

# CSIS 235

# Object Oriented Programming

Dr. Anthony Varghese  
Dept. of Computer Science

Lecture 1 – September 5<sup>th</sup> 2012 – Java Review

# What I expect you to know from CSIS 161 and 162

**CSIS 161**: An introduction to fundamental computer concepts and structured programming techniques. The programming language Java will be used to teach the basic concepts of program analysis, design, implementation, debugging and testing. Topics include: simple data types, problem solving, program design, conditional execution, **loops**, simple **GUI** applications and **methods**.

**CSIS 162**: A continuation of fundamental computer concepts and programming. Java will be used to teach the basic concepts of program analysis, design and implementation. Topics include: methods, **exceptions**, **file IO**, **arrays** and their applications, abstract data types and **classes**.

# Instructor's notes from CSIS 162:

*“CSIS 162 worked well this semester with the new book and the new sequence of topics, here is a list of what new students in 235 should know very well:*

*(a) Reading and writing from/to text Files*

*(b) Methods and scope rules*

*(c) Arrays 1 and 2 dimensional*

*(d) Searching in arrays*

*(e) Classes and Objects fully*

*(f) Arrays of objects*

*(g) Inheritance and composition this include full understanding of the IS-A and HAS-A relationships*

*(h) Abstract classes and abstract methods (polymorphism) should have very good understanding of the concepts.*

*(i) Interfaces (students should know what they are)”*

# Let's review some of the basics

# Java's built-in primitive types

Integral types:

**byte, short, int, long.**

Real types:

**float, double.**

The operations for these types include:

addition, subtraction, multiplication, and division.

# More built-in primitive types

## **char**

values include the upper and lower case letters, digits, punctuation marks, and spaces that constitute text.

## **boolean**

Can be only one of two values: `true` and `false`.

# Java Identifiers

We use *identifiers* to name things in Java.

A Java identifier is a sequence of characters consisting of letters, digits, dollar signs(\$), and/or underscores(\_). An identifier can be of any length, and cannot begin with a digit.

x	Abc	aVeryLongIdentifier
b29	a2b	A_a_x
B\$2	\$_	\$\$\$ IXL8

Not legal identifiers:

2BRnot2B	a.b
Hello!	A-a Test.java

# Identifiers

Java identifiers are *case sensitive*. This means that upper and lower case characters are different. For example, the following are all different identifiers.

```
total  Total  TOTAL  tOtAl
```



# Identifiers

- There are a number of *keywords* and *identifier literals reserved for special purposes*.
- *Cannot be used as identifiers*.
- Identifier literals: **true**, **false**, **null**

# Guidelines in choosing identifiers

- Use lower-case characters, with upper-case characters inserted for readability.
- Capitalize class names.
- Choose descriptive identifiers
- Avoid overly long identifiers.
- Avoid abbreviations.
- Be as specific as possible.
- Take particular care to distinguish closely related entities.
- Don't incorporate the name of its syntactic category in the name of an entity

# Java Literals

A *literal* is a representation of a value.

e.g.:

5

3.1415

"Hello"

'g'

# Integer Literals

- Integer literals look like ordinary decimal numbers, and denote values of the type **int**.

25    0    1233456    289765    7

- Integer literals cannot contain commas or **decimal points** and shouldn't have leading zeros unless you know what you are doing.

What will the following program print?

```
int j = 050;  
System.out.println("  j is " + j);
```

# Floating point Literals

Numbers that include a **decimal point** denote values of type **double**.

```
0.5  2.67  0.00123  12.0  2.  .6
```

Q: What is the difference between **double** and **float**?

# Floating point literals

*Exponential notation* can also be used for **double** literals. The following are legal **double** literals:

0.5e+3      0.5e-3      0.5E3      5e4      2.0E-27

What will the following program do?

```
float a = 0.5;  
System.out.println("  a is " + a);
```

# Java puzzler

What will the following program print out?

```
double a = 1.0;  
double b = 0.9;  
double c = a - b;  
System.out.println("  a is " + a + " and b is " + b );  
System.out.println("  a - b is " + c );
```

# Java puzzler

What will the following program print out?

```
int i=Integer.MAX_VALUE;  
System.out.println(" i is      " + i );  
i = i + 1;  
System.out.println(" i is now " + i );
```



# Java Values and Types

```
int speed = 10;
```

What does “**int**” mean?  
What does any type mean?

