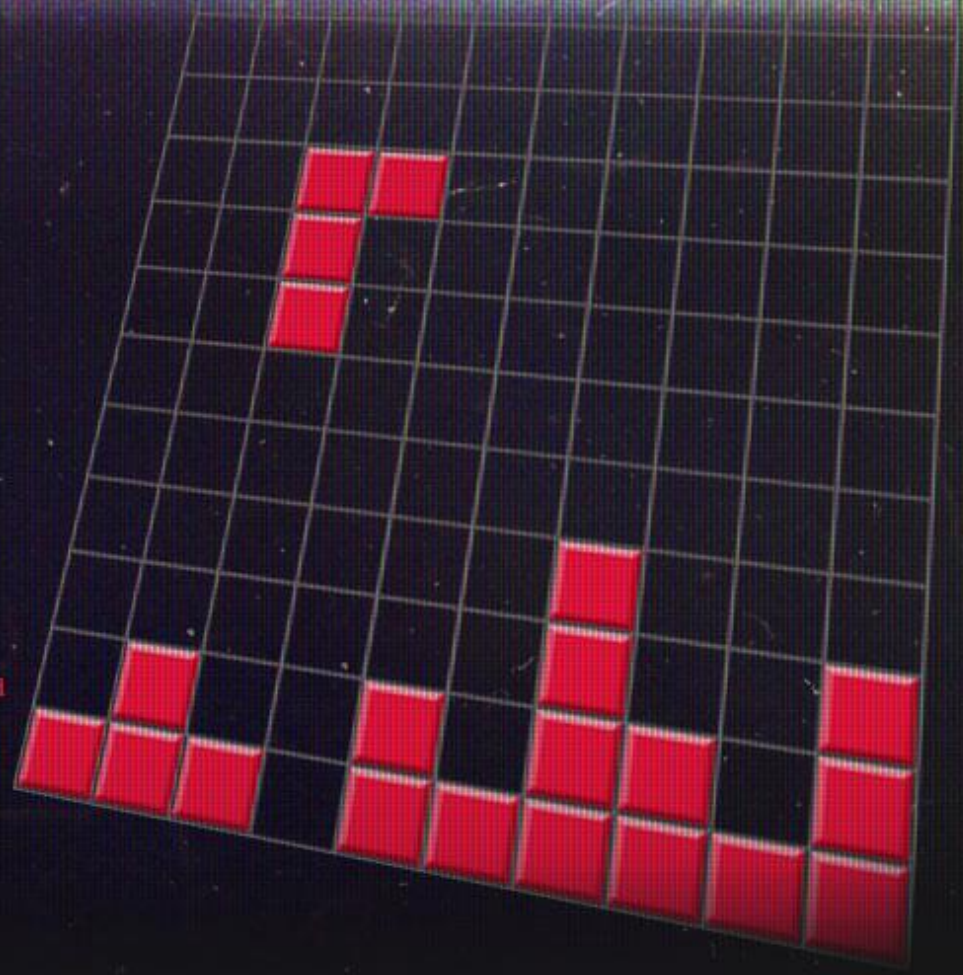
```
void newBlock(){
  isActiveBlockFree = true;
  block = new Block(int(random(8)),1);
  //op rij 0, middenste kolom
  block.setRowColumn(0, 5);
}

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

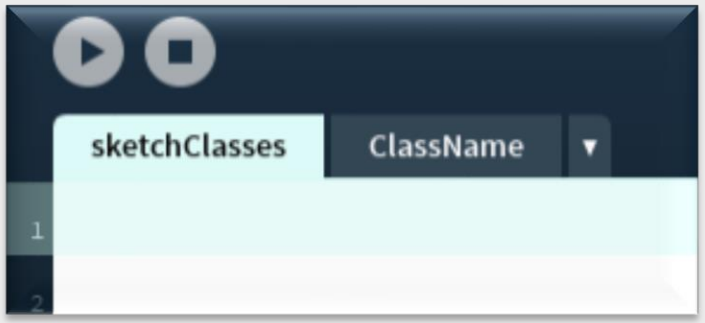
CLASSES: OBJECT ORIENTED PROGRAMMING

OOP: CLASSES & OBJECTS

CREATION **class** keyword + class name (**capitalize** the name!)

USE creating **objects** (= instances)

- a **CLASS** is an **extensible** program-code-**template** for creating objects
- an **OBJECT** is a **single instance** of a class, with **related state** and **behavior**



Include class in sketch:
→ Add code in main tab
→ Add code in separate tab

NEVER forget to create the setup() function!

