EECS 1710
Programming for Digital Media

Week 3 :: Expressions & Operators

YORK U
UNIVERSITÉ
UNIVERSITY

This Week

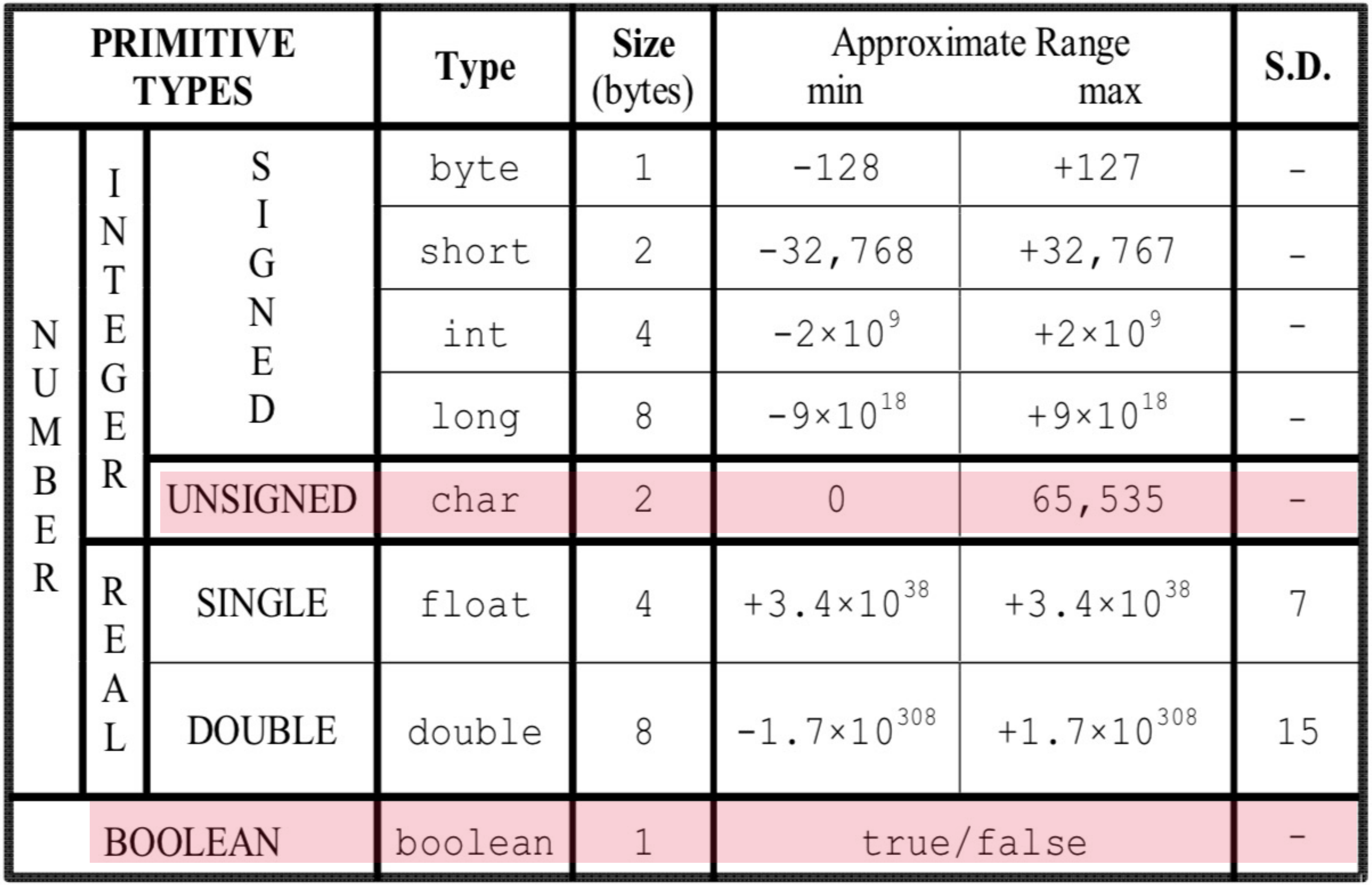
Lecture 5 (expressions/operators):

- 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 type)*
- *more drawing methods*
- *math-based methods*
- *Strings & string methods*

Primitive types (summary)



PRIMITIVE TYPES			Type	Size (bytes)	Approximate Range min	max	S.D.
NUMBER	INTEGER	SIGNED	byte	1	-128	+127	-
			short	2	-32,768	+32,767	-
			int	4	-2×10^{9}	$+2\times10^{9}$	-
			long	8	-9×10^{18}	$+9\times10^{18}$	-
		UNSIGNED	char	2	0	65,535	-
	REAL	SINGLE	float	4	$+3.4\times10^{38}$	$+3.4\times10^{38}$	7
		DOUBLE	double	8	-1.7×10^{308}	$+1.7\times10^{308}$	15
BOOLEAN			boolean	1	true/false		-

The Boolean Type (boolean)

- Stores the result of a condition
- Has only two possible values, `true` or `false`

  (can think of this as a pure binary type)
- `true` and `false` are reserved words
- Boolean variables are not integers !

- Declaration & Assignment:

```
boolean myBool;
myBool = true;
myBool = false;
```

The Character Type (char)

- A `char` is a letter, digit, or symbol
- Examples:

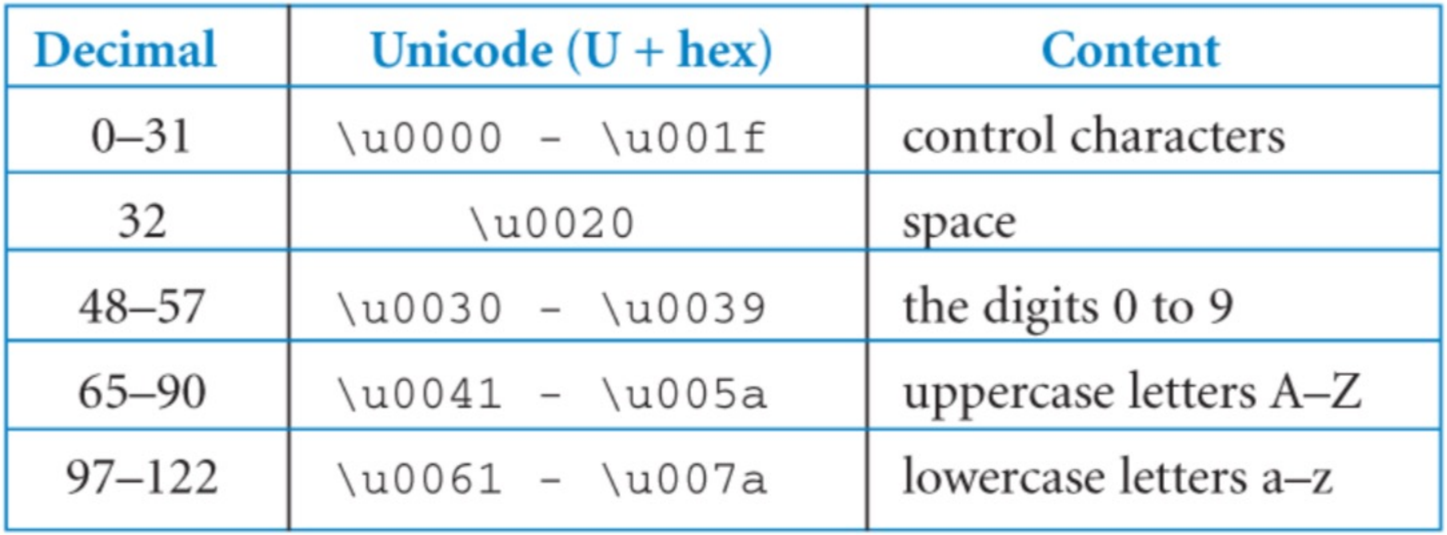
  1 @ a A b B & *] { ~ %

- Stores a code for a character, not the typeface itself
- The codes for English use ASCII1
- `char` is stored as an (unsigned) integer type