# Java Values and Types

```
int speed = 10;
```

A “**type**” is a set of values, operations on those values

For **ints**, the set of values go from about -2 billion to about +2 billion.

For **ints** the basic operations include +, −, /, \*, and %

*See p. 23 of textbook*

# Java Values and Types

```
int speed = 10;
```

Type == a set of  
values + operations  
on those values

Value:  
What is “**10**”?  
What is a value?

# Java Values and Types

```
int speed = 10;
```

Value:

abstraction used to represent (or model)  
properties of objects

# Values and Types

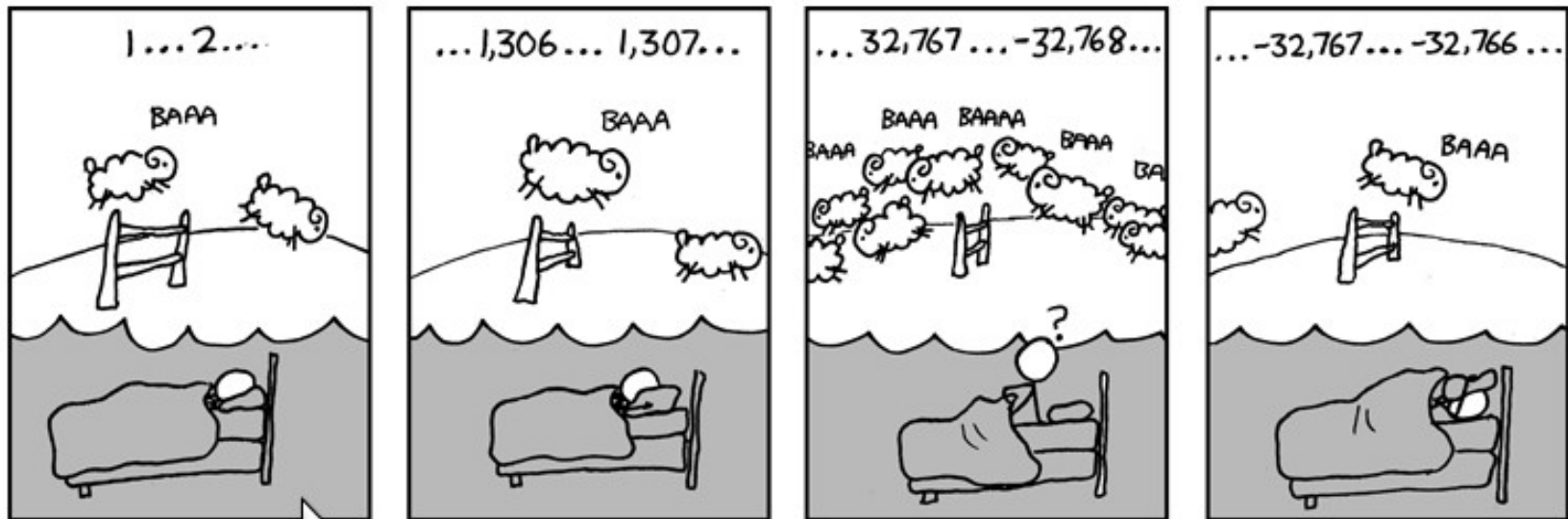
- Integers model problem features we **count**:
  - the number of students in a class
  - the number of words on a page
- Real numbers model problem features we **measure**:
  - the width of a table
  - the time it takes to run a 100 meters

# Ranges of integer type values

<i>type</i>	<i>Smallest value</i>	<i>Largest value</i>
<b>byte</b>	-128	127
<b>short</b>	-32,768	32,767
<b>int</b>	-2,147,483,648	2,147,483,647
<b>long</b>	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

# Ranges of integer type values

## CAN'T SLEEP



If androids someday DO dream of electric sheep, don't forget to declare sheepCount as a long int.

