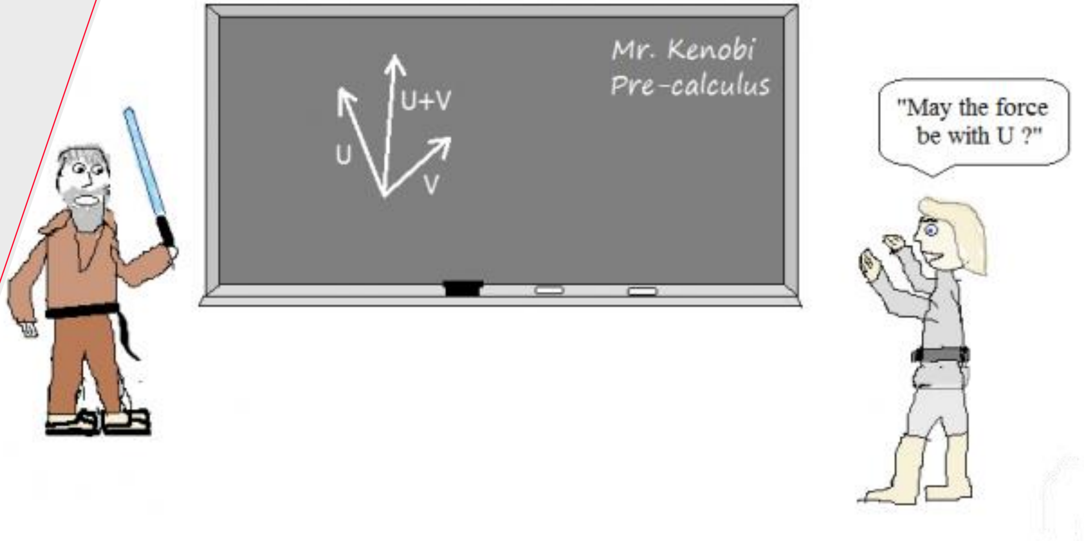
THE PVECTOR OBJECT

= describes a two or three dimensional vector
x,y for 2D
x,y,z for 3D

A vector is a collection of values that describe relative position in space. In Processing, PVector is used when we need a position, velocity or acceleration.

vector = difference between two points

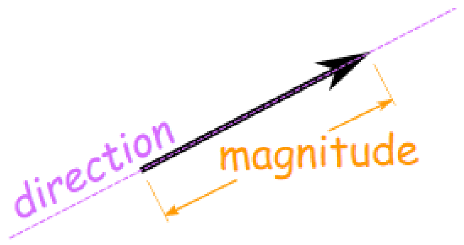




VECTOR? WHAT?

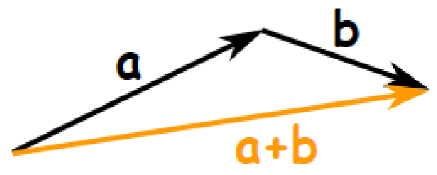
This is a vector:

A vector has **magnitude** (size) and **direction**:

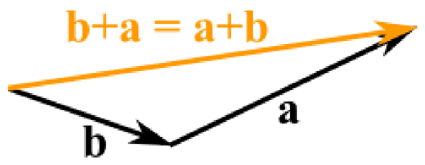


The length of the line shows its magnitude and the arrowhead points in the direction.

We can add two vectors by joining them head-to-tail:



And it doesn't matter which order we add them, we get the same result:

