EECS 1710
Programming for Digital Media

Lecture 6 :: Methods & Arguments

YORK UNIVERSITÉ UNIVERSITY

This Week

Lecture 5 (expressions/operators):

- *boolean & char types (from last lecture)*
- *numeric operators (revisited)*
- *numeric expressions*
- *mixing data types*
- *promotion & demotion of data types*
- *constants*
- *style*

Lecture 6 (methods & arguments)

- methods: general structure (arguments and return types)
- more drawing methods
- math-based methods
- Strings & string methods

Useful programs are MODULAR

As programs grow in complexity, it becomes necessary to:

- organize code layout/style (make it more readable)
- begin **delegating** functionality to other components (modularizing the code)

This facilitates RE-USE

- We have already seen this to some extent…
- In our dynamic sketches, we have two code blocks:
  - setup() { } and
  - draw() { }
- draw() was essentially reused (as it was repeatedly run)

Area.pde & AreaToOrigin.pde [reusable?]

```
// Area.pde

int rectWidth = 8;
int rectHeight = 3;
int area = rectWidth *
           rectHeight;

println(area);
```

```
// AreaToOrigin.pde

int rectWidth = 0;
int rectHeight = 0;
int area;

void setup() {
  size(640, 480);
}

void draw() {

  background(255, 255, 255);
  fill(0, 0, 0);
  rect(0, 0, rectWidth, rectHeight);

  rectWidth = mouseX;
  rectHeight = mouseY;

  area = rectWidth * rectHeight;
  print("Area = ");
  println(area);
}
```

we have seen these..

- setup() {…}
- draw() {…}

Statements are re-run

(code is re-used)

```
// AreaToOrigin.pde

int rectWidth = 0;
int rectHeight = 0;
int area;

void setup() {
  size(640, 480);
}

void draw() {

  background(255, 255, 255);
  fill(0, 0, 0);
  rect(0, 0, rectWidth, rectHeight);

  rectWidth = mouseX;
  rectHeight = mouseY;

  area = rectWidth * rectHeight;
  print("Area = ");
  println(area);
}
```

What if we want to compute areas for many different rectangles (arbitrarily in our code)??

```
int rectWidth = 8;
int rectHeight = 3;
int area = rectWidth * rectHeight;
println(area);


rectWidth = 11;
rectHeight = 5;
area = rectWidth * rectHeight;
println(area);
```

Copy + Paste?

NOT EFFICIENT, INFLEXIBLE

A block of statements (alone)

- Not really reusable
- Needs a way to be invoked/re-run
- Needs an identifier
- Needs a way to compute for different variable values

```
{
    int rectWidth = 8;
    int rectHeight = 3;
    int area = rectWidth *
           rectHeight;

    println(area);
}
```

What we really need is a (reusable) template we can delegate the task of computing an area to

“arguments/ parameters” (optional)

“method”

“return values” (optional)

Methods (re-usable + adaptive)

- A **method** in java (sometimes called a “function”):
  - A sequence of instructions (statements) that we package in a separate block of code {…}, along with an identifier + some other features
  - The sequence of statements can then be re-used as often as we need to

- We call a method (“invoke the method”) each time we want to re-use that code.
  - Features?
    - We may pass data to the method (optional)
    - We may get back data from the method (also optional)

What we really need is a (reusable) template we can delegate the task of computing an area to

“arguments/ parameters” (optional)

“method”

“return values” (optional)

Lets delegate area calculation to its own method

- void?
  - returns nothing
  - i.e. no output
- ( )
  - no arguments
  - i.e. no input

Here, the variables used have global scope

variables are set within draw(), then method invoked from inside draw()

```
// AreaToOrigin.pde

int rectWidth = 0;
int rectHeight = 0;
int area;

void setup() {
  size(640, 480);
}

void rectArea() {
  area = rectWidth * rectHeight;
  print("Area = ");
  println(area);
}

void draw() {
  background(255, 255, 255);
  fill(0, 0, 0);
  rect(0, 0, rectWidth, rectHeight);
  rectWidth = mouseX;
  rectHeight = mouseY;
  rectArea();
}
```

How about a version with inputs and an output?!

