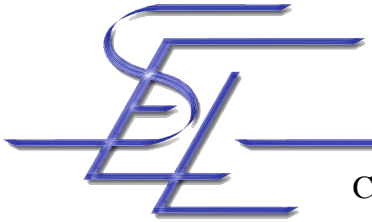
Identifiers

- *Identifiers* are the words a programmer uses in a program
- They are used to give names to things
- Example of things that can be named with identifiers:
  - Classes
  - Methods
  - Variables
  - Constants
- An identifier can be made up of *letters*, *digits*, the *underscore character* (_), and the *dollar sign* ($)
- Identifiers cannot begin with a digit
- Java is *case sensitive*, therefore `Result` and `result` are different identifiers

More on Identifiers

- Sometimes we choose identifiers ourselves when writing a program (such as `input1`, `AddTwoIntegers`)
  - We should choose identifiers that are meaningful so that it is more readable for humans
- Sometimes we are using another programmer's code, so we use the identifiers that they chose (such as `println()`)
- Often we use special identifiers called reserved words that already have a predefined meaning in the language
  - A reserved word cannot be used in any other way
  - Examples of reserved words: **`class`**, **`public`**, **`static`**, **`void`**
  - There are many others



McGill

Identifier Exercise

- Which of the following identifiers are invalid according to the Java rules for identifiers? Why are they invalid?

```
myIdentifier
_my_other_identifier
yet-another-identifier
$can
2for1
twoFor1
class
my_class
```

Variables

- A *variable* is a *placeholder* for values
  - We can store a value in a variable
  - We can use the value a variable contains to compute other values (which may be stored in other variables) or make decisions
- Each variable has a name
  - The compiler allocates memory cells to hold the value of the variable, and assigns the name chosen for the variable to these cells
  - We can then refer to the value stored in these memory cells using the variable name
  - Much easier and more practical than using the addresses of the memory cells (and less error-prone)

Variable Declarations

- Before we can store a value in a variable or use the value it contains, we have to *declare* it
- A *variable declaration* is a statement that announces that we want to create and use a new variable
  - They can occur anywhere in a method, but they are typically at the beginning
  - They must indicate the name and the type of the new variable
- Once a variable has been declared, we can store a value in it or use the value it contains
- If you try to use a variable before it is declared, the compiler will generate an error

Syntax for Declaring Variables

modifier	type	identifier	= value	;
(optional)	(mandatory)	(mandatory)	(optional)	(mand.)

- Example modifiers: **`final`**, **`static`**, ...
- Example types: **`int`**, **`float`**, **`char`**, ...
- Identifier: as defined previously
- Value: an expression matching the type (the expression is evaluated and its value is assigned to the variable)