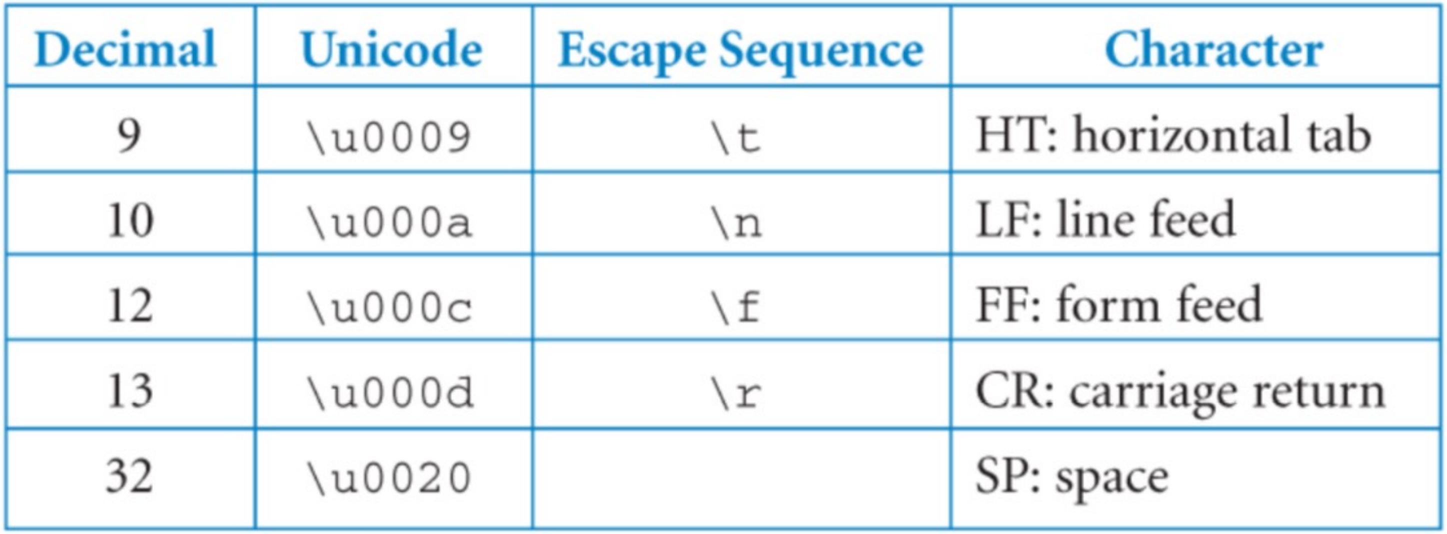
- Numeric coding of characters uses the *Unicode* character set
- Unicode has 64K codes (see following slides)

[1] ASCII codes are the first 256 entries in the Unicode character set. Try Wikipedia for more details.

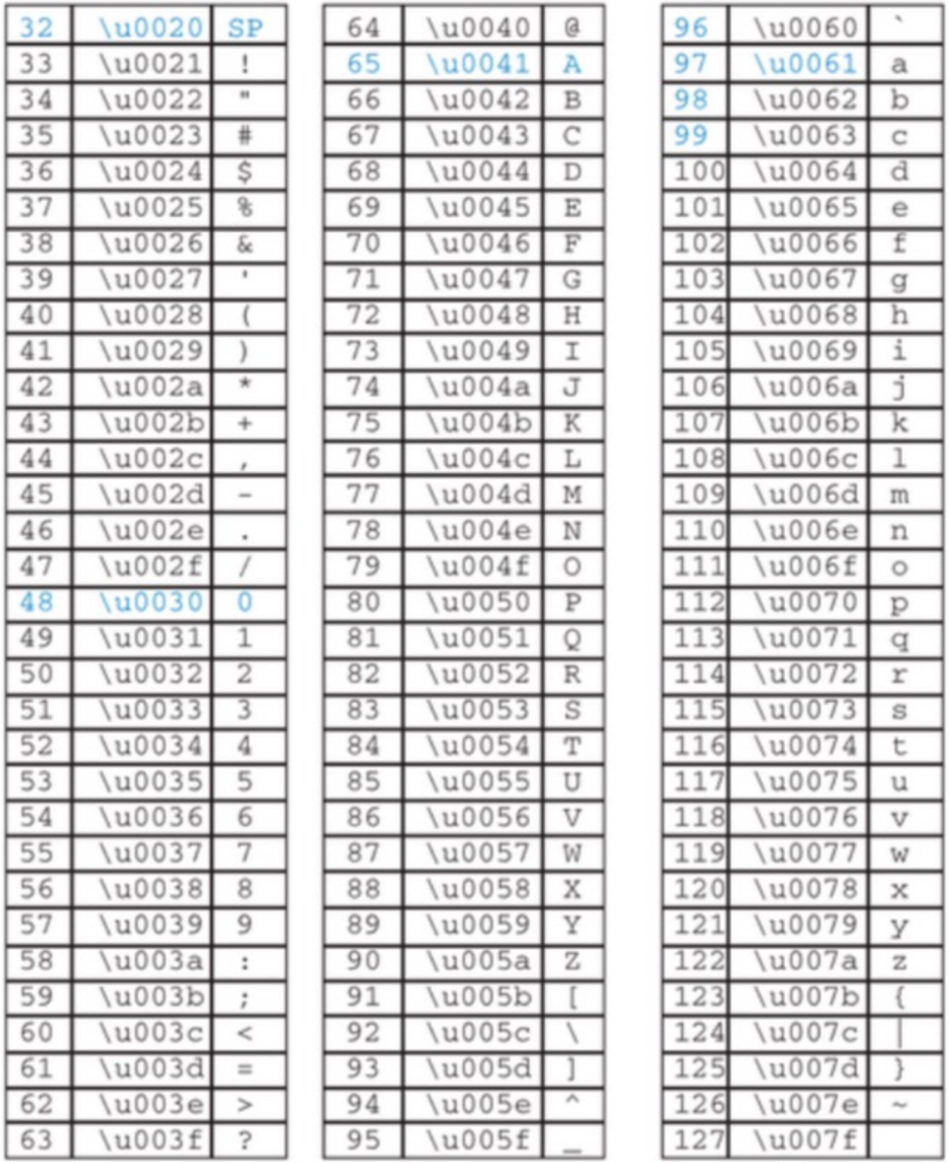
Unicodes



Decimal	Unicode (U + hex)	Content
0–31	`\u0000 - \u001f`	control characters
32	`\u0020`	space
48–57	`\u0030 - \u0039`	the digits 0 to 9
65–90	`\u0041 - \u005a`	uppercase letters A–Z
97–122	`\u0061 - \u007a`	lowercase letters a–z