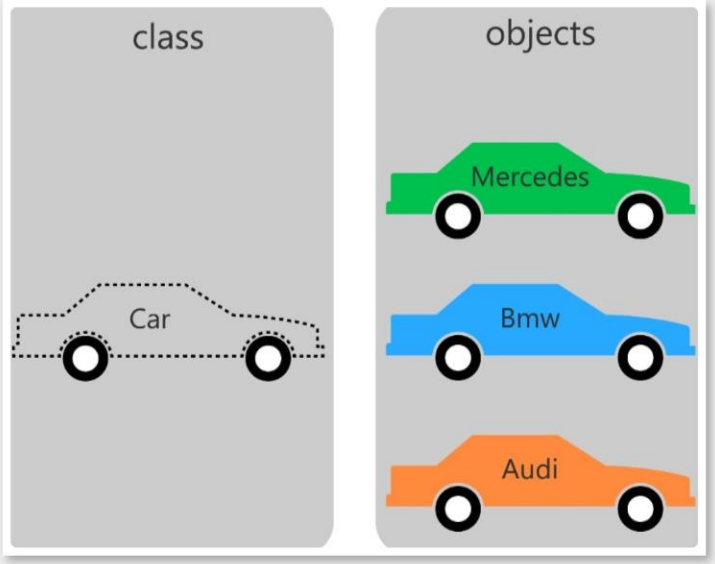
CREATE A CLASS

* A Class **contains**:

  - data members (=properties): built-in variables of the class
  - member functions/methods: actions performed with the variables

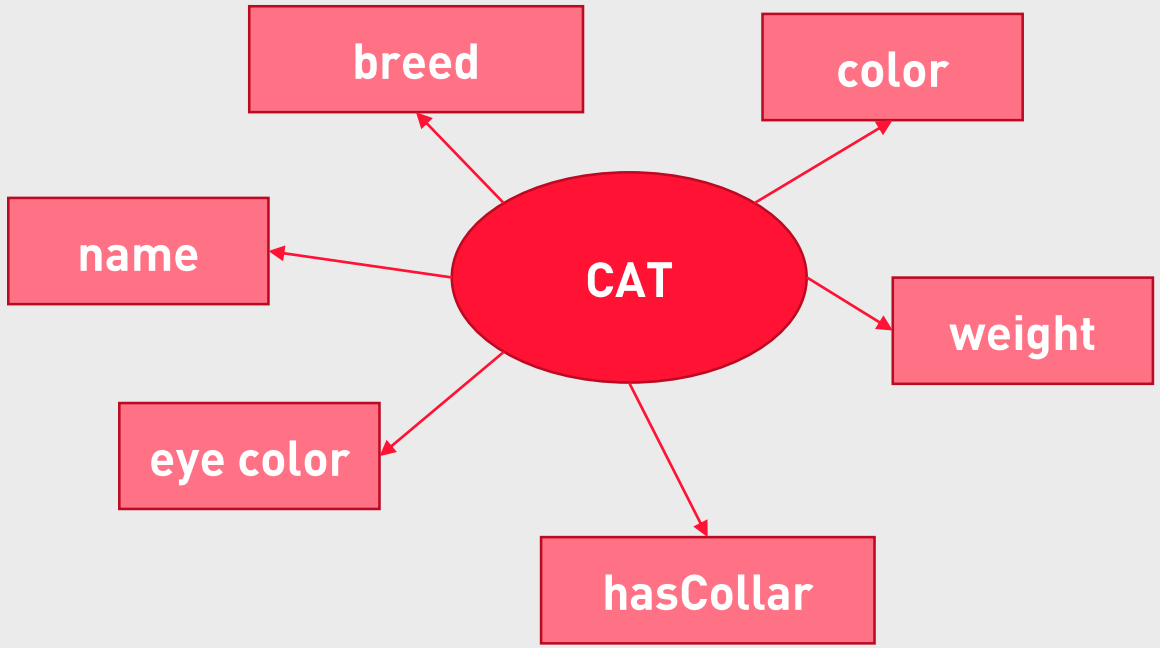
* An **object** is **assigned** to a **variable** (reference variables)

  - Primitive variables are assigned with a **single value** (int, float, char)
  - Reference variables need to refer to **multiple values**, the variable will hold a reference to the object data, not the data directly



ADVANTAGES OF OOP

- Clear modular structure
- Easy to maintain for new objects
- Provides good framework for libraries



STRUCTURE OF A CLASS

- **CLASS DEFINITION**
- **DATA MEMBERS**
- **CONSTRUCTOR(S)**
- **METHODS**