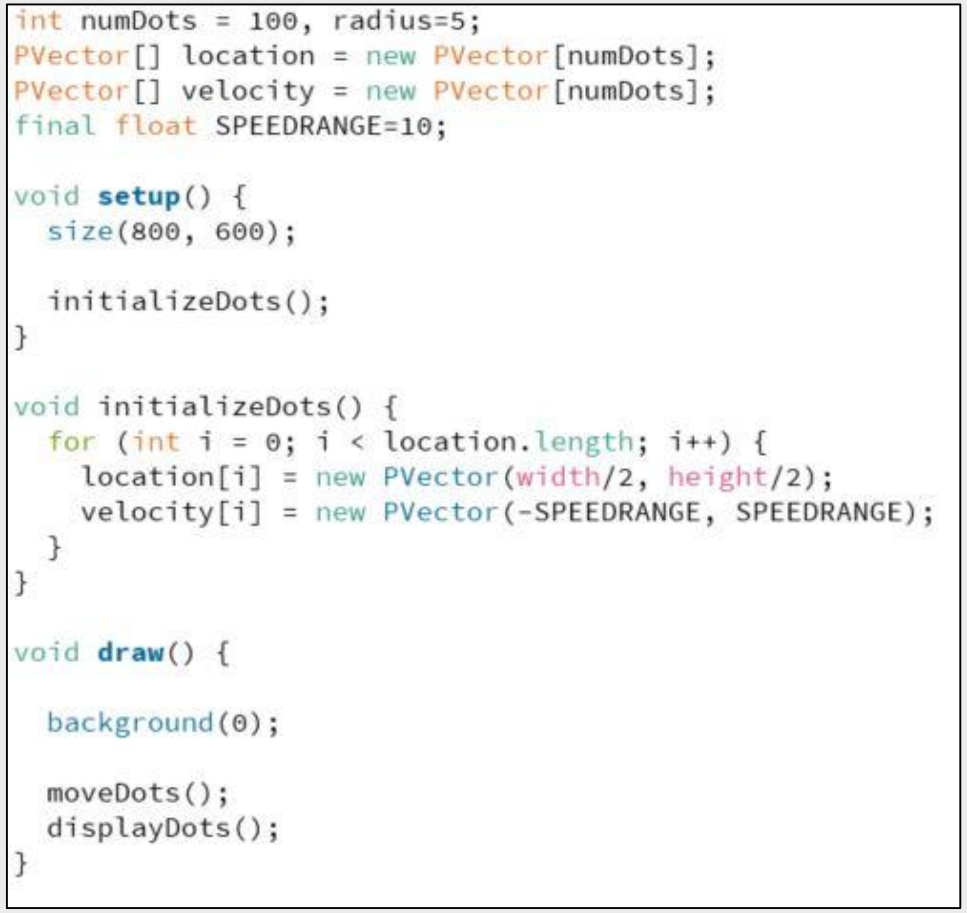


programming for artists II

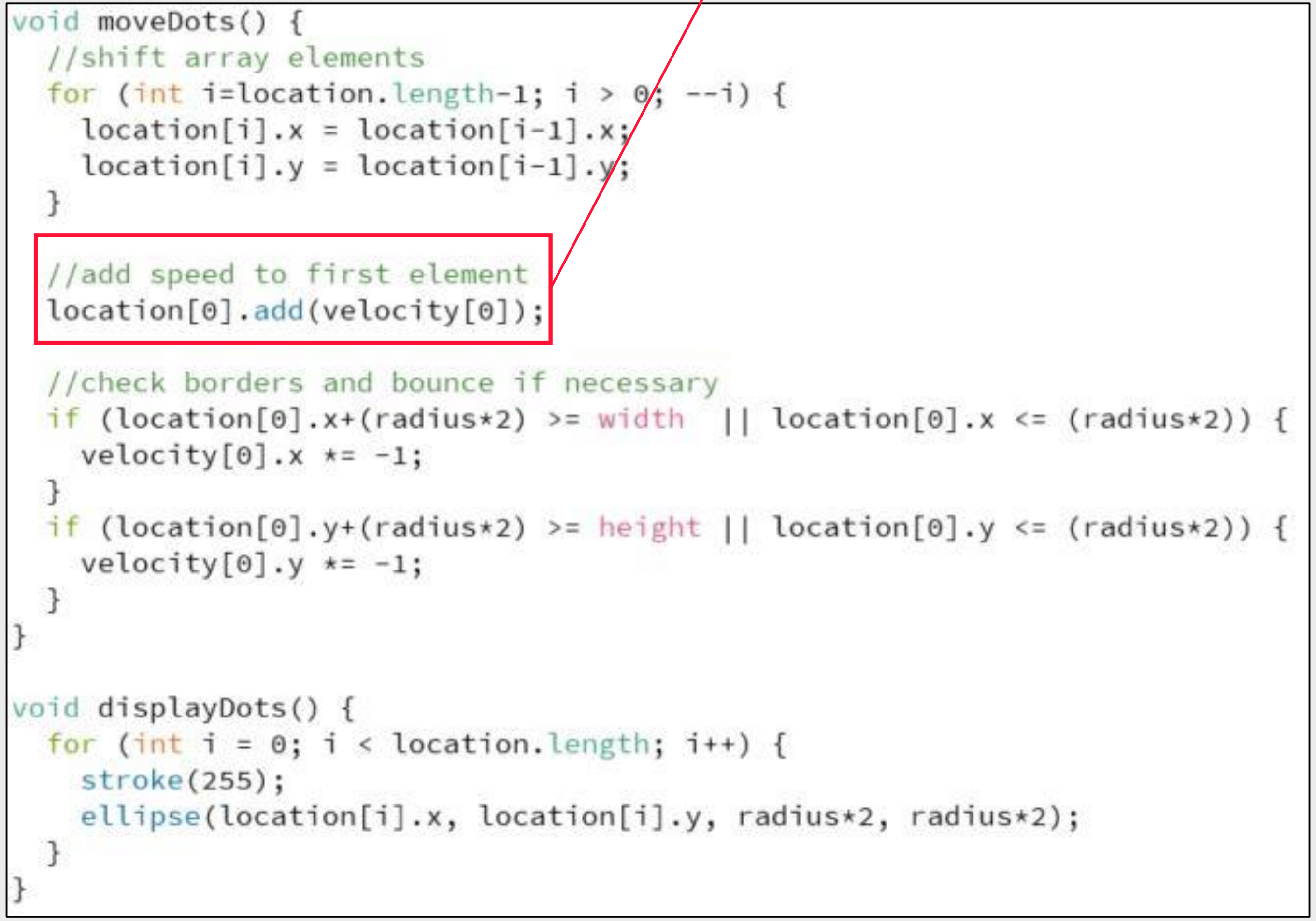
PVECTOR ARRAY

To calculate 2 PVectors, use the **add-method**!

```
//in PVector class:
void add(PVector v) {
    x = x + v.x;
    y = y + v.y;
  }
```