Decimal	Unicode	Escape Sequence	Character
9	`\u0009`	`\t`	HT: horizontal tab
10	`\u000a`	`\n`	LF: line feed
12	`\u000c`	`\f`	FF: form feed
13	`\u000d`	`\r`	CR: carriage return
32	`\u0020`		SP: space

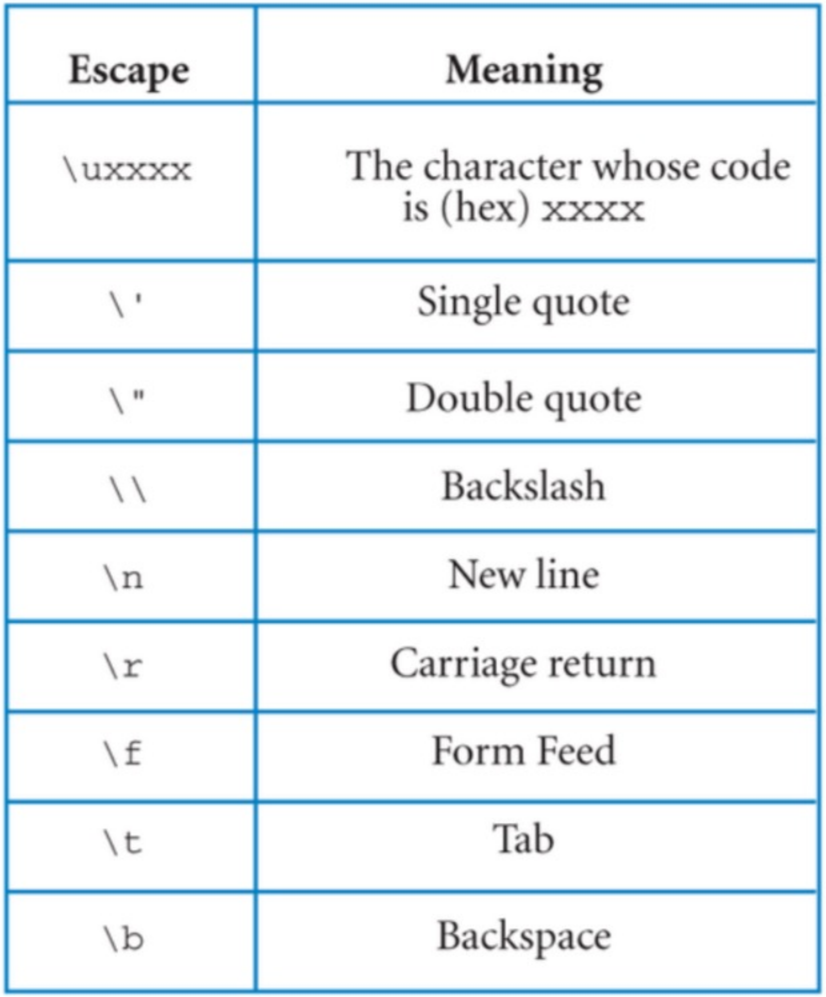


32	\u0020	SP	64	\u0040	@	96	\u0060	`
33	\u0021	!	65	\u0041	A	97	\u0061	a
34	\u0022	"	66	\u0042	B	98	\u0062	b
35	\u0023	#	67	\u0043	C	99	\u0063	c
36	\u0024	$	68	\u0044	D	100	\u0064	d
37	\u0025	%	69	\u0045	E	101	\u0065	e
38	\u0026	&	70	\u0046	F	102	\u0066	f
39	\u0027	'	71	\u0047	G	103	\u0067	g
40	\u0028	(	72	\u0048	H	104	\u0068	h
41	\u0029	)	73	\u0049	I	105	\u0069	i
42	\u002a	*	74	\u004a	J	106	\u006a	j
43	\u002b	+	75	\u004b	K	107	\u006b	k
44	\u002c	,	76	\u004c	L	108	\u006c	l
45	\u002d	-	77	\u004d	M	109	\u006d	m
46	\u002e	.	78	\u004e	N	110	\u006e	n
47	\u002f	/	79	\u004f	O	111	\u006f	o
48	\u0030	0	80	\u0050	P	112	\u0070	p
49	\u0031	1	81	\u0051	Q	113	\u0071	q
50	\u0032	2	82	\u0052	R	114	\u0072	r
51	\u0033	3	83	\u0053	S	115	\u0073	s
52	\u0034	4	84	\u0054	T	116	\u0074	t
53	\u0035	5	85	\u0055	U	117	\u0075	u
54	\u0036	6	86	\u0056	V	118	\u0076	v
55	\u0037	7	87	\u0057	W	119	\u0077	w
56	\u0038	8	88	\u0058	X	120	\u0078	x
57	\u0039	9	89	\u0059	Y	121	\u0079	y
58	\u003a	:	90	\u005a	Z	122	\u007a	z
59	\u003b	;	91	\u005b	[	123	\u007b	{
60	\u003c	<	92	\u005c	\	124	\u007c	|
61	\u003d	=	93	\u005d	]	125	\u007d	}
62	\u003e	>	94	\u005e	^	126	\u007e	~
63	\u003f	?	95	\u005f	_	127	\u007f	

More complete set:
https://www.rapidtables.com/code/text/unicode-characters.html

Declaration & assignment of characters:

- Character literals are recognized by single quotes surrounding a character, e.g., `'A'`
- Special characters, such a single quote itself, are represented as literals using *escape sequences*



Escape	Meaning
`\uxxxx`	The character whose code is (hex) xxxx
`\'`	Single quote
`\"`	Double quote
`\\`	Backslash
`\n`	New line
`\r`	Carriage return
`\f`	Form Feed
`\t`	Tab
`\b`	Backspace

```
// declaration
   char myChar;

// standard characters
   myChar = ‘a’;
   myChar = ‘A’;
   myChar = ‘$’;
   myChar = ‘)’;
   myChar = ‘>’;

// using escape characters
   myChar = ‘\’’;              // single quote ‘
   myChar = ‘\”’;              // double quote “
   myChar = ‘\\’;              // backslash /
   myChar = ‘\n’;              // new line

// using unicodes
   myChar = ‘\u0061’;          // ’a’
   myChar = ‘\u0041’;          // ‘A’
   myChar = ‘\u0024’;          // ‘$’
   myChar = ‘\u007c’;          // ‘[‘
   myChar = ‘\u0151’;          // ‘ő’
   myChar = ‘\u03A3’;          // ‘Σ’
```