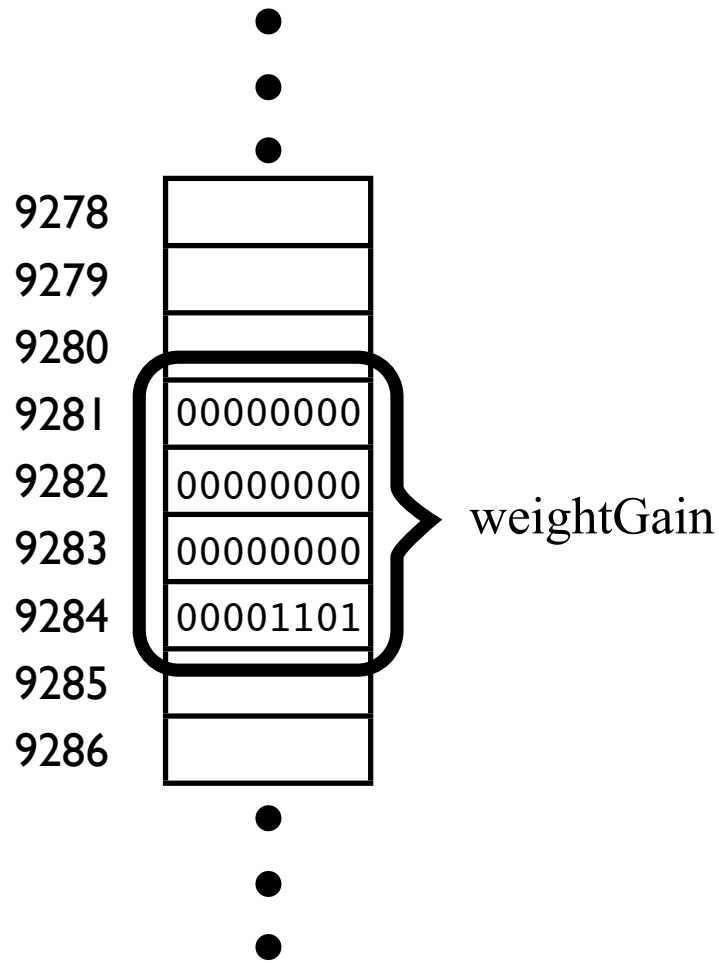
More on Variable Declarations

- Note that you can declare more than one variable with one variable declaration statement
  - The variable names are separated using commas
- Example:

```
int var1, var2, var3;
```

Variable Declaration Illustration

```
int weightGain = 13;
```



Types

- A type is a category that a variable belongs to and determines:
  - How to interpret the value it contains
  - What are the possible values it can contain
- All variables have a type
- The type of a variable provides an answer to the following question: should the value of the variable be considered as an integer, a real number, a character, or maybe even something else?
  - For a computer, everything is stored as a series of ones and zeroes
  - We need a way to tell what the ones and zeroes represent and how they should be interpreted

Java Types

- There are two broad kinds of types in Java
  - *Primitive* types
  - *Reference* types
    - We will see reference types later in the course
- Primitive types represent very basic kinds of values
- They are defined by the Java compiler

Primitive Types

- There are exactly 8 primitive types in Java
  - Four of them represent integers (positive and negative whole numbers):
    - **`byte`**, **`short`**, **`int`**, **`long`**
  - Two of them represent floating point numbers (positive and negative numbers with decimal parts):
    - **`float`**, **`double`**
  - One of them represents characters:
    - **`char`**
- And one of them represents boolean values (*true* or *false*):
    - **`boolean`**

Numeric Types

- The difference between the various numeric primitive types is their size, and therefore the values they can store:

Type	Space	Literal	Minimum	Maximum
`byte`	8 bits	5	-128	127
`short`	16 bits	5	-32768	32768
`int`	32 bits	5	-2147483648	2147483647
`long`	64 bits	5	$< -9 * 10^{18}$	$> 9 * 10^{18}$
`float`	32 bits	5.2f	$+/- 3.4 * 10^{38}$ (with 7 significant digits)	
`double`	64 bits	5.2	$+/- 1.7 * 10^{308}$ (with 15 signif. digits)	

Real vs. Floating Point

- Real number: number that can be given by an infinite decimal representation
  - 3.14159265...
  - However, we would need infinite memory to store a number with infinite decimal representation
- Floating point number:
  - Approximation of a real number; can only hold a finite amount of decimal places
  - Needs only finite space (fits in a memory cell or a finite set of memory cells)

Characters

- A **`char`** variable stores a single character from the Unicode character set

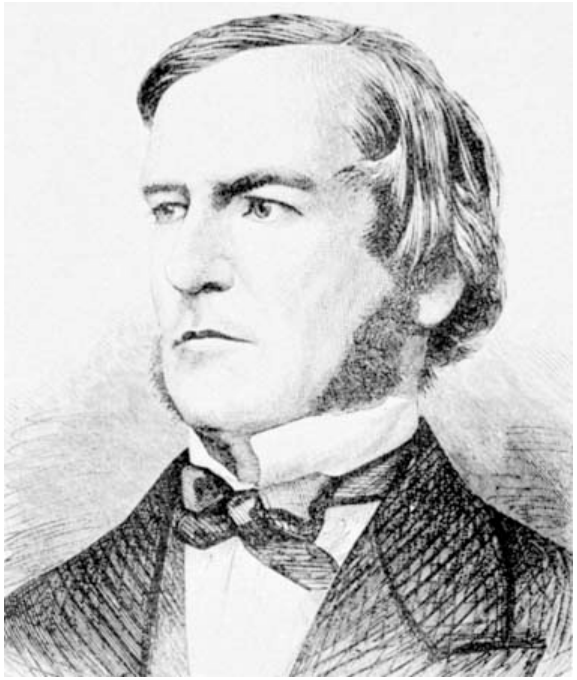
```
char gender;
gender = 'M';
```

- A character set is an ordered list of characters, and each character corresponds to a unique number
- The Unicode character set uses 16 bits (2 bytes) per character, allowing for 65536 unique characters
- It is an international character set, containing symbols and characters from many world languages
- Character literals are delimited by single quotation marks:

```
'a'  'X'  '7'  '$'  ','  '\n'
```