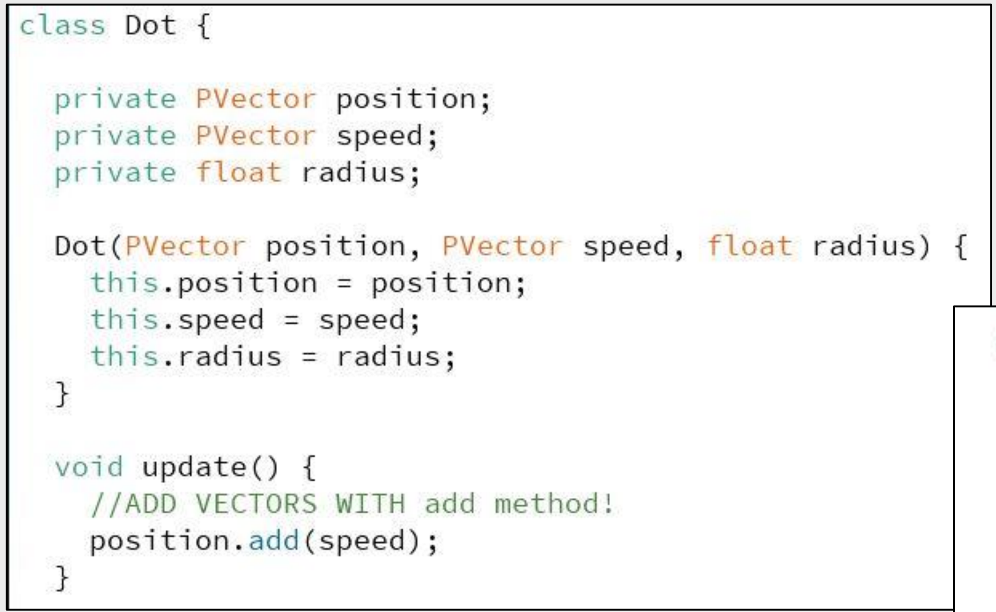


```
int numDots = 100, radius=5;
PVector[] location = new PVector[numDots];
PVector[] velocity = new PVector[numDots];
final float SPEEDRANGE=10;

void setup() {
  size(800, 600);

  initializeDots();
}

void initializeDots() {
  for (int i = 0; i < location.length; i++) {
    location[i] = new PVector(width/2, height/2);
    velocity[i] = new PVector(-SPEEDRANGE, SPEEDRANGE);
  }
}

void draw() {

  background(0);

  moveDots();
  displayDots();
}
```