IMAGE URL (FOR HOTLINKING/EMBEDDING): [HTTP://IMGS.XKCD.COM/COMICS/CANT\\_SLEEP.PNG](http://imgs.xkcd.com/comics/cant_sleep.png)

# Values and Types

Values are grouped together according to the *operations* we perform with them.

Adding a 1 to positive int values will give you a bigger number ... **except** for the largest positive value: adding 1 to this gives you the largest negative value!

Although this does not model how we count, this is how integer addition works in most programming languages!



# Floating point literals – rounding errors

What will the following program print out?

```
float x = 300000.0f;  
float y = x + 0.02f;  
System.out.println(" x is " + x );  
System.out.println(" y is " + y );
```

# Floating point literals – rounding errors

What will the following program print out?

```
float x = 300000.0f;  
float y = x + 0.02f;  
System.out.println("  x is " + x );  
System.out.println("  y is " + y );
```

x is 300000.0

y is 300000.03

Reason: fixed precision of floats – about 8 digits.

# Floating point literals – rounding errors

What will the following program print out?

```
double a = 1.0;  
double b = 0.9;  
double c = a - b;  
System.out.println("  a is " + a + " and b is " + b );  
System.out.println("  a - b is " + c );
```

# Floating point literals – rounding errors

What will the following program print out?

```
double a = 1.0;  
double b = 0.9;  
double c = a - b;  
System.out.println("  a is " + a + " and b is " + b );  
System.out.println("  a - b is " + c );
```

a is 1.0 and b is 0.9

a - b is 0.09999999999999999

Reason: not easy to do exact arithmetic, even with **double** precision

# String literals

String literal is a possibly empty sequence of characters enclosed in quotations:

```
"ABC "
```

```
"123 "
```

```
"A "
```

```
" "
```

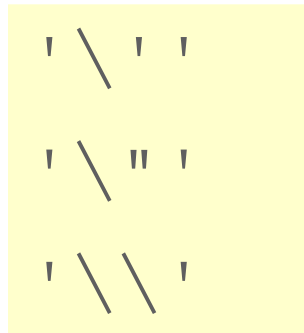
# Character literals

Character literals (denoting values of type **char**) consist of a single character between apostrophes.

```
'A'  'a'  '2'  ';'  '.'  ' '  ''
```

# Character literals

The apostrophe, quotation mark, and backslash must be preceded by a backslash in a character literal.



```
' \ ' '  
' \ " '  
' \ \ ' '
```

# Character literals and operations

What will the following program print?

```
System.out.println( "A" + "t" );  
System.out.println( 'A' + 't' );
```



# Character literals and operations

What will the following program print?

```
System.out.println( "A" + "t" );  
System.out.println( 'A' + 't' );
```

At

181

Reason: one is string concatenation, the other is addition.

# Character Literals

`'\t'` //represents the tab character

`'\n'` //represents the end of line character.

# Boolean Literals

The two values of type **boolean** are:

`true`

`false`

# Types and variables

A variable can contain values of only one type.

- An **int** variable contains a single **int** value;
- a **double** variable contains a **double** value, etc.

What will the following program do?

```
int i = 1;  
float a = i;  
double b = a;  
System.out.println("  b is " + b);
```

# Types and variables

The type of **value** a variable contains is

- “*type of the variable*” and
- fixed when the object is designed.

Example: The width of a window might be 100 pixels at one time, and 150 pixels some time later.

Value will always be an integer, and instance variable modeling it is of type **int**

# Operators

Variables can be combined with *operators* to form expressions.