Boolean Values

- A boolean expression is an expression that evaluates either to *true* or to *false*
- Named after George Boole, the inventor of Boolean algebra
- Similar concept in natural language
  - "The traffic light is red "
  - This expression is either *true* or *false*



Boolean Type

- You can declare variables which are of type boolean
- A boolean variable represents a true or false condition
- A boolean can also be used to represent any two states, such as a light bulb being on or off
- The literals `true` and `false` are the only valid values for a boolean type
- Example:

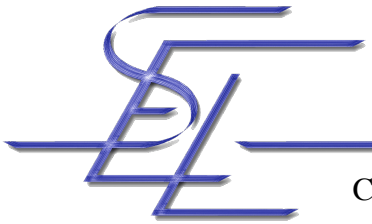
```
boolean done = false;
// Some code here
done = true;
```

The Assignment Statement

- To store a value in a variable, we use the assignment statement
- The assignment operator in Java is the = sign
- The syntax for the assignment statement is the following:

```
variable = expression;
```

  - The expression on the right of the = sign is evaluated
  - The result of evaluating the expression is stored in the variable on the left of the = sign
  - The left-hand side of the = must be a variable; it cannot be an expression
  - The = operator does not define an equation!



Reading From the Keyboard (1)

- To read values from the keyboard, we use the `Scanner` class
- First, we must declare a variable of type `Scanner`, and initialize it
  - `Scanner variable = new Scanner(System.in);`
  - `variable` can be replaced by any identifier; `keyboard` is a good choice, but there is nothing special about it
  - Everything else, however, must be *exactly* as above
- To read a value of type `int` from the keyboard, we use the following expression: `variable.nextInt()`
- variable must be replaced by the identifier you used when you declared the `Scanner` variable

Reading From the Keyboard (2)

- Thus, if you called your `Scanner` variable `foo`, you must write `foo.nextInt()` if you want to read an `int` value from the keyboard

- The expression `variable.nextInt()` (where variable is a variable of type `Scanner`) can be used as the right side of an assignment statement:

```
int input;
input = keyboard.nextInt();
```

  - The right side of the assignment statement is evaluated; this results in a value of type `int` being read from the keyboard
  - The value read from the keyboard is then stored in the variable on the left side of the assignment statement

- To read a value of type `double` instead of type `int`, replace `nextInt()` by `nextDouble()`

Displaying to the Monitor

- Two built-in commands to print on the screen:
  - `System.out.println(`….stuff to print out….`);`
    - A line-break is printed after ….stuff to print out…
  - `System.out.print(`….stuff to print out….`);`
    - Only ….stuff to print out…. is printed

More on `print` / `println`

- `println()` and `print()` each take one input (also called *parameter* or *argument*)
  - a character string:
    - `println("Hello world!");`
  - the value of a variable:
    - `println(output);`
  - the combination of both:
    - `println("The sum is " + output);`
  - a combination can have more than two parts:
    - `println("The sum is " + sum + " and the " + "difference is " + diff);`

Examples of Displaying Text

- `System.out.println("Hello world!");`
  - The character string `Hello world!` is displayed
-
```
int value = 5;
int result = 7 * value + 9;
System.out.println(result);
```
  - The value 44 is displayed
-
```
double price = 44.99;
System.out.println("This book costs "
    + price +  " dollars");
```
  - The character string `This book costs 44.99 dollars` is displayed

Countdown.java

```
class Countdown {
 public static void main(String args[]) {
    System.out.print("Three… ");
    System.out.print("Two… ");
    System.out.print("One… ");
    System.out.print("Zero… ");

    System.out.println("Liftoff!");

    System.out.println
       ("Houston, we have a problem!");
 }
}
```

What does this output?