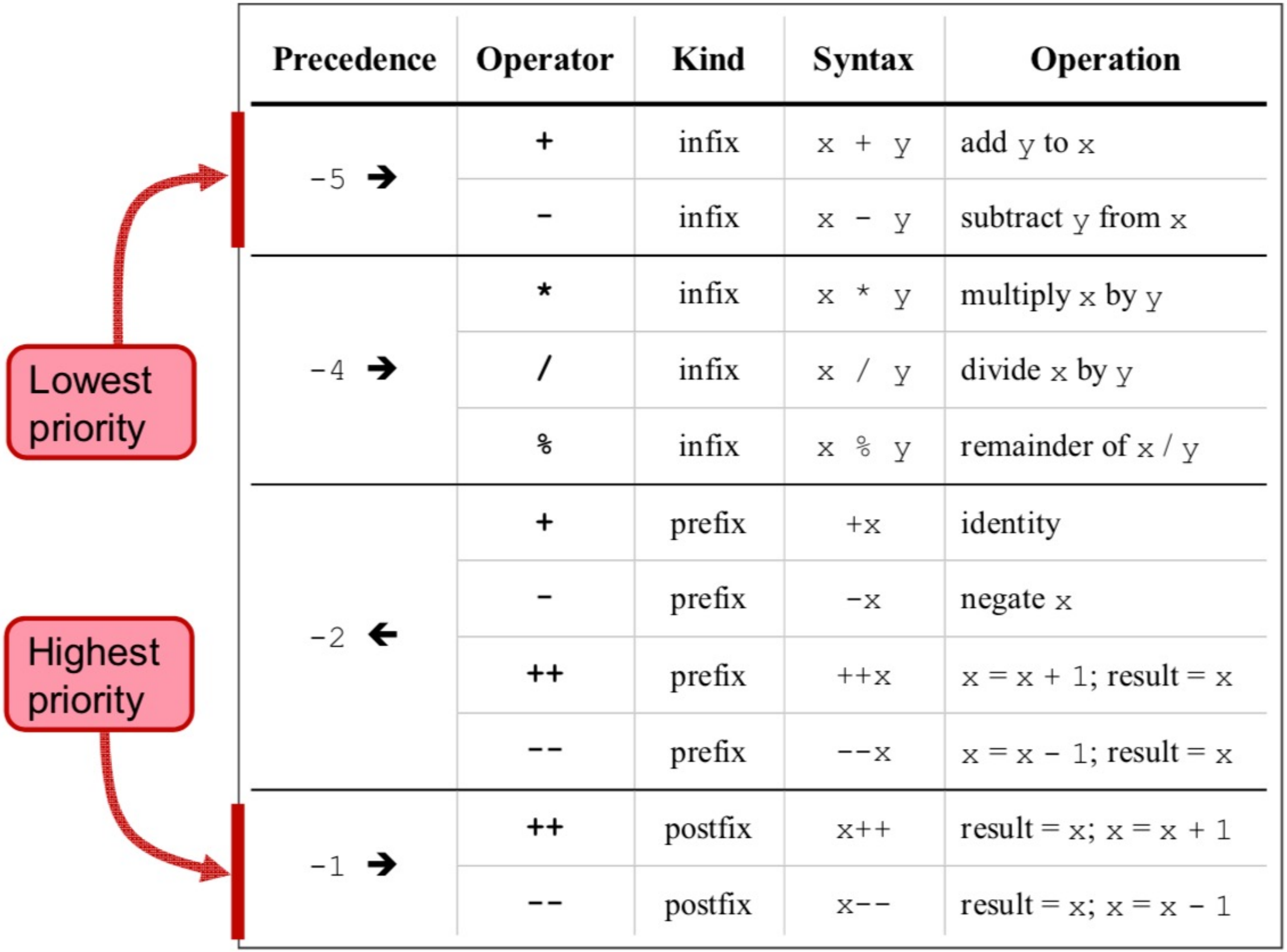
```
// booleans and chars

{
  char grade = 'B';
  char exclaim = '\u0021';
  boolean isFound = false;

  print("grade = ");
  print(grade);
  println(exclaim);

  int gradeNum = grade;
  print("gradeNum = ");
  println(gradeNum);

  println();
  print("isFound = ");
  println(isFound);


}
```

Expressions & Operators

- *Expressions* involve one or more data values that appear together with *operators*
- *Operators* define specific actions on data
- *Operators* are usually specific to a given type
  - E.g. standard operators + - * / in general, work on integer and real types
  - Their function may differ slightly depending on the type they are operating on

- Expressions are typically processed from left to right (though there are exceptions that give some operators precedence over others)
- Parenthesis in an expression can override operator precedence

`int` arithmetic operators (summary)

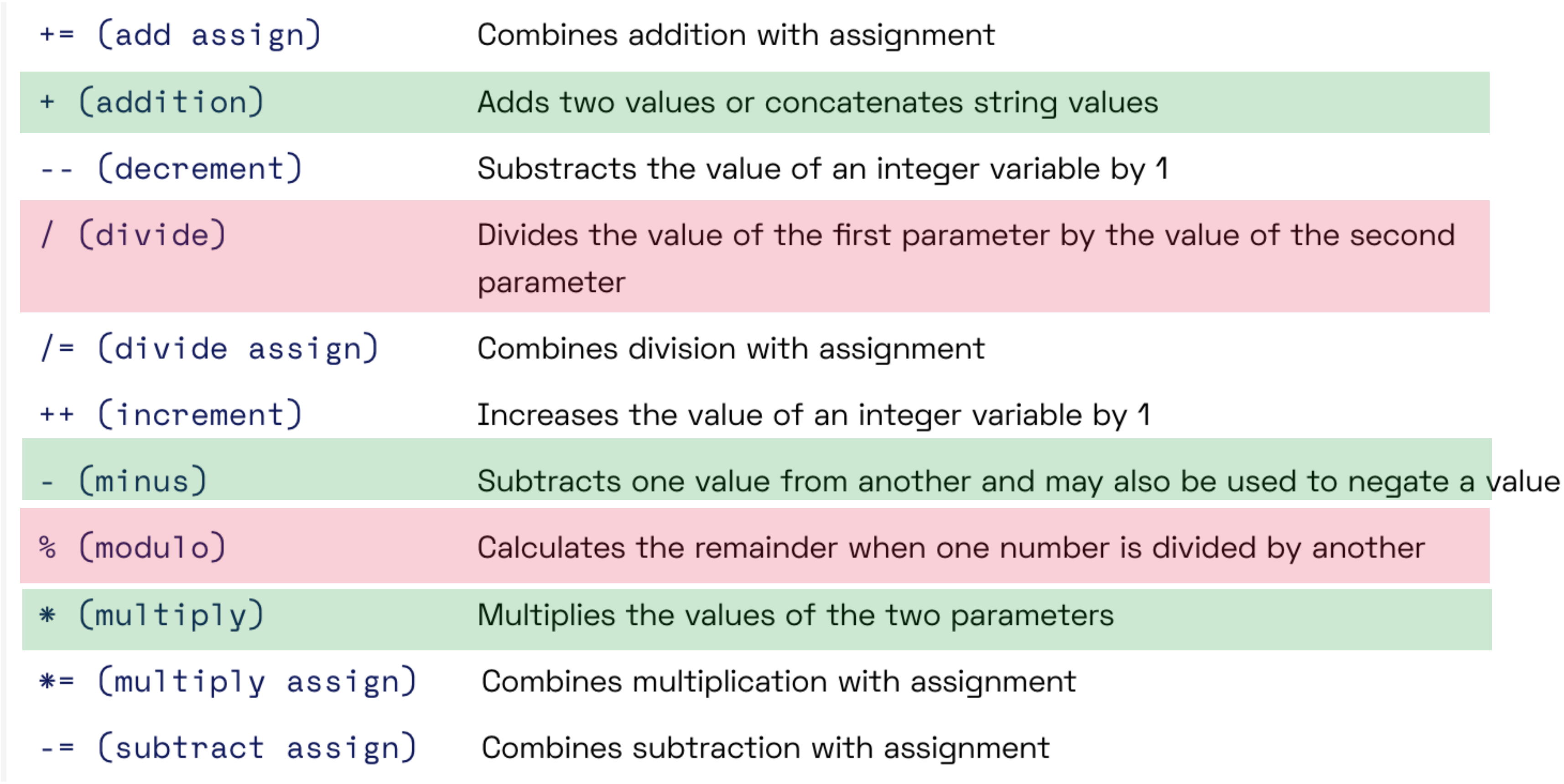


Precedence	Operator	Kind	Syntax	Operation
-5 →	+	infix	x + y	add y to x
	-	infix	x - y	subtract y from x
-4 →	*	infix	x * y	multiply x by y
	/	infix	x / y	divide x by y
	%	infix	x % y	remainder of x / y
-2 ←	+	prefix	+x	identity
	-	prefix	-x	negate x
	++	prefix	++x	x = x + 1; result = x
	--	prefix	--x	x = x - 1; result = x
-1 →	++	postfix	x++	result = x; x = x + 1
	--	postfix	x--	result = x; x = x - 1