`i1 / -3       $\Rightarrow$       -3`

`-7 / 2       $\Rightarrow$       -3`

`i1 % -3       $\Rightarrow$       1`

`-7 % 2       $\Rightarrow$       -1`

# Arithmetic expressions

- Expressions that evaluate to integer and floating point values.
- Can be built by combining literals and variable names with *arithmetic operators*

+	addition
-	subtraction
*	multiplication
/	division
%	remainder

# *Unary operators: “+” and “-”*

assume that **i1** is an **int** variable containing 10

$$+3 \Rightarrow 3$$

$$-3 \Rightarrow -3$$

$$+i1 \Rightarrow 10$$

$$-i1 \Rightarrow -10$$



# Division operator

“/” denotes *division* when applied to two floating point operands, but *integer quotient* when applied to two integer operands.

$$2.0/4.0 \Rightarrow 0.5$$

$$2/4 \Rightarrow 0$$

$$5.0/4.0 \Rightarrow 1.25$$

$$5/4 \Rightarrow 1$$

# Remainder operator %

Primarily used with integer operands

10	%	5	⇒	0
10	%	3	⇒	1
10	%	6	⇒	4
10	%	11	⇒	10

# Arithmetic expressions: assigning literals

## Literals:

0      7      23      0.5      2.0      3.14159      2.4e-23

Variable names denote values currently stored.

```
int    i1 = 10;    int    i2 = -20;    int i3 = 30;  
double d1 = 2.5;    double d2 = 0.5;
```

//evaluating these produces:

i1	⇒10	i2	⇒-20	i3	⇒30
d1	⇒2.5	d2	⇒0.5		

# Mixed types: Numeric promotion

Mixed Operands: what happens if one operand in an expression is **int** and the other **double**?

The **int** operand is converted (**promoted**) to a **double** representing same value

```
7 / 2.0  
int i1 = 10;  
i1 * 0.5
```

# Mixed types: Numeric promotion

What will the following program print?

```
int i1 = 7;  
System.out.println(" i1 / 2 is " + (i1 / 2));  
System.out.println(" i1 * 0.5 is " + (i1 * 0.5) );  
int i2 = i1 * 0.5;  
System.out.println(" i2 is " + i2 );
```

# Mixed types: Numeric promotion

What will the following program print?

```
int i1 = 7;  
System.out.println(" i1 / 2 is " + (i1 / 2));  
System.out.println(" i1 * 0.5 is " + (i1 * 0.5) );  
int i2 = i1 * 0.5;  
System.out.println(" i2 is " + i2 );
```

The second last line does not compile. Why?

i1 / 2 is 3

i1 \* 0.5 is 3.5

# Numeric promotion

The value of an operand of a binary or a unary operator is automatically converted to a similar value of a different type when necessary:

`7 / 2.0`  $\Rightarrow$  `7.0 / 2.0`

$\Rightarrow$  `3.5`

`i1 * 0.5`  $\Rightarrow$  `10 * 0.5`

$\Rightarrow$  `10.0 * 0.5`

$\Rightarrow$  `5.0`

# Expressions

There are two ways to interpret:

5 - 3 \* 2

What are the two possible values that result from interpreting the above expression?

Which one is “correct”?



# Operator precedence

What is the order of evaluation in

5 - 3 \* 2

- Unary + and – have *higher precedence* than binary operators.
- \*, /, % have *higher precedence* than operators +,-

# Operator precedence

If two operators have equal precedence, operations are performed *left to right*. i.e.

$$10 / 5 * 3 = 6$$

□ Parentheses can be used to override precedence.  
i.e.

$$10 / ( 5 * 3 )$$

# Arithmetic expressions: **Precedence**

- Operators  $*$ ,  $/$ , and  $\%$  have *higher precedence* than binary operators  $+$  and  $-$ .
- In an un-parenthesized expression multiplication is done before addition

**Multiply before adding**

$i1 + 10 * 2 \Rightarrow i1 + 20 \Rightarrow 30$

$i1 * 10 + 2 \Rightarrow 100 + 2 \Rightarrow 102$

$10 / 2 + 1 \Rightarrow 5 + 1 \Rightarrow 6$

$5 + 6 / 10 \Rightarrow 5 + 0 \Rightarrow 5$

$- 5 + i1 \Rightarrow (-5) + 10 \Rightarrow 5$

# Expressions

There are two ways to interpret:

5 - 3 - 2

What are the two possible values that result from interpreting the above expression?