Countdown Result

- The following is displayed when the program `Countdown` is executed:

```
Three… Two… One… Zero… Liftoff!
Houston, we have a problem!
```

Cursor ends up here!

Arithmetic Expressions

- An *expression* is a combination of *operators* and *operands*
- Arithmetic expressions compute numeric results and make use of the arithmetic operators:

Addition	`x + y`
Subtraction	`x - y`
Multiplication	`x * y`
Division	`x / y`
Remainder	`x % y`
Negation	`-x`

- An operand can be a *literal* value, a *variable*, or the *value returned by a method call*

Division with Integers

- If both operands to the division operator (/) are integers, *the result is an integer* (the fractional part is discarded)
- The remainder operator (%) returns the remainder after dividing the second operand into the first
- Example:

```
int numHours = 52;
int fullDays = numHours / 24;
int remainingHours = numHours % 24;
```

- Division by 0
  - Produces run-time error
  - Program has to avoid it, or it will crash

DivisonInt

```
import java.util.Scanner;
public class DivisionInt {
  public static void main(String[] args) {
    int numerator, denominator, output, remainder;
    Scanner keyboard = new Scanner(System.in);
    // Read the values
    System.out.println("Enter the numerator: ");
    numerator = keyboard.nextInt();
    System.out.println("Enter the denominator: ");
    denominator = keyboard.nextInt();
    output = numerator / denominator;
    remainder = numerator % denominator;
    System.out.println("The result is: " + output);
    System.out.println("The remainder is: " + remainder);
  }
}
```

Operator Precedence

- Operators can be combined into complex expressions

```
result  =  total + count / max - offset;
```

- Operators have a well-defined precedence which determines the order in which they are evaluated
- Multiplication, division, and remainder are evaluated prior to addition, subtraction, and string concatenation
- Arithmetic operators with the same precedence are evaluated from left to right
- Parentheses can always be used to force the evaluation order

Operator Precedence

- What is the order of evaluation in the following expressions?

```
a + b + c + d + e
```

1 2 3 4

```
a + b * c - d / e
```

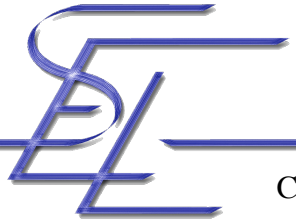
3 1 4 2

```
a / (b + c) - d % e
```

2 1 4 3

```
a / (b * (c + (d - e)))
```

4 3 2 1

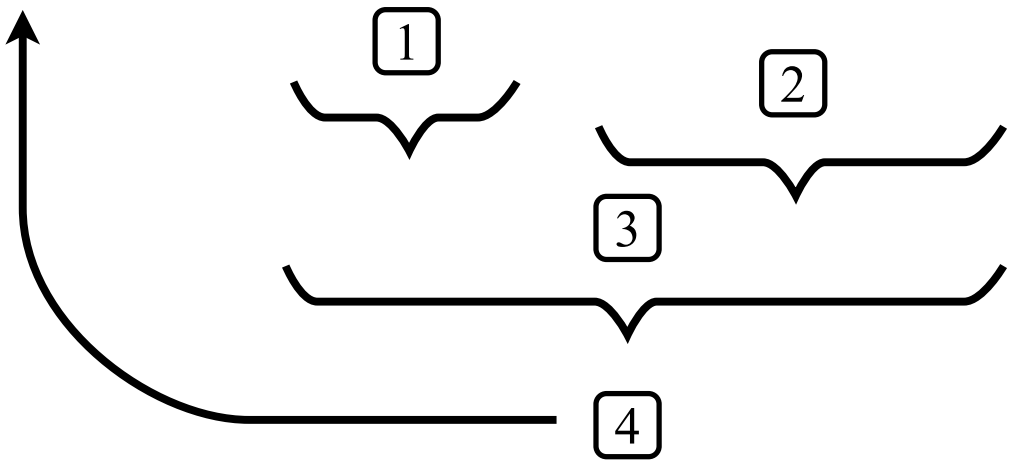


McGill

Assignment Operator Precedence

- The assignment operator has a *lower* precedence than the arithmetic operators

```
answer  =  sum / 4 + MAX * lowest;
```