“arguments/ parameters” (optional)

“method”

“return values” (optional)

YORK UNIVERSITÉ UNIVERSITY

Lets delegate area calculation to its own method

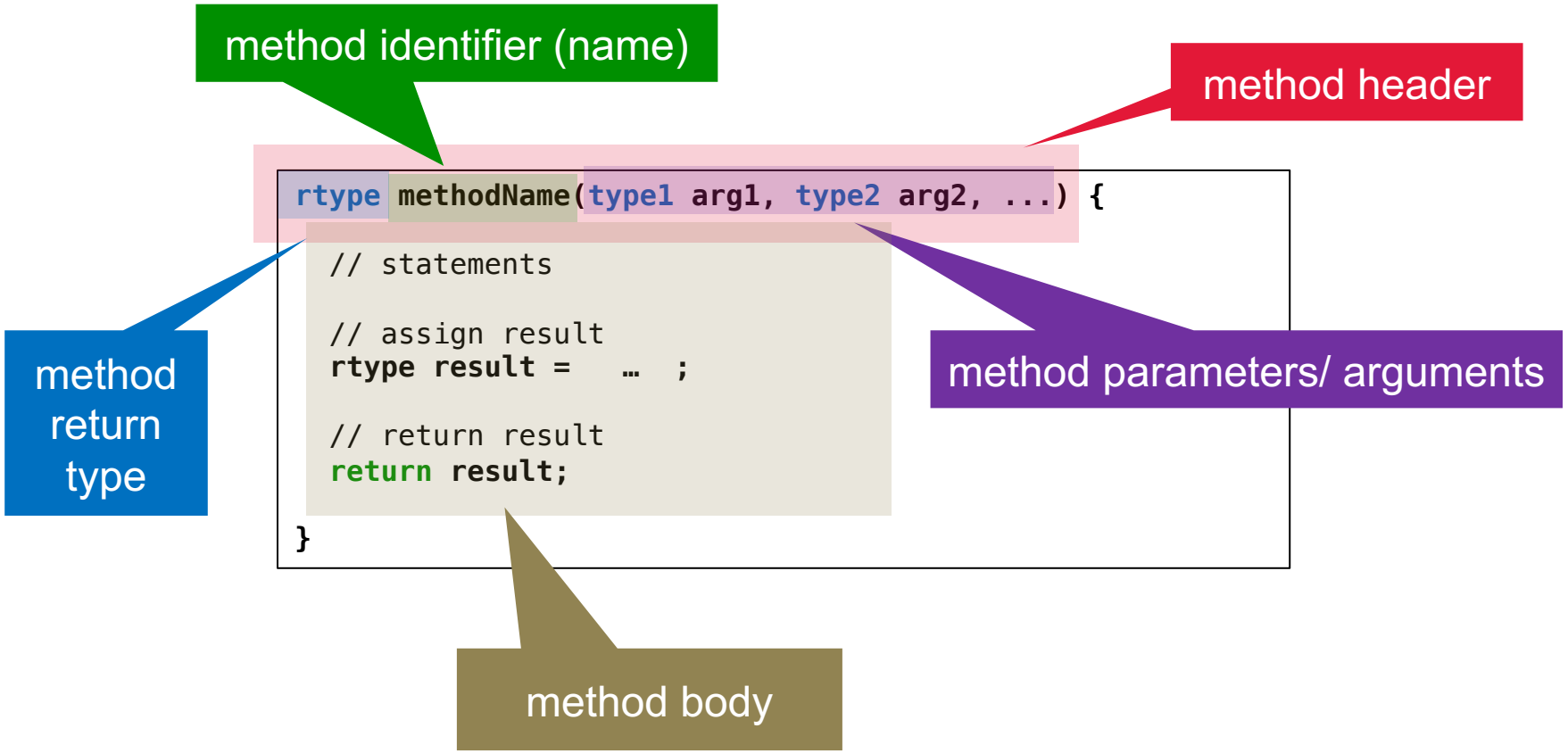
```
type rectArea(type1 arg1,
              type2 arg2, ...) {

  // statements
  // assign result
  // return result

}
```