Lowest priority

Highest priority

In Processing:



`+= (add assign)`	Combines addition with assignment
`+ (addition)`	Adds two values or concatenates string values
`-- (decrement)`	Substracts the value of an integer variable by 1
`/ (divide)`	Divides the value of the first parameter by the value of the second parameter
`/= (divide assign)`	Combines division with assignment
`++ (increment)`	Increases the value of an integer variable by 1
`- (minus)`	Subtracts one value from another and may also be used to negate a value
`% (modulo)`	Calculates the remainder when one number is divided by another
`* (multiply)`	Multiplies the values of the two parameters
`*= (multiply assign)`	Combines multiplication with assignment
`-= (subtract assign)`	Combines subtraction with assignment

Notes (1)

- Division (/)
  - For integer operands, the result is an integer rounded toward zero, so

```
 5 / 4 → 1
-5 / 4 → -1
```

  - For real operands, the result is a real

```
 5.0 / 4.0 → 1.25
-5.0 / 4.0 → -1.25
```

Example

```
// block 5: division caveats

{
  int oneHour = 60;      // mins
  int onePres = 20;      // mins per presentation

  int presPerHour = oneHour/onePres;

  print("oneHour = "); print(oneHour); println(" mins");
  print("onePres = "); print(onePres); println(" mins");
  print("presPerHour = "); print(presPerHour); println();

}
```

Notes (2)

- Remainder (`%`) is the remainder after division
- I.e., `a % b` yields `a – (a / b) * b`

```
 5 % 3   →   2
 5 % -3  →   2
-5 % 3   →  -2
```

Note: the sign is always the same as the divisor

Resolving numeric expressions

- A numeric expression is generally found to the right of an assignment statement
- The result resolves down to a single numeric value that is then assigned to a variable of a numeric type

- The literals and variables used in such an expression must all be numeric also, and must be compatible with the variable being assigned to

Example (operator precedence)

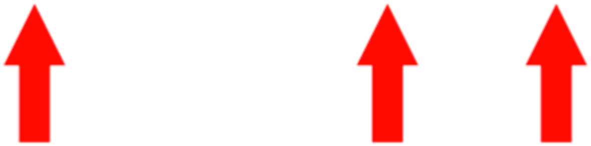
```
5 + (4 - 3) / 5 - 2 * 3 % 4
```

Example (operator precedence)

```
  5 + (4 - 3) / 5 - 2 * 3 % 4
= 5 + 1 / 5 - 2 * 3 % 4
```

Example (operator precedence)

```
    5 + (4 - 3) / 5 - 2 * 3 % 4
  = 5 + 1 / 5 - 2 * 3 % 4
```



Example (operator precedence)

```
  5 + (4 - 3) / 5 - 2 * 3 % 4
= 5 + 1 / 5 - 2 * 3 % 4
= 5 + 0 - 2 * 3 % 4
```

Example (operator precedence)

```
  5 + (4 - 3) / 5 - 2 * 3 % 4
= 5 + 1 / 5 - 2 * 3 % 4
= 5 + 0 - 2 * 3 % 4
```

Example (operator precedence)

```
  5 + (4 - 3) / 5 - 2 * 3 % 4
= 5 + 1 / 5 - 2 * 3 % 4
= 5 + 0 - 2 * 3 % 4
= 5 + 0 - 6 % 4
```

Example (operator precedence)

```
  5 + (4 - 3) / 5 - 2 * 3 % 4
= 5 + 1 / 5 - 2 * 3 % 4
= 5 + 0 - 2 * 3 % 4
= 5 + 0 - 6 % 4
= 5 + 0 - 2
```

Example (operator precedence)

```
    5 + (4 - 3) / 5 - 2 * 3 % 4
  = 5 + 1 / 5 - 2 * 3 % 4
  = 5 + 0 - 2 * 3 % 4
  = 5 + 0 - 6 % 4
  = 5 + 0 - 2
```

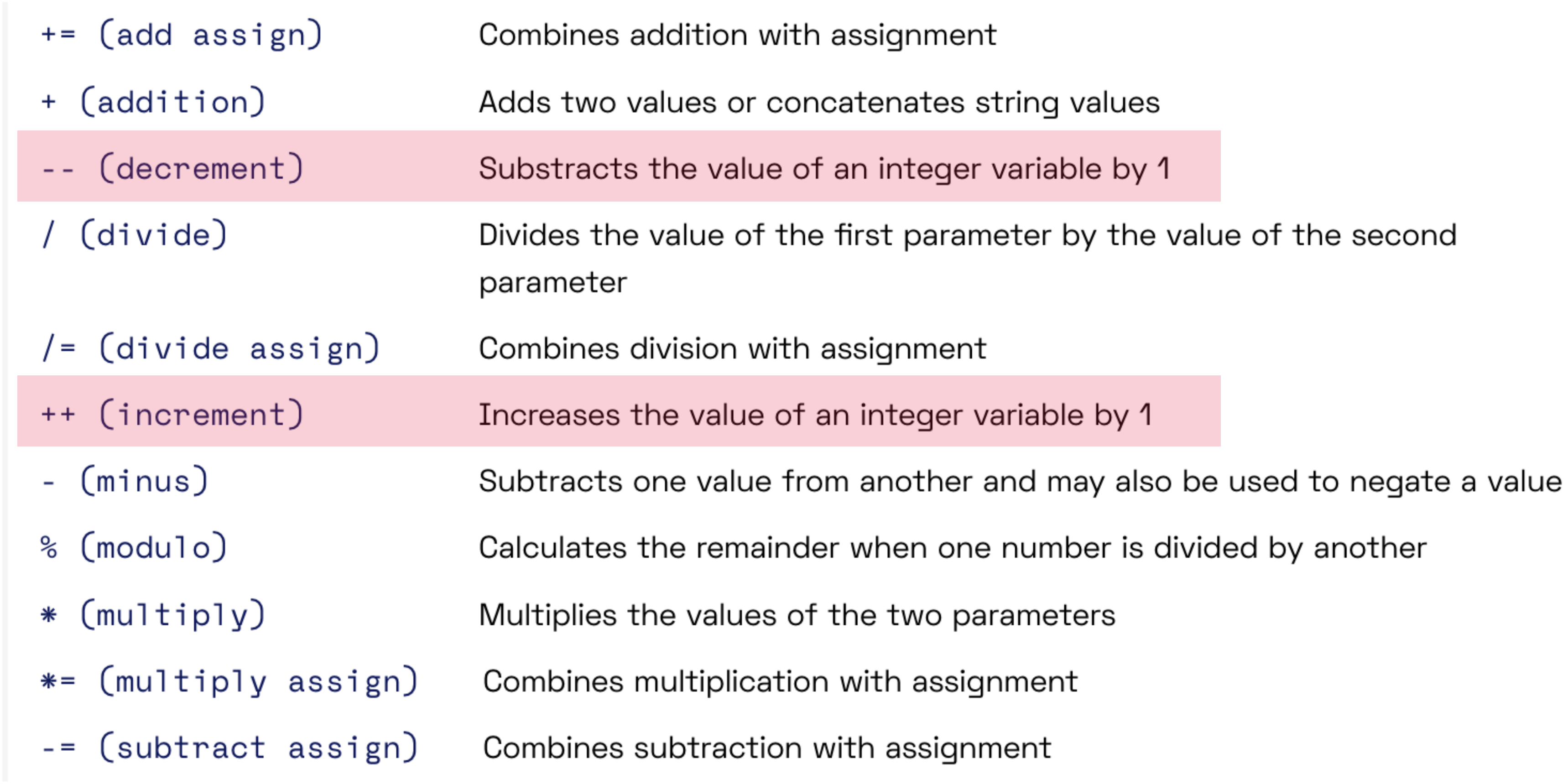
Example (operator precedence)

```
  5 + (4 - 3) / 5 - 2 * 3 % 4
= 5 + 1 / 5 - 2 * 3 % 4
= 5 + 0 - 2 * 3 % 4
= 5 + 0 - 6 % 4
= 5 + 0 - 2
= 5 - 2
```