First the expression on the right hand side of the = operator is evaluated

Then the result is stored in the variable on the left hand side

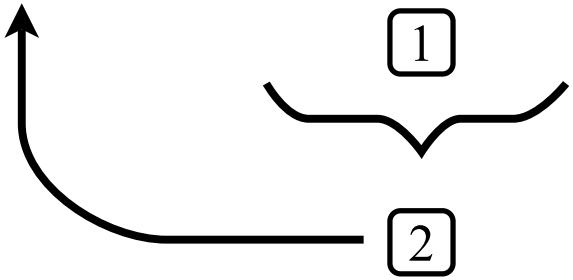
TempConvert.java

```
import java.util.Scanner;
public class TempConvert {
   public static void main (String args[]) {
     int base = 32;
     double conversionFactor = 9.0 / 5.0;
     double fahrenheitTemp;
     int celsiusTemp;
     Scanner scan = new Scanner(System.in);
     celsiusTemp = scan.nextInt();
     farenheitTemp = celsiusTemp * conversionFactor + base;
     System.out.println(Celsius Temp   = “ + celsiusTemp);
     System.out.println(Farenheit Temp = “ + FarenheitTemp);
   }
 }
```

Assignment Operator Sides

- The right and left hand sides of an assignment statement can contain the same variable

```
count  =  count + 1;
```



First, 1 is added to the original value of `count`. The result is stored in a temporary memory location.

Then the result is stored back into `count` (overwriting the original value)

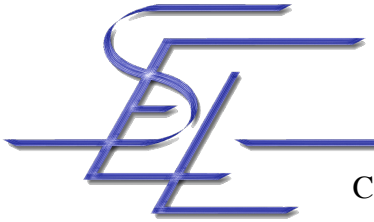
The fact that the assignment operator has lower precedence than arithmetic operators makes this possible.

Constants

- A *constant* is an identifier that is similar to a variable except that it holds *one value for its entire existence*
- The compiler will issue an error if you try to assign a value to a constant more than once in the program
- In Java, we use the `final` modifier to declare a constant

```
final double PI = 3.14;
```

- Constants have the following advantages:
  - They give names to otherwise unclear literal values
  - They facilitate changes to the code
    - More precision required? Change PI to 3.14159 in only one place
    - No need to search the whole program for occurrences of the value
  - They prevent inadvertent errors
  - Cannot type the wrong value by mistake



McGill

Circle.java

```
import java.util.Scanner;
public class Circle {
  public static void main(String[] args) {
    double radius, circumference, area;
    final double PI = 3.14;
    Scanner keyboard = new Scanner(System.in);
    // Read the radius
    System.out.println("Enter the radius: ");
    radius = keyboard.nextDouble();
    // Perform calculations
    circumference = 2.0 * radius * PI;
    area = radius * radius * PI;
    System.out.println("The circumference is: " + circumference);
    System.out.println("The area is: " + area);
  }
}
```

Boolean Expressions

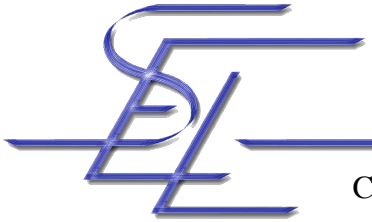
- Instead of evaluating to a numeric value, boolean expressions evaluate to either *true* or *false*

```
myNumber > 0 // can be either true or false
```

- You can assign the result of a boolean expression to a variable of type boolean:

```
boolean positive;
positive = (myNumber > 0);
```

- Boolean expressions are often used to *control the flow of execution* of a program
  - We will see control flow in detail later in the course; we will cover boolean expressions in more detail then



McGill

Increment / Decrement Operators

- The increment and decrement operators are arithmetic and operate on one operand
- The increment operator (++) adds one to its operand
- The decrement operator (--) subtracts one from its operand
- The statement

```
count++;
```

  is essentially equivalent to

```
count = count + 1;
```

- But there are some traps...

Increment / Decrement Forms

- The increment and decrement operators can be applied in *prefix* form (before the variable) or *postfix* form (after the variable)
- When used alone in a statement, the prefix and postfix forms are basically equivalent; that is, when used alone in a statement,
  `++count;`
  is equivalent to
  `count++;`
  which is equivalent to
  `count = count + 1;`
- However, they are not equivalent when they are used in expressions!