Here, the variables used have local scope

This version has a return type (int), and two arguments (also int)

```
// AreaToOrigin.pde

int rectWidth = 0;
int rectHeight = 0;
int area;

void setup() {
  size(640, 480);
}

int rectArea(int rectW, int rectH) {
  int area = rectW * rectH;
  return area;
}

void draw() {
  background(255, 255, 255);
  fill(0, 0, 0);
  rect(0, 0, rectWidth, rectHeight);
  rectWidth = mouseX;
  rectHeight = mouseY;
  print("Area = ");
  println(rectArea(rectWidth, rectHeight));
}
```

General form of a method



YORK UNIVERSITÉ UNIVERSITY

General form of a method

```
rtype methodName(type1 arg1, type2 arg2, ...) {

  // statements

  // assign result
  rtype result =   …  ;

  // return result
  return result;

}
```

```
void methodName(type1 arg1, type2 arg2, ...) {

  // statements

  // assign result
  type0 result =   …  ;


}
```

No return type

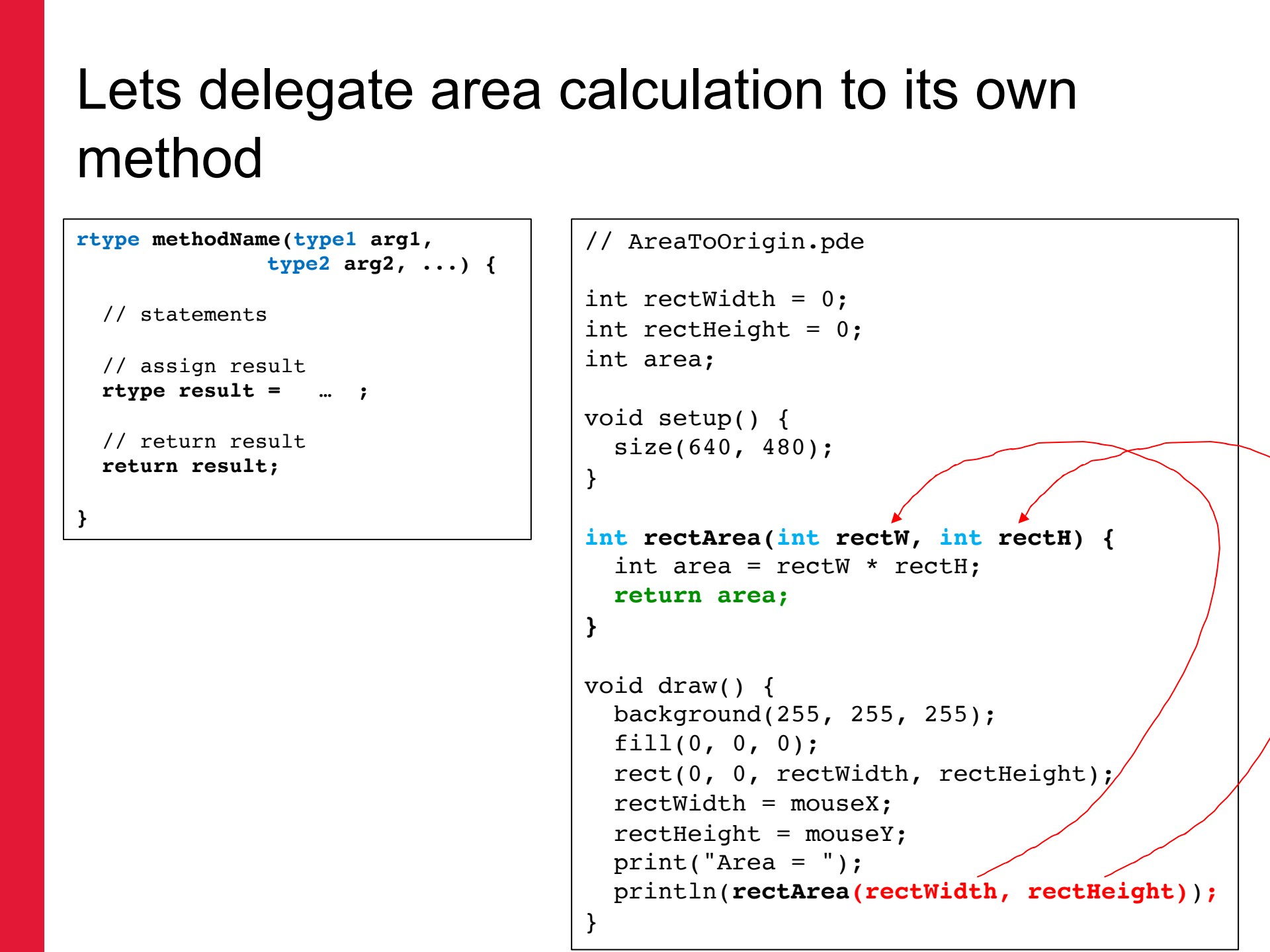
```
rtype methodName() {

  // statements

  // assign result
  rtype result =   …  ;

  // return result
  return result;
}
```