```
class ClassName {
  // DATAMEMBERS (properties of the class)
  int property1;
  float property2;

  // CONSTRUCTORS
  ClassName() {
  }

  ClassName(int prop1, float prop2) {
    this.property1 = prop1;
    this.property2 = prop2;
  }

  // METHODS
  void setProperty1(int prop1) {
    this.property1 = prop1;
  }

  int getProperty1() {
    return property1;
  }
}
```

CLASS DEFINITION

- **ALWAYS STARTS** with the keyword **class**
- **Class name ALWAYS STARTS** with a CAPITAL LETTER, then use camelCasing *(class Car, class BankAccount,...)*
- **Class body** (the area between curly braces)

```
class Button{
  private String txtBtn;
  private int xPos, yPos, widthBtn, heightBtn;
  private boolean  isArmed, isTriggered;

  private color backColor = color(227, 227, 227);
  private color backColorArmed = color(0, 255, 0);
  private color foreColor = color(0, 0, 0);

  Button(String txtBtn, int xPos, int yPos, int widthBtn, int heightBtn){
    this.txtBtn = txtBtn;
    this.xPos = xPos;
    this.yPos = yPos;
    this.widthBtn = widthBtn;
    this.heightBtn = heightBtn;
  }
```

DATA MEMBERS

```
class Button{
  private String txtBtn;
  private int xPos, yPos, widthBtn, heightBtn;
  private boolean  isArmed, isTriggered;

  private color backColor = color(227, 227, 227);
  private color backColorArmed = color(0, 255, 0);
  private color foreColor = color(0, 0, 0);

  Button(String txtBtn, int xPos, int yPos, int widthBtn, in
    this.txtBtn = txtBtn;
    this.xPos = xPos;
    this.yPos = yPos;
    this.widthBtn = widthBtn;
    this.heightBtn = heightBtn;
  }
```

- Data members stores information of every object
- Any data type
- **Automatically destroyed** (removed from memory) when the object is deleted
- **Global** scope within the class
- **Final** = constant. You can have final variables inside the method as well as at class level.

```
final float constant = 12.84753;
println(constant);  // Prints "12.84753" to the console
constant += 12.84;  // ERROR! It's not possible to change a
                                              "final" value
```

CONSTRUCTOR

```
Button(String txtBtn, int xPos, int yPos, int widthBtn, int heightBtn){
  this.txtBtn = txtBtn;
  this.xPos = xPos;
  this.yPos = yPos;
  this.widthBtn = widthBtn;
  this.heightBtn = heightBtn;
}
```

- Begins with the **same name** as the class, **no return type!**
- Utilizes **parentheses** for an optional **parameter list**
- **Data members are assigned** in the constructor (*use keyword **this** when using the same name for the data member and local variable in constructor)*
- When an **object** is **created**, the **constructor** will be **called**
  If you don't explicitly include a constructor, a **default** (internal) class **constructor** is called that has no parameters: **Button(){}**

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 {
   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**.

METHODS OR MEMBER FUNCTIONS

```
boolean isPressed(){
  return isTriggered;
}

void reset(){
  isTriggered = false;
}

void display(){
  int padding=2;
  noStroke();

  fill(backColor);
  textSize(20);
  textAlign(LEFT, TOP);