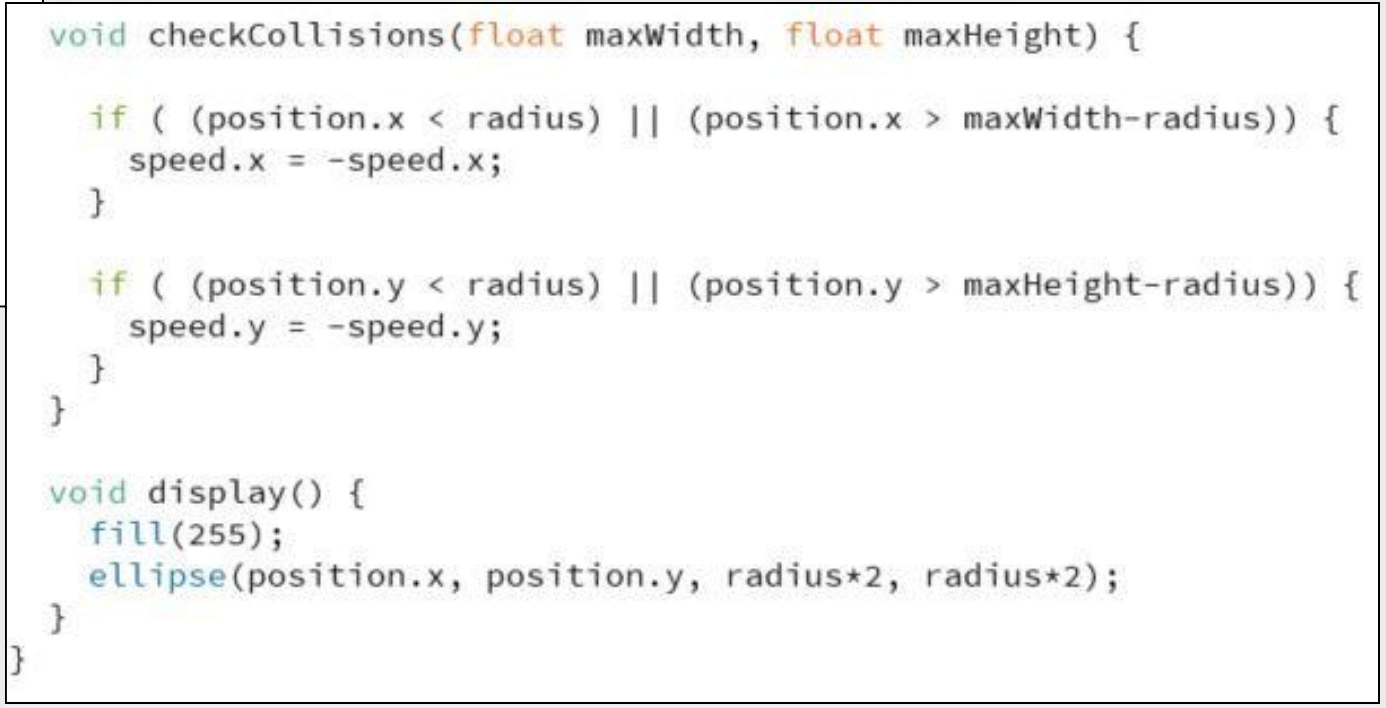


```
void moveDots() {
  //shift array elements
  for (int i=location.length-1; i > 0; --i) {
    location[i].x = location[i-1].x;
    location[i].y = location[i-1].y;
  }

  //add speed to first element
  location[0].add(velocity[0]);

  //check borders and bounce if necessary
  if (location[0].x+(radius*2) >= width  || location[0].x <= (radius*2)) {
    velocity[0].x *= -1;
  }
  if (location[0].y+(radius*2) >= height || location[0].y <= (radius*2)) {
    velocity[0].y *= -1;
  }
}

void displayDots() {
  for (int i = 0; i < location.length; i++) {
    stroke(255);
    ellipse(location[i].x, location[i].y, radius*2, radius*2);
  }
}
```

OBJECTS IN AN ARRAY: CREATING THE CLASS DOT



```
class Dot {

  private PVector position;
  private PVector speed;
  private float radius;

  Dot(PVector position, PVector speed, float radius) {
    this.position = position;
    this.speed = speed;
    this.radius = radius;
  }

  void update() {
    //ADD VECTORS WITH add method!
    position.add(speed);
  }
```



```
  void checkCollisions(float maxWidth, float maxHeight) {

    if ( (position.x < radius) || (position.x > maxWidth-radius)) {
      speed.x = -speed.x;
    }

    if ( (position.y < radius) || (position.y > maxHeight-radius)) {
      speed.y = -speed.y;
    }
  }

  void display() {
    fill(255);
    ellipse(position.x, position.y, radius*2, radius*2);
  }
}
```

OBJECTS IN AN ARRAY

```
int numDots = 100;
Dot[ ] dotsArr = new Dot[numDots];
```

the array can hold references to 100 Dot objects

it does not create the Dot objects themselves!