No input arguments

Lets delegate area calculation to its own method

```
rtype methodName(type1 arg1,
                 type2 arg2, ...) {

  // statements

  // assign result
  rtype result =   …  ;

  // return result
  return result;

}
```

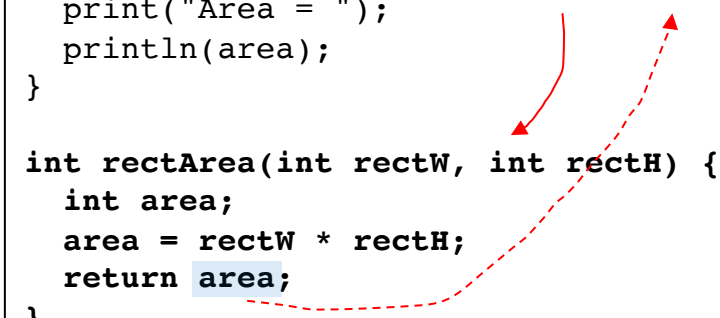
```
// AreaToOrigin.pde

int rectWidth = 0;
int rectHeight = 0;
int area;

void setup() {
  size(640, 480);
}

int rectArea(int rectW, int rectH) {
  int area = rectW * rectH;
  return area;
}

void draw() {
  background(255, 255, 255);
  fill(0, 0, 0);
  rect(0, 0, rectWidth, rectHeight);
  rectWidth = mouseX;
  rectHeight = mouseY;
  print("Area = ");
  println(rectArea(rectWidth, rectHeight));
}
```



Example

```
// AreaToOrigin.pde

int rectWidth = 0;
int rectHeight = 0;
int area;

void setup() {
  size(640, 480);
}

void draw() {

  background(255, 255, 255);
  fill(0, 0, 0);
  rect(0, 0, rectWidth, rectHeight);

  rectWidth = mouseX;
  rectHeight = mouseY;

  area = rectWidth * rectHeight;
  print("Area = ");
  println(area);
}
```

```
// AreaToOriginWithMethod.pde

int rectWidth = 0;
int rectHeight = 0;

void setup() {
  size(640, 480);
}
void draw() {

  background(255, 255, 255);
  fill(0, 0, 0);
  rect(0, 0, rectWidth, rectHeight);
  rectWidth = mouseX;
  rectHeight = mouseY;
}

void mousePressed() {
  int area = rectArea(rectWidth, rectHeight);
  print("Area = ");
  println(area);
}

int rectArea(int rectW, int rectH) {
  int area;
  area = rectW * rectH;
  return area;
}
```



So… ”commands” are actually “methods”

```
// AreaToOriginWithMethod.pde

int rectWidth = 0;
int rectHeight = 0;

void setup() {
  size(640, 480);
}
void draw() {

  background(255, 255, 255);
  fill(0, 0, 0);
  rect(0, 0, rectWidth, rectHeight);
  rectWidth = mouseX;
  rectHeight = mouseY;
}

void mousePressed() {
  int area = areaRect(rectWidth, rectHeight);
  print("Area = ");
  println(area);
}

int areaRect(int rectW, int rectH) {
  int area;
  area = rectW * rectH;
  return area;
}
```