  if (!isArmed){
    rect(xPos, yPos, widthBtn, heightBtn);
    fill(foreColor);
    text(txtBtn, xPos+padding, yPos);
  }
  else{
    fill(backColorArmed);
    rect(xPos+padding, yPos+padding, widthBtn-padding, heightBtn-padding);
    fill(foreColor);
    text(txtBtn, xPos+padding, yPos+padding);
  }
}
```

- **Members** of a class
- **Interact** with data
- **Determines** what **object** can do
- **Access** to local **variables** and **data members**

CREATING OBJECTS

3 STEPS:

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

   ```
   Button myButton;
   ```

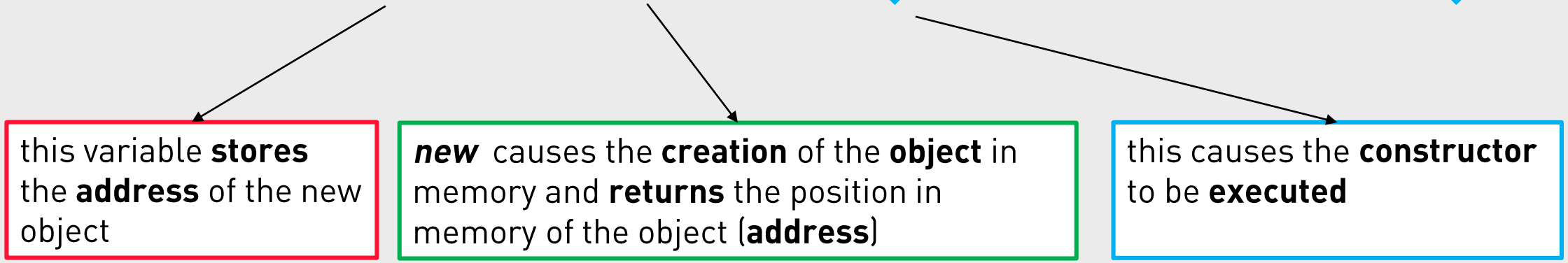
2. **INITIALIZE**
   The **constructor** will be called automatically when a new object is created with ***new***

   ```
   myButton = new Button( list of arguments if required );
   ```

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

CREATING OBJECTS

```
Button myButton = new Button( list of arguments if required );
```



Properties and methods of an object are called, using the **dot syntax:**

```
myButton.display();
```

DATA MEMBERS: ENCAPSULATION

- Refers to the **encasing** and grouping of an object's **data members** (properties) with its methods
- **Direct access** to data members and methods of a class is kept **as limited as possible**
- If other programmers use your class, you want to ensure that errors from **misuse cannot happen**
- **Avoid public fields** except for constants
- The **accessibility** of elements of a class is **set** through the use of **access specifiers** (**3 levels**):
    - **Private** (least accessible)
    - **Protected**
    - **Public** (most accessible)

ENCAPSULATION: ACCESS SPECIFIERS

PRIVATE

Only methods of the **class itself can access** the private elements of the class!

This level should be used for:

- all data members
- methods that only need to be called from **inside** the class

Private is the **default level** inside a class.

PROTECTED

Accessible to **all classes** within the **package**, also **within** the body of **any subclass** of the class, inherited classes and all classes of the current package.

This is more restrictive than public access, but less restrictive than package access.

ENCAPSULATION: ACCESS SPECIFIERS

PUBLIC

Public elements can be **accessed from anywhere** and are **visible** for **all classes**.

This level is meant **for constructors and ordinary methods**. These are meant to be called from other classes (or more accurately: called by methods of other classes)

The set of public elements of a class is called the **public interface** of that class. It is the collection of things that other classes can access.

- To be able to change/read the values of the data members, we will write **get- and set methods**.

** inside ".pde" tab files, all classes & interfaces are nested, so everything is visible for the enclosing classes.*

MEMBER METHODS: GETTERS & SETTERS

- Structure methods with the **get/set prefix**
- Class properties must not be directly accessible (properties should be **private → encapsulation**)

```
Class Student {
    private String stuName;
    private int age;

     /* a getter function gets information,
        so it returns a value */
    String  getStudentName(){
        return stuName;
    }

    /* a setter function sets a value to a data member,
    so you need to pass the new value as a parameter */
    void setStudentAge(int age){
        stuAge = age;
    }
}
```

Main application:

```
void draw(){

    println(myObject.getStudentName());

    println(myObject.stuName);

    myObject.setStudentAge(20);