Example (operator precedence)

```
  5 + (4 - 3) / 5 - 2 * 3 % 4
= 5 + 1 / 5 - 2 * 3 % 4
= 5 + 0 - 2 * 3 % 4
= 5 + 0 - 6 % 4
= 5 + 0 - 2
= 5 - 2
= 3
```

Increment/Decrement operators



`+= (add assign)`	Combines addition with assignment
`+ (addition)`	Adds two values or concatenates string values
`-- (decrement)`	Substracts the value of an integer variable by 1
`/ (divide)`	Divides the value of the first parameter by the value of the second parameter
`/= (divide assign)`	Combines division with assignment
`++ (increment)`	Increases the value of an integer variable by 1
`- (minus)`	Subtracts one value from another and may also be used to negate a value
`% (modulo)`	Calculates the remainder when one number is divided by another
`* (multiply)`	Multiplies the values of the two parameters
`*= (multiply assign)`	Combines multiplication with assignment
`-= (subtract assign)`	Combines subtraction with assignment

Notes (3)

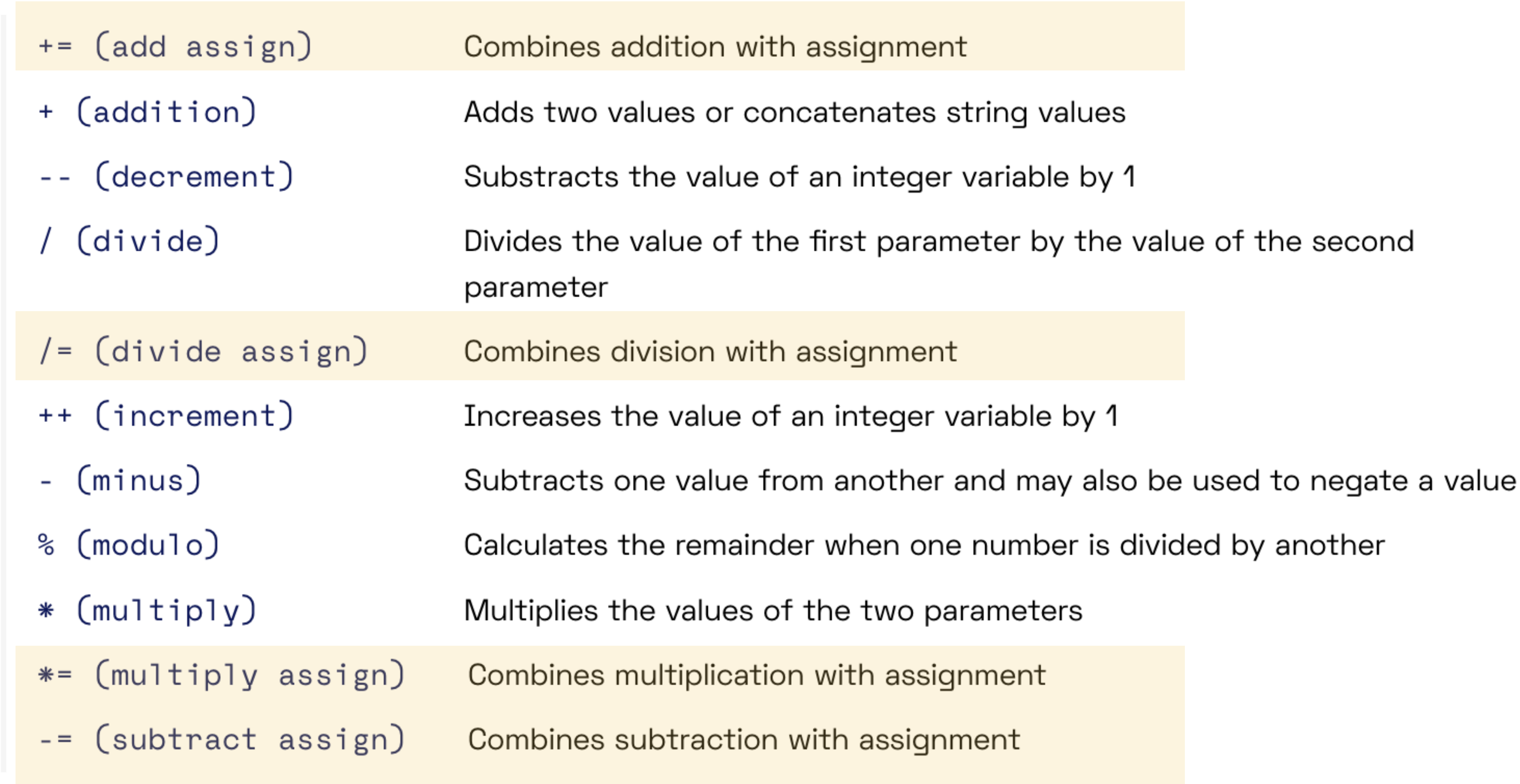
- Auto increment (++), auto decrement (−−)
  - Prefix:
    - Increment/decrement *before* using in an expression
  - Postfix:
    - Increment/decrement *after* using in an expression
  - Example:
    - If x is 5, `z = ++x` leads to z being 6
    - If x is 5, `z = x++` leads to z being 5
    - In both cases above, x becomes 6

Example: Drawing with increments

```
int radius = 20;
float alpha = 255;
int centreX = 150;
int centreY = 150;

void setup(){
  size(300,300);
  ellipseMode(CENTER);
}

void draw() {
  background(255,255,255);
  ellipse(centreX,centreY,radius,radius);
  radius++;
  stroke(0,0,0,alpha);
  alpha--;       // OR  alpha = alpha * 0.99;
}
```

```
void mousePressed() {
 centreX = mouseX;
 centreY = mouseY;
 radius = 20;    // reset
 alpha = 255;    // reset
}
```