Syntax

```
rect(a, b, c, d)
rect(a, b, c, d, r)
rect(a, b, c, d, tl, tr, br, bl)
```

Parameters

a	(float)	x-coordinate of the rectangle by default
b	(float)	y-coordinate of the rectangle by default
c	(float)	width of the rectangle by default
d	(float)	height of the rectangle by default
r	(float)	radii for all four corners
tl	(float)	radius for top-left corner
tr	(float)	radius for top-right corner
br	(float)	radius for bottom-right corner
bl	(float)	radius for bottom-left corner

Return `void`

YORK UNIVERSITÉ UNIVERSITY

So… ”commands” are actually “methods”

Syntax

```
rect(a, b, c, d)
rect(a, b, c, d, r)
rect(a, b, c, d, tl, tr, br, bl)
```

Parameters

a	(float)	x-coordinate of the rectangle by default
b	(float)	y-coordinate of the rectangle by default
c	(float)	width of the rectangle by default
d	(float)	height of the rectangle by default
r	(float)	radii for all four corners
tl	(float)	radius for top-left corner
tr	(float)	radius for top-right corner
br	(float)	radius for bottom-right corner
bl	(float)	radius for bottom-left corner

Return `void`

Distinguished by their signature
(identifier + list of argument types)

```
void rect(float a, float b,
          float c, float d) {
  // …

}

void rect(float a, float b,
          float c, float d,
            float r) {
  // …

}


void rect(float a, float b,
          float c, float d
           float t1, float tr,
            float br, float bl) {
  // …

}


//…
```

Math methods

Calculation

`abs()`	Calculates the absolute value (magnitude) of a number
`ceil()`	Calculates the closest int value that is greater than or equal to the value of the parameter
`constrain()`	Constrains a value to not exceed a maximum and minimum value
`dist()`	Calculates the distance between two points
`exp()`	Returns Euler's number *e* (2.71828...) raised to the power of the `value` parameter
`floor()`	Calculates the closest int value that is less than or equal to the value of the parameter
`lerp()`	Calculates a number between two numbers at a specific increment
`log()`	Calculates the natural logarithm (the base-*e* logarithm) of a number
`mag()`	Calculates the magnitude (or length) of a vector
`map()`	Re-maps a number from one range to another
`max()`	Determines the largest value in a sequence of numbers
`min()`	Determines the smallest value in a sequence of numbers
`norm()`	Normalizes a number from another range into a value between 0 and 1
`pow()`	Facilitates exponential expressions
`round()`	Calculates the integer closest to the `value` parameter
`sq()`	Squares a number (multiplies a number by itself)
`sqrt()`	Calculates the square root of a number

https://processing.org/reference/#math

Math methods

Trigonometry

`acos()`	The inverse of `cos()`, returns the arc cosine of a value
`asin()`	The inverse of `sin()`, returns the arc sine of a value
`atan2()`	Calculates the angle (in radians) from a specified point to the coordinate origin as measured from the positive x-axis
`atan()`	The inverse of `tan()`, returns the arc tangent of a value
`cos()`	Calculates the cosine of an angle
`degrees()`	Converts a radian measurement to its corresponding value in degrees
`radians()`	Converts a degree measurement to its corresponding value in radians
`sin()`	Calculates the sine of an angle
`tan()`	Calculates the ratio of the sine and cosine of an angle