}
```

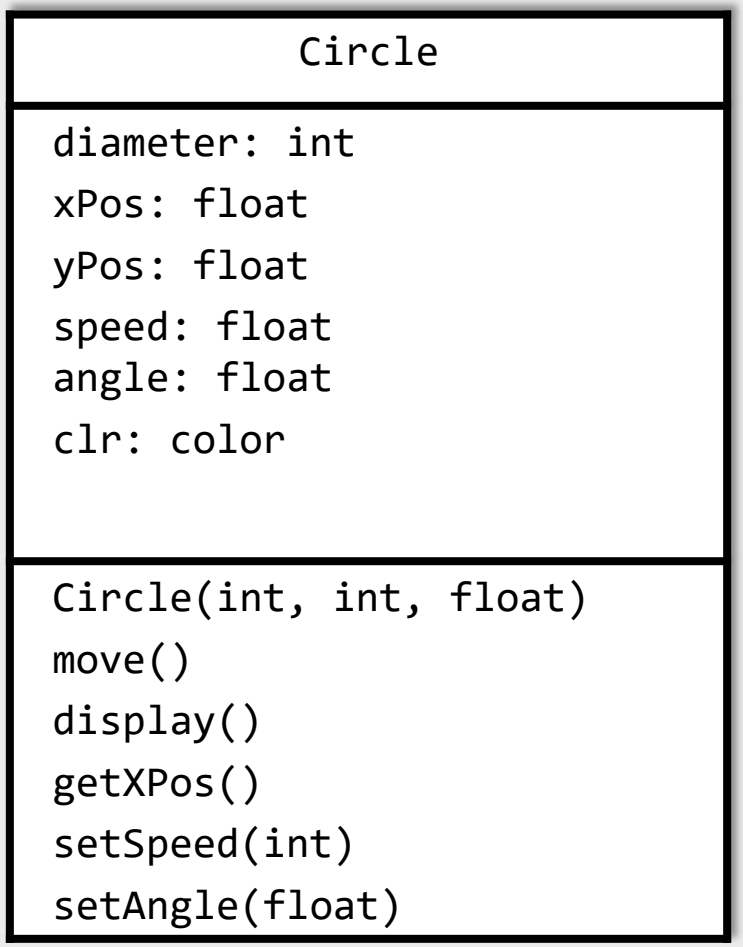
CLASS DIAGRAMS (UML)

You'll typically follow these steps when making a new class:

- Decide on the data members
- Write the constructor
- Write the member functions or methods

To visualize this, we can use a UML diagram:
Unified Modeling Language

An UML diagram is a **clear** and **concise way** of specifying the **contents** of a **class** and very useful for a programmer.



DEMO BOUNCING CIRCLES

Class Circle

```
class Circle{
  private float xPos, yPos, angle;
  private int speed, diameter;
  private color clr;

  //constructor
  Circle(float xPos, float yPos, int diameter, color clr){
    this.xPos = xPos;
    this.yPos = yPos;
    this.clr=clr;
    this.diameter = diameter;
  }

  void display(){
    noStroke();
    fill(clr);
    ellipse(xPos, yPos, diameter, diameter);
  }

  void setAngle(float angle){
    this.angle = angle;
  }

  void setSpeed(int speed){
    this.speed = speed;
  }

  void move(float screenWidth, float screenHeight){
    xPos += speed * cos(angle);
    if(xPos < diameter/2 || xPos > screenWidth - diameter/2){  speed *= -1; }

    yPos += speed * sin(angle);
    if(yPos < diameter/2 || yPos > screenHeight - diameter/2){ speed *= -1; }
  }

  int getXPos(){
    return (int)xPos;
  }
}
```

DEMO BOUNCING CIRCLES

Main application

```
//DECLARATION
Circle randomCircle, centerCircle, fixedCircle;

void setup(){
  size(842,480);

  initializeCircles();
}

void initializeCircles(){
  //random red circle
  randomCircle = initNewCircle(random(width), random(height), 100, #F51B1B, 6.28, 8);

  //centered green circle
  centerCircle = initNewCircle(width/2, height/2, 80, #1BF527, 2.75, 15);

  //fixed blue circle
  fixedCircle = initNewCircle(random(width), random(height), 50, #1B98F5, 0, 0);
}

Circle initNewCircle(float x, float y, int diam, color col, float angle, int speed){

  Circle tempCircle = new Circle(x, y, diam, col);
  tempCircle.setAngle(angle);
  tempCircle.setSpeed(speed);

  return tempCircle;
}
```

DEMO BOUNCING CIRCLES

Main application

```
void draw(){
  background(30);

  //USE THE OBJECTS METHODS
  randomCircle.move(width, height);
  randomCircle.display();

  centerCircle.move(width, height);
  centerCircle.display();

  fixedCircle.display();
  fill(255);
  text("The x position of the fixed circle is: " + fixedCircle.getXPos(), 20, 20);
}
```