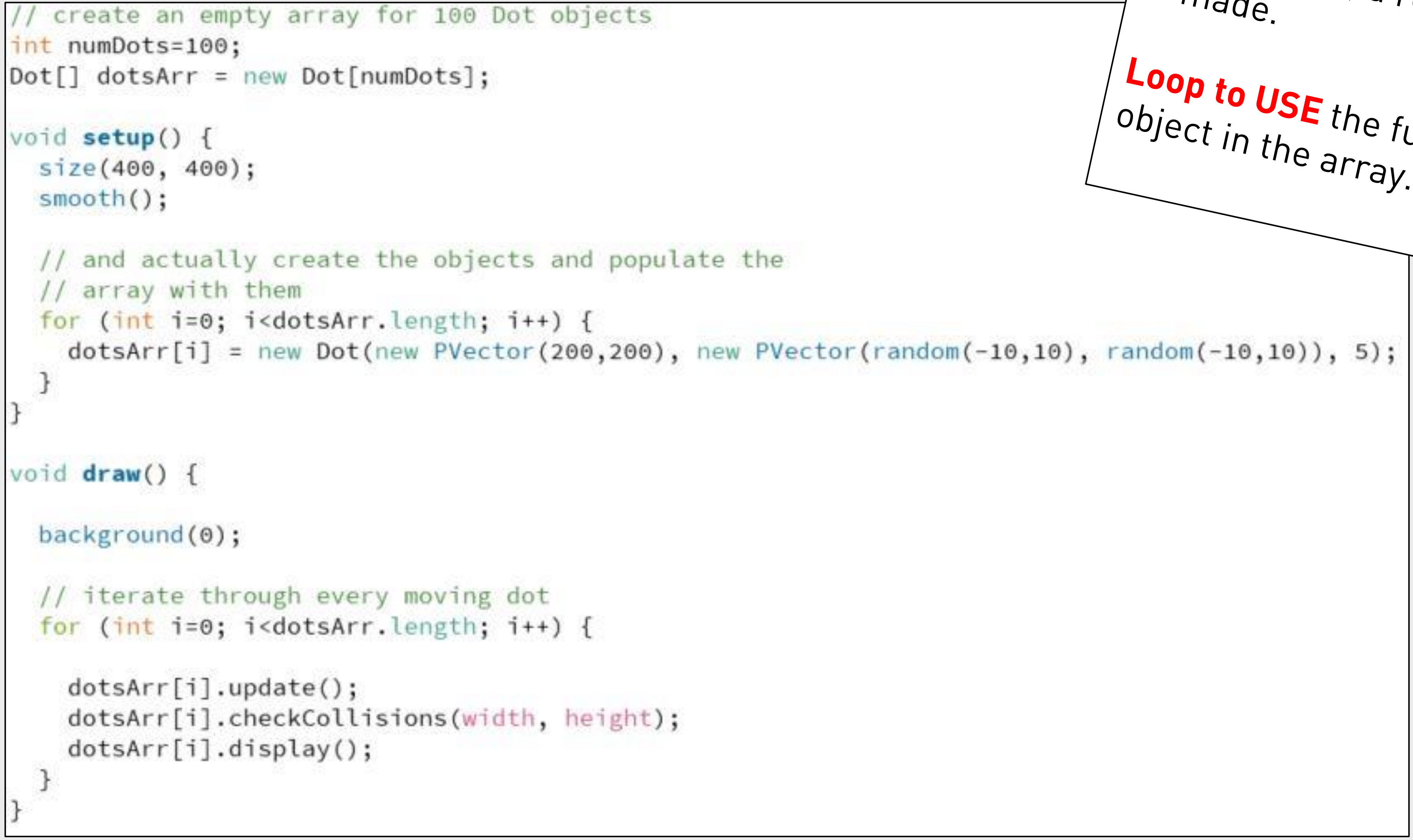
NEXT STEP → CREATE THE OBJECTS! (NullPointerException if objects are not created!)

```
//create 100 Dot objects:

for(int i = 0 ; i < dotsArr.length; i++){
   dotsArr[i] = new Dot(...);
}
```

creates a Dot object by executing the constructor

OBJECTS IN AN ARRAY

Main application



```
// create an empty array for 100 Dot objects
int numDots=100;
Dot[] dotsArr = new Dot[numDots];

void setup() {
  size(400, 400);
  smooth();

  // and actually create the objects and populate the
  // array with them
  for (int i=0; i<dotsArr.length; i++) {
    dotsArr[i] = new Dot(new PVector(200,200), new PVector(random(-10,10), random(-10,10)), 5);
  }
}

void draw() {

  background(0);

  // iterate through every moving dot
  for (int i=0; i<dotsArr.length; i++) {

    dotsArr[i].update();
    dotsArr[i].checkCollisions(width, height);
    dotsArr[i].display();
  }
}
```

Loop to CREATE the objects and store in the array.
With objects, a reference to that object is made.

Loop to USE the functions for each object in the array.

OBJECTS WITH ARRAY METHODS

Declaration of an array to store 1 object:

```
Dot[] dotsArr = new Dot[1];
```

In the setup, we create a first circle object and assign it to the element of the array:

```
dotsArr[0] = new Dot(...);
```