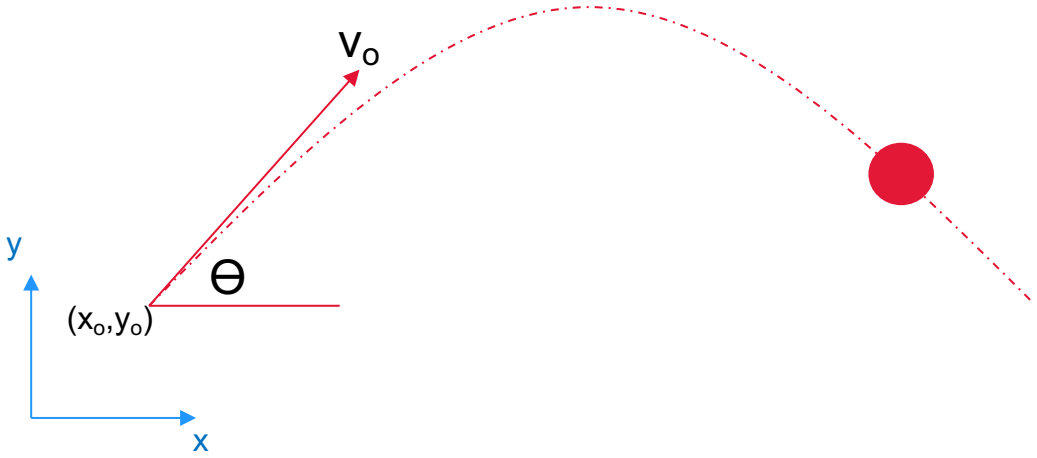
https://processing.org/reference/#math

Calculate basic projectile motion?

$$x_t = x_0 + v_0 t \cos(\theta)$$

$$y_t = y_0 + v_0 t \sin(\theta) - \frac{1}{2} g t^2$$

```
final float GRAVITY = 9.8; // m per sec squared

// assume x0,y0,v0,t and theta are all declared as floats
// and are set to have some initial values…

float x = x0 + v0*t*cos(theta);
float y = y0 + v0*t*sin(theta) + 0.5*GRAVITY*pow(t, 2) ;
```



Syntax	`cos(angle)`
Parameters	**`angle`** `(float)` an angle in radians
Return	`float`

Syntax	`sin(angle)`
Parameters	**`angle`** `(float)` an angle in radians
Return	`float`

Syntax	`radians(degrees)`
Parameters	**`degrees`** `(float)` degree value to convert to radians
Return	`float`

Syntax	`pow(n, e)`
Parameters	**`n`** `(float)` base of the exponential expression **`e`** `(float)` power by which to raise the base
Return	`float`

The String Type

- This is a non-primitive type (often confused as a primitive type because we can directly assign a literal to it

- Literal string (specified as many characters in “”):

e.g. `“Hello World”`
`“Hello Terminal”`
`“EECS 1710”`

- We can output strings using print & println

```
print(“Hello World”);
println(“EECS 1710”);
```

- How do we create a variable that is a String?

```
String myString;              // declaration
myString = “EECS 1710”;       // assignment
```

Strings can use + operator!

```
“Hello ” + “EECS 1710”   →   “Hello EECS 1710”

“Hello ” + 6.5           →   “Hello 6.5”
```

```
int rectW = 3;
int rectH = 8;
int area = rectArea(rectW,rectH);
print(“Area = ” + area);
```

→ prints: “Area = 24” to console

YORK UNIVERSITÉ UNIVERSITY

String Expressions

- We can use the '+' operator on Strings to join them together!

```
String strWorld = "World";
String str = "Hello" + " " + strWorld;
String str2 = str + "\n" + "EECS" + 1710;
```

- We can join any type to a string using a string expression
  - The type will be converted automatically to a string

other String methods in Processing?

```
String str = “    hello world  ”;
print(trim(str));
```

String Functions

`join()`	Combines an array of `Strings` into one `String`, each separated by the character(s) used for the `separator` parameter
`matchAll()`	This function is used to apply a regular expression to a piece of text
`match()`	The function is used to apply a regular expression to a piece of text, and return matching groups (elements found inside parentheses) as a `String` array
`nf()`	Utility function for formatting numbers into strings
`nfc()`	Utility function for formatting numbers into strings and placing appropriate commas to mark units of 1000
`nfp()`	Utility function for formatting numbers into strings
`nfs()`	Utility function for formatting numbers into strings
`splitTokens()`	The `splitTokens()` function splits a `String` at one or many character "tokens"
`split()`	The `split()` function breaks a string into pieces using a character or string as the divider
`trim()`	Removes whitespace characters from the beginning and end of a `String`

Remove whitespace (spaces, tabs, newlines)

Formats numbers into strings (e.g. with nearest decimal places, commas, etc

YORK UNIVERSITÉ UNIVERSITY