Step-wise operation & assignment



`+= (add assign)`	Combines addition with assignment
`+ (addition)`	Adds two values or concatenates string values
`-- (decrement)`	Substracts the value of an integer variable by 1
`/ (divide)`	Divides the value of the first parameter by the value of the second parameter
`/= (divide assign)`	Combines division with assignment
`++ (increment)`	Increases the value of an integer variable by 1
`- (minus)`	Subtracts one value from another and may also be used to negate a value
`% (modulo)`	Calculates the remainder when one number is divided by another
`* (multiply)`	Multiplies the values of the two parameters
`*= (multiply assign)`	Combines multiplication with assignment
`-= (subtract assign)`	Combines subtraction with assignment

examples

```
myvar += 340;       (equivalent to)      myvar = myVar + 340;

myvar *= 340;       (equivalent to)      myvar = myVar * 340;

myvar –= 40;        (equivalent to)      myvar = myvar – 40;

myvar /= 40;        (equivalent to)      myvar = myvar / 40;
```

What happens for mixed numeric types?

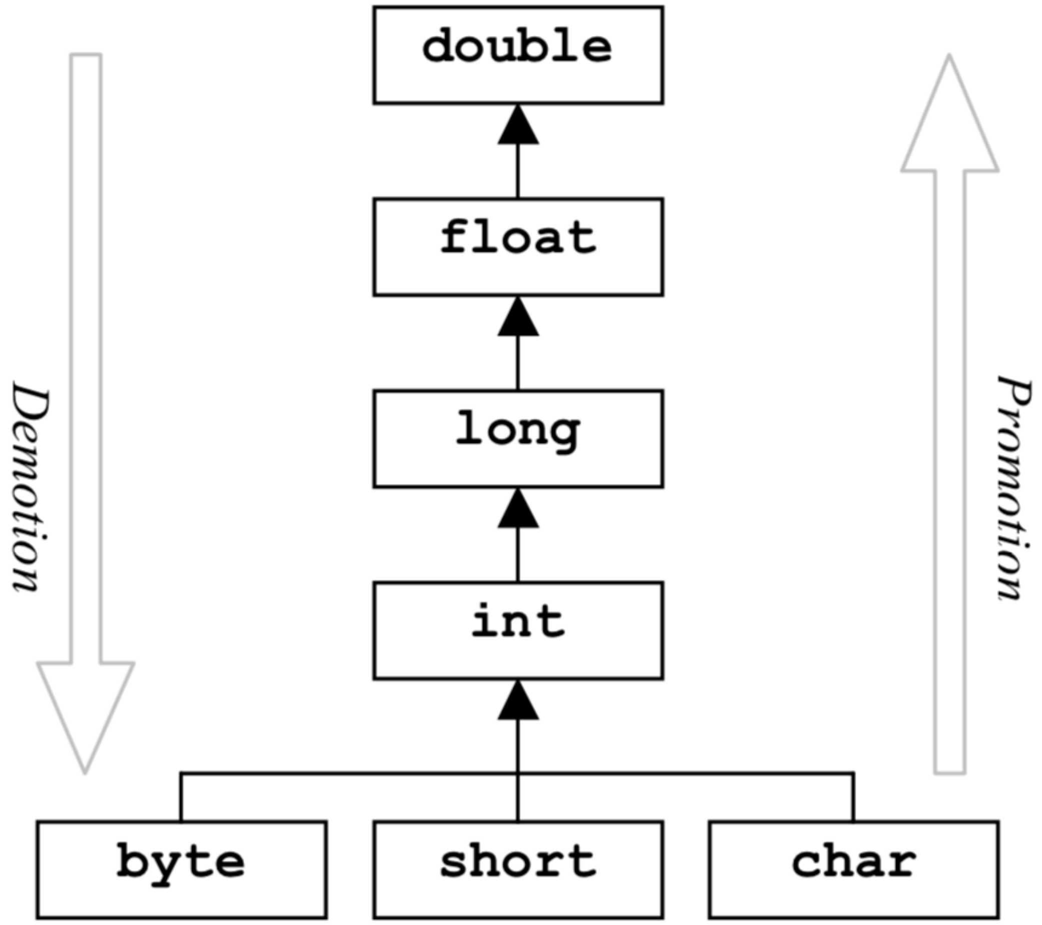
- 5 / 4.0 → ?

3 * 8.0 → ?

- 5.0 / 4 → ?

2L + 1 → ?

Promotion & Demotion



What is promotion?

- An operation will be performed according to the **widest** operand used by the operator

- i.e. all types in the operation will be automatically “promoted” to the same type as the widest operand being used, the result will also then be computed in the same type as the widest operand

Examples

- Promotion

```
float x = 1.5f;
double xSquared = x * x;
```

The result is a `float`, but it is automatically promoted to a `double` when assigned to `xSquared`.

- Demotion

```
double x = 1.5;
float xSquared = (float)(x * x);
```

The initial result is a `double`, but it is cast down (i.e., demoted) to a `float` when assigned to `xSquared`.

Notes

- The cast operation has a precedence that is higher than `*` but less than `++`
- The `=` operator has the lowest precedence of all operators
- There are shorthand operators to combine assignment with an operator:

`x op= y` is shorthand for `x = x op y`

- E.g., `x += 1` is like `x = x + 1`

Example

```
int iVar = 15;
long lVar = 2;
float fVar = 7.6f - iVar / lVar;
double dVar = 1L / lVar + fVar / lVar;
int result = 100 * dVar;
```

What is the value of `result`?

Constants

- Variables can have different values assigned throughout the course of a program
- Sometimes we want a variable not to change
    - E.g. acceleration due to gravity g = 9.81 m/s
- There is a keyword final that can be used at declaration (only) to force a variable to stay constant after it is assigned.

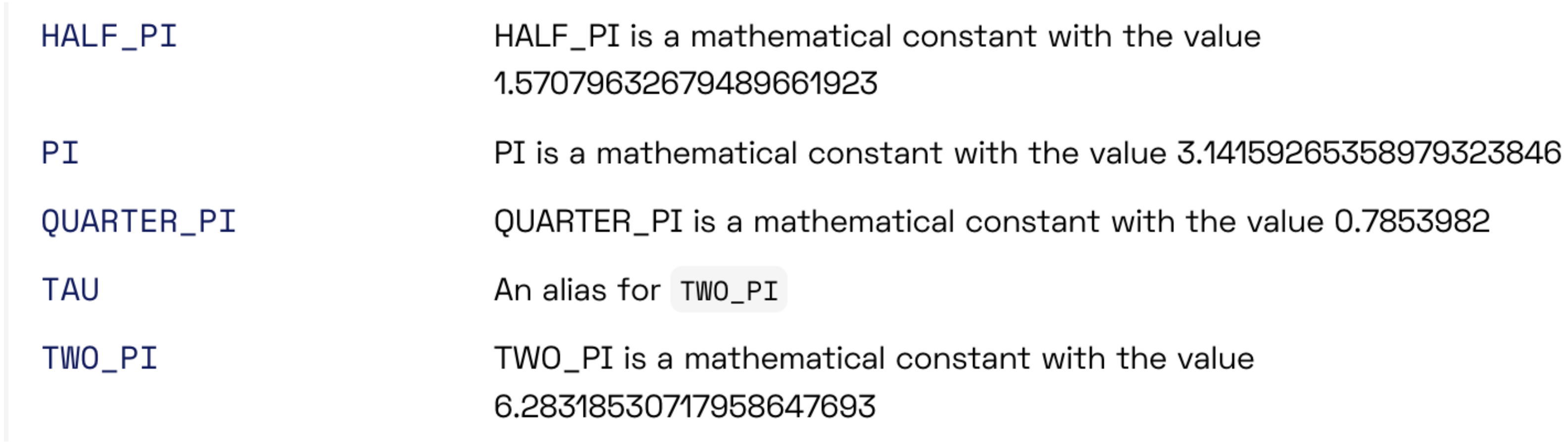
```
final double GRAVITY;
GRAVITY = 9.81;
```