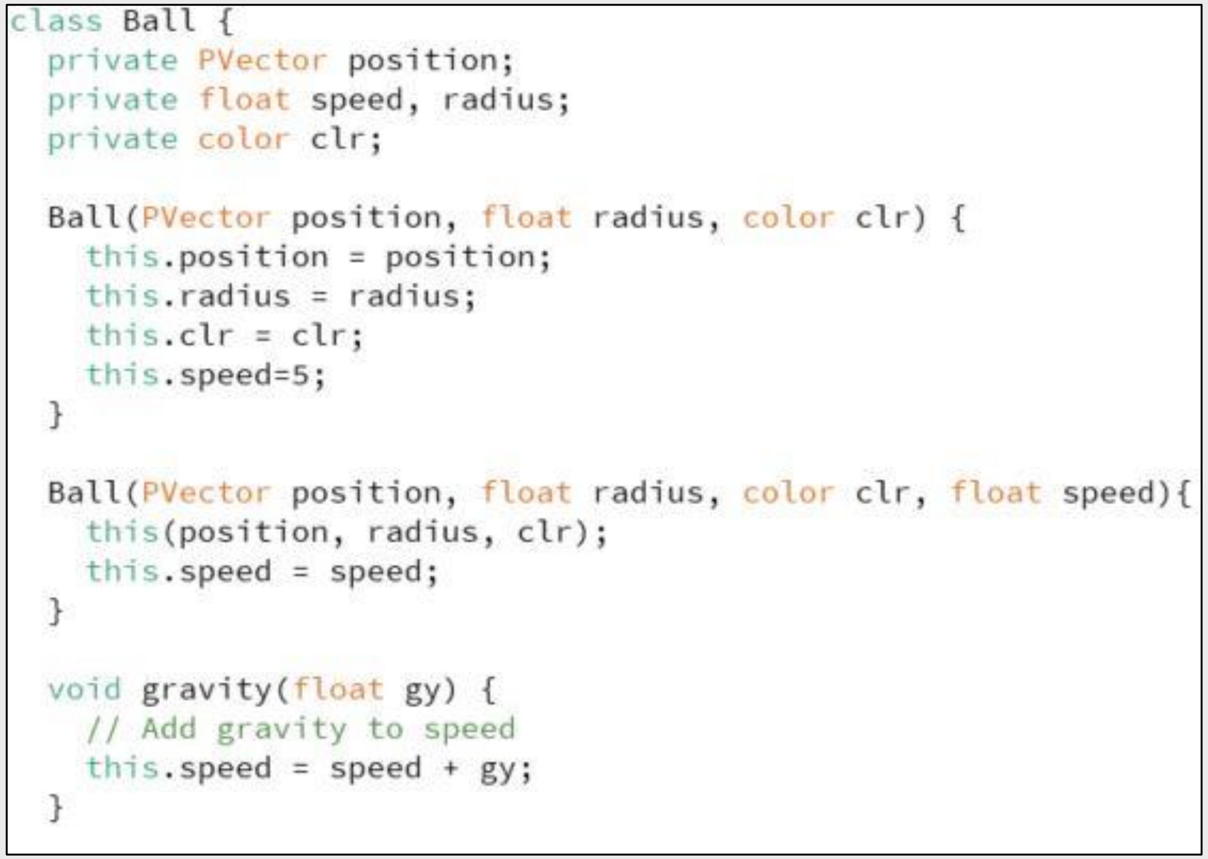
When using an array of objects, the data returned from the function must be **cast to the object array's data type**:

```
Dot d = new Dot();
dotsArr = (Dot[]) append(dotsArr, d);

//remove the last object from the array
dotsArr = (Dot[]) shorten(dotsArr);

//double the size of the array
dotsArr = (Dot[]) expand(dotsArr);
```

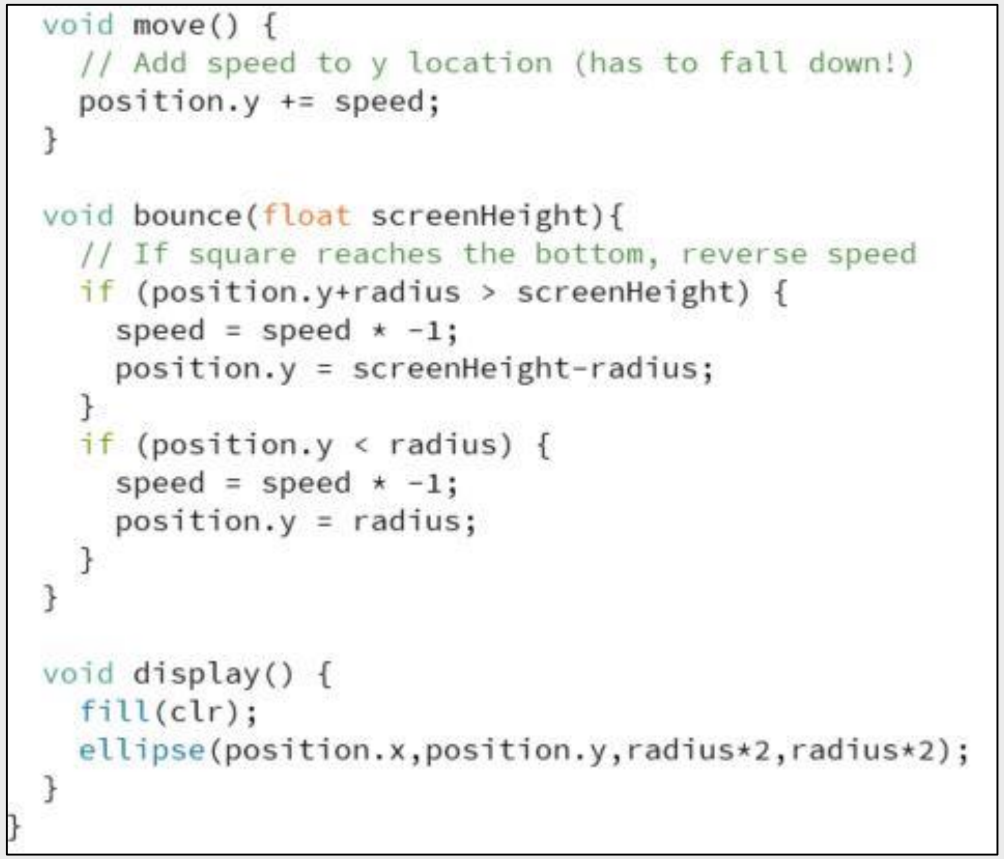
programming for artists II

DEMO BALLS

Class Ball



```
class Ball {
  private PVector position;
  private float speed, radius;
  private color clr;

  Ball(PVector position, float radius, color clr) {
    this.position = position;
    this.radius = radius;
    this.clr = clr;
    this.speed=5;
  }

  Ball(PVector position, float radius, color clr, float speed){
    this(position, radius, clr);
    this.speed = speed;
  }

  void gravity(float gy) {
    // Add gravity to speed
    this.speed = speed + gy;
  }
```

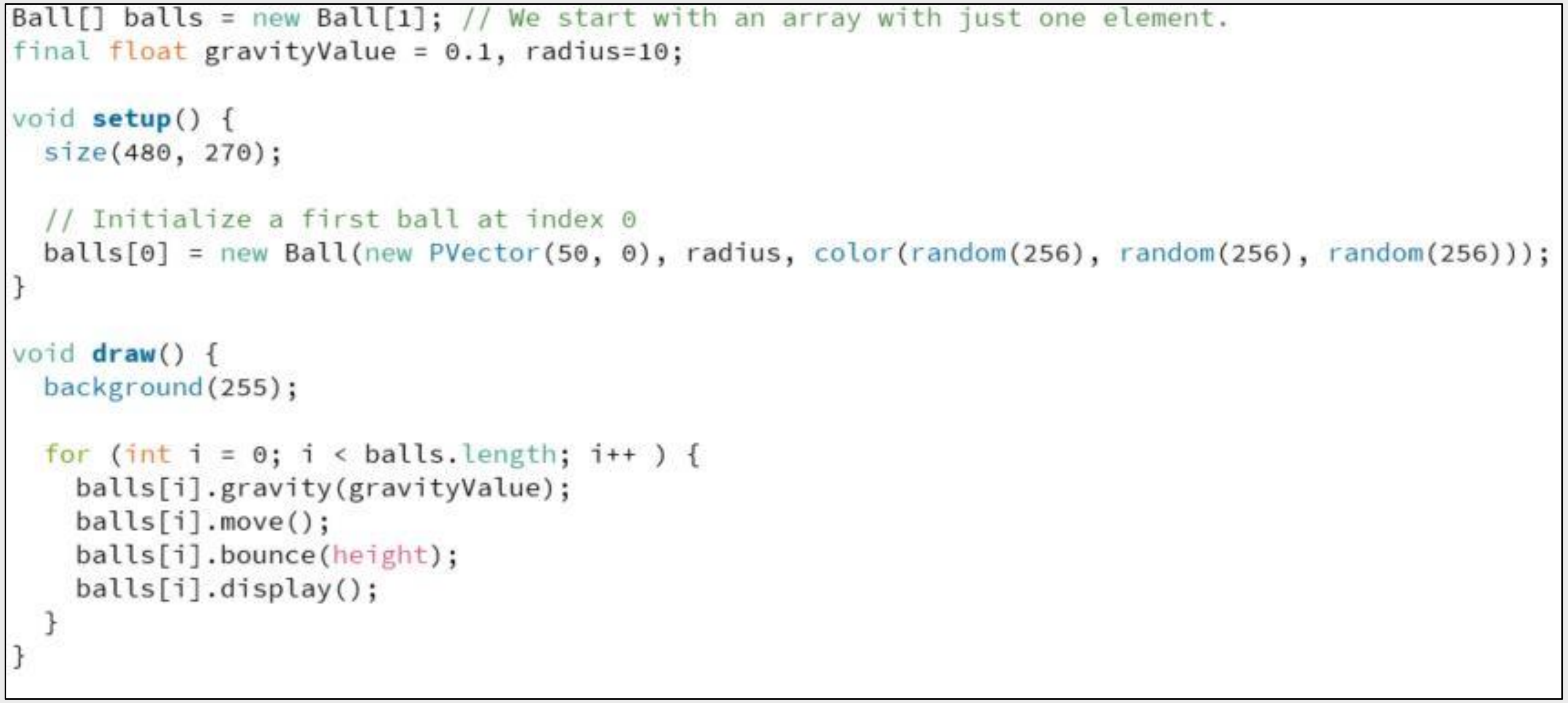


```
  void move() {
    // Add speed to y location (has to fall down!)
    position.y += speed;
  }

  void bounce(float screenHeight){
    // If square reaches the bottom, reverse speed
    if (position.y+radius > screenHeight) {
      speed = speed * -1;
      position.y = screenHeight-radius;
    }
    if (position.y < radius) {
      speed = speed * -1;
      position.y = radius;
    }
  }

  void display() {
    fill(clr);
    ellipse(position.x,position.y,radius*2,radius*2);
  }
}
```