- For style purposes, we capitalize the identifier of a constant – this way we instantly know in our code whether a variable is constant or not

Constants

- Are variables that can only be set once!
- Use keyword “final” when declaring
  - after assignment occurs, this means we cannot re-assign

- “built-in” constants



HALF_PI	HALF_PI is a mathematical constant with the value 1.57079632679489661923
PI	PI is a mathematical constant with the value 3.14159265358979323846
QUARTER_PI	QUARTER_PI is a mathematical constant with the value 0.7853982
TAU	An alias for `TWO_PI`
TWO_PI	TWO_PI is a mathematical constant with the value 6.28318530717958647693

Handling constants

- Replace all *magic numbers* (literals) in your program with *finals*
- Instead of

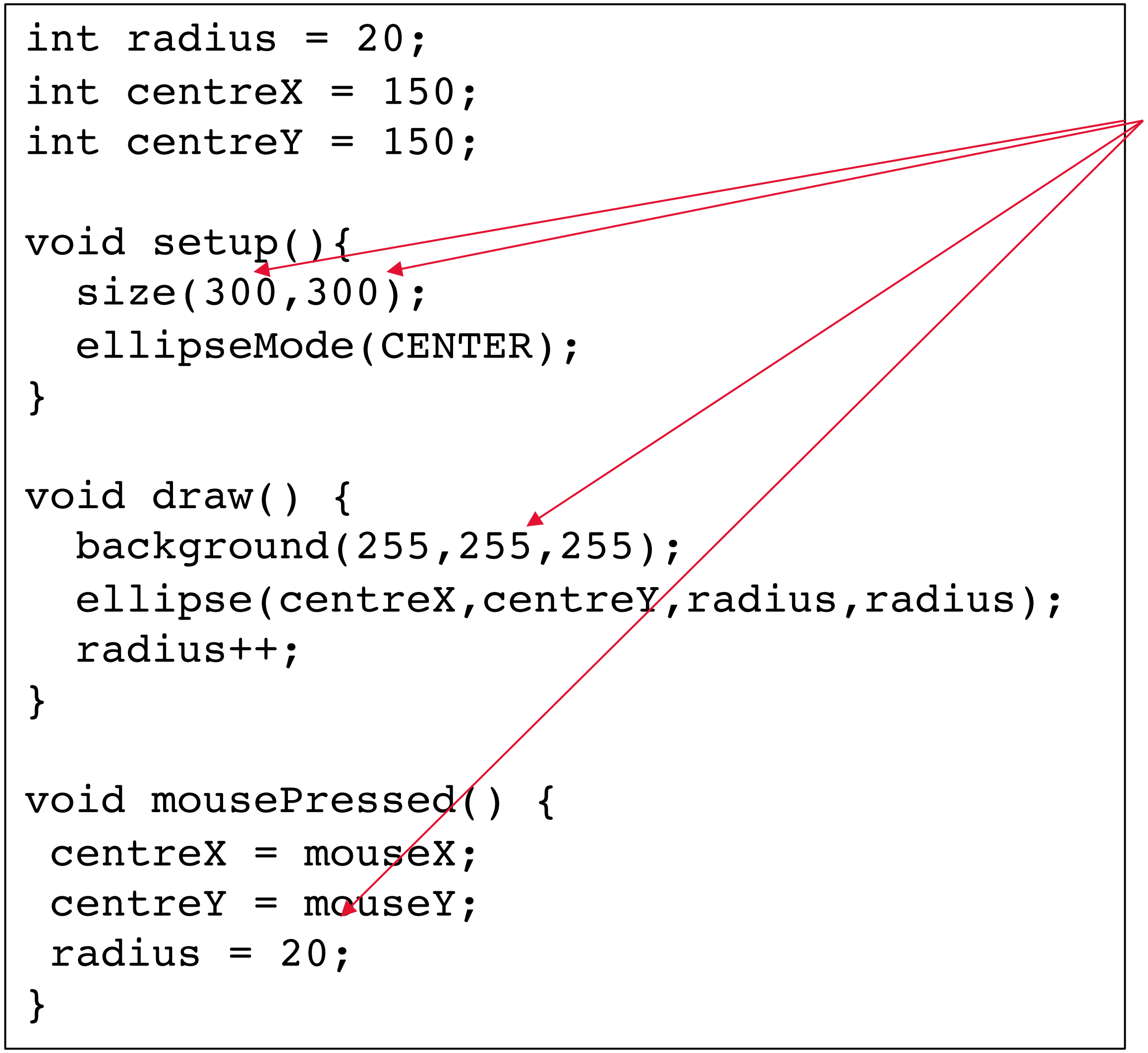
```
width = width / 12;
```

Note the style for naming constants

- Write

```
final int INCH_PER_FOOT = 12;
width = width / INCH_PER_FOOT;
```

- Advantages of finals versus literals:
  - The final has a name and, thus, is self-documenting
  - Avoids inadvertently changing the value



```
int radius = 20;
int centreX = 150;
int centreY = 150;

void setup(){
  size(300,300);
  ellipseMode(CENTER);
}

void draw() {
  background(255,255,255);
  ellipse(centreX,centreY,radius,radius);
  radius++;
}

void mousePressed() {
 centreX = mouseX;
 centreY = mouseY;
 radius = 20;
}
```

? Magic Numbers ?

A magic number is a literal value that is floating in your code

Reading the code it is not immediately obvious where the value comes from or why it is what it is…

Not a good practice!

```
final int INIT_RADIUS = 20;     // defining constants
final int INIT_CENTERX = 150;
final int INIT_CENTERY = 150;
final int APP_SIZE = 300;

int radius = INIT_RADIUS;
int centreX = INIT_CENTERX;
int centreY = INIT_CENTERY;

void setup(){
  size(APP_SIZE, APP_SIZE);
  ellipseMode(CENTER);
}

void draw() {
  background(255,255,255);
  ellipse(centreX,centreY,radius,radius);
  radius++;
}

void mousePressed() {
 centreX = mouseX;
 centreY = mouseY;
 radius = INIT_RADIUS;
}
```

Making magic numbers constants, makes them more readable in the code

A note on style conventions

- sketch naming:
  - Use title case (capitalize first letter of each word in identifier, no spaces) unless class name is an acronym
- variable & method naming:
  - Use lowercase letters, except…
    - For multi-word names, capitalize the first letter of each subsequent word (no spaces)
    - E.g., main , equals , toString , isLeapYear
- block layouts:
  - Braces must align vertically and the all statements must be left justified and indented by one tab position
- no magic numbers!
  - use constants with intuitive names