programming for artists II

DEMO BALLS

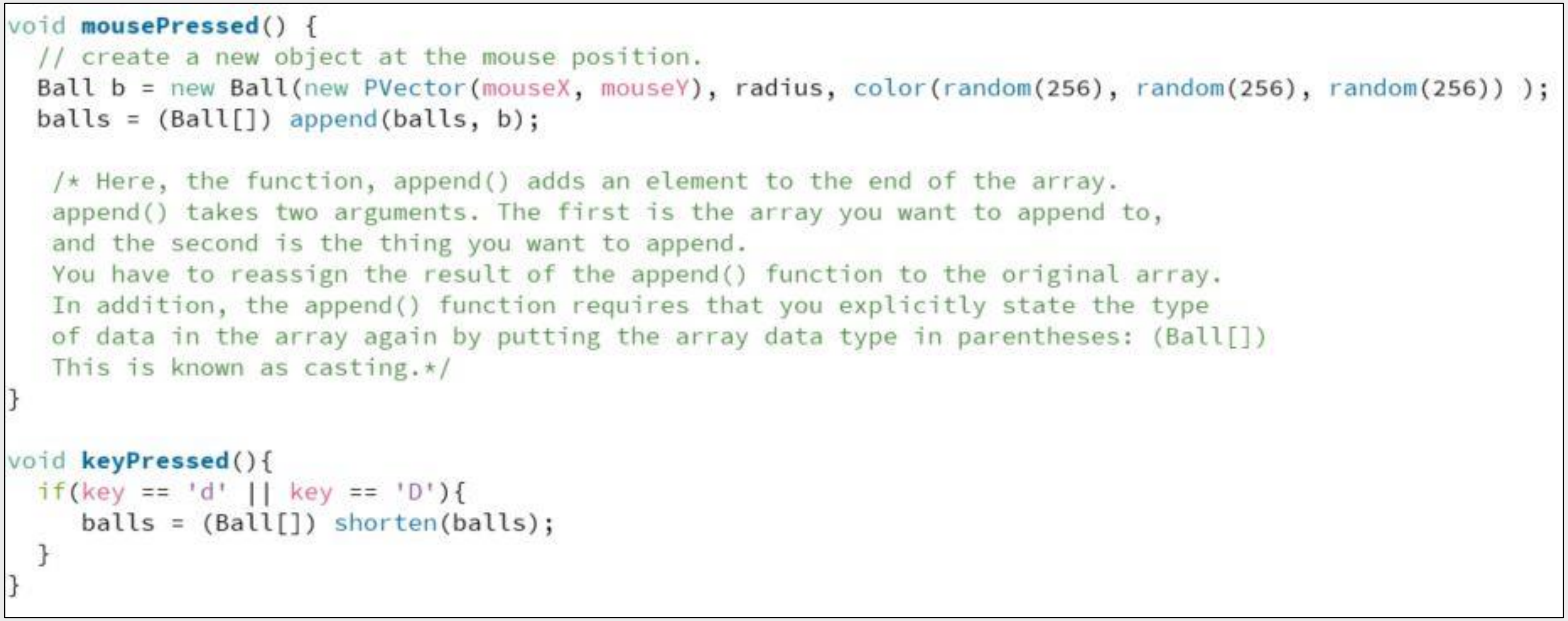
Main application



```
Ball[] balls = new Ball[1]; // We start with an array with just one element.
final float gravityValue = 0.1, radius=10;

void setup() {
  size(480, 270);

  // Initialize a first ball at index 0
  balls[0] = new Ball(new PVector(50, 0), radius, color(random(256), random(256), random(256)));
}

void draw() {
  background(255);

  for (int i = 0; i < balls.length; i++ ) {
    balls[i].gravity(gravityValue);
    balls[i].move();
    balls[i].bounce(height);
    balls[i].display();
  }
}
```

DEMO BALLS

Main application



```
void mousePressed() {
  // create a new object at the mouse position.
  Ball b = new Ball(new PVector(mouseX, mouseY), radius, color(random(256), random(256), random(256)) );
  balls = (Ball[]) append(balls, b);

   /* Here, the function, append() adds an element to the end of the array.
   append() takes two arguments. The first is the array you want to append to,
   and the second is the thing you want to append.
   You have to reassign the result of the append() function to the original array.
   In addition, the append() function requires that you explicitly state the type
   of data in the array again by putting the array data type in parentheses: (Ball[])
   This is known as casting.*/
}

void keyPressed(){
  if(key == 'd' || key == 'D'){
     balls = (Ball[]) shorten(balls);
  }
}
```