Which one is “correct”?

# Arithmetic expressions **Associativity**

Binary operators are *left associative*: when expression contains two operators with equal precedence, operations are performed left to right.

Left operator before right

$i1 / 5 * 2 \Rightarrow 2 * 2 \Rightarrow 4$

$10 - 4 - 3 \Rightarrow 6 - 3 \Rightarrow 3$

$i1/20 * 2 \Rightarrow 10/20 * 2 \Rightarrow 0 * 2 \Rightarrow 0$

$2 * i1/20 \Rightarrow 2 * 10/20 \Rightarrow 20/20 \Rightarrow 1$

$20 / i1 * 2 \Rightarrow 20 / 10 * 2 \Rightarrow 2 * 2 \Rightarrow 4$

# Casting

- Occasionally must convert a value to a different type to perform certain operations.
- Syntax: **(type) expression**

```
10/40 = 0
```

```
(double)10/(double)40 = 0.25
```

```
10.0/40.0 = 0.25
```

```
(int)10.0/(int)40.0 = 0
```

Cast operators have *higher precedence* than arithmetic operators.

# Casting

*(type)expression*

`(double)i1`  $\Rightarrow$  `(double)10`  $\Rightarrow$  10.0

`(double)i1 / i3`  $\Rightarrow$  10.0 / 30  $\Rightarrow$  0.333...

`(int)d3`  $\Rightarrow$  2

`(int)d4`  $\Rightarrow$  -2

`(double)i1/i3`  $\Rightarrow$  10.0/30  $\Rightarrow$  0.333...

`(double)(i1/i3)`  $\Rightarrow$  `(double)0`  $\Rightarrow$  0.0

# String Concatenation

For *String* operands: the binary operator “+” denotes string *concatenation*.

```
string1 + string2
```

Evaluates to a *String* containing characters of *string1* with characters of *string2* appended.

```
"abc" + "def" ⇒ "abcdef"
```



# String Concatenation

If one operands of “+” is a *String* and the other isn’t, the non-*String* operand will be converted to a *String*, and concatenation performed.

```
int i = 23;
```

```
"abc" + i           ⇒      "abc23"
```

```
i + " "            ⇒      "23 "
```

```
"2*i=" + 2*i        ⇒      "2*i=46"
```

```
2+i+"!"            ⇒      "25!"
```

```
"!" + 2+i           ⇒      "!223"
```

( “+” is left associative!)

# Classes

Why use classes?

What does using classes allow us to do?

# Classes

Why use classes?

What does using classes allow us to do?

Classes allow us to set up our own “\_\_\_\_\_”s

Allows \_\_\_\_\_

Allows \_\_\_\_\_ via \_\_\_\_\_, \_\_\_\_\_

# Classes

Why use classes?

What does using classes allow us to do?

Classes allow us to set up our own “t\_\_\_\_\_”s

Allows enc\_\_\_\_\_

Allows code re\_\_\_\_\_ via inh\_\_\_\_\_, po\_\_\_\_\_

# Classes

Why use classes?

What does using classes allow us to do?

Classes allow us to set up our own “**type**”s

Allows **encapsulation**

Allows **code reuse** via **inheritance**, **polymorphism**

# Classes

- Class: collection of similar objects, that is, objects supporting the same queries and commands.
- Every object is an **instance** of some class, which determines the object's features.

# Example: A class modeling a counter

```
/**                                     1
 * A simple integer counter.
 */
public class Counter {

    private int count;

    /**
     * Create a new Counter, with
     * the count initialized to 0
     */
    public Counter () {
        count = 0;
    }

    /**
     * The number of items counted
     */
    public int currentCount () {
        return count;
    }
}
```

```
/**                                     2
 * Increment the count by 1.
 */
public void incrementCount
    () {
        count = count + 1;
    }

    /**
     * Reset the count to 0.
     */
    public void reset () {
        count = 0;
    }

} // end of class Counter
```

# Reference values: Objects as properties of objects

When we write classes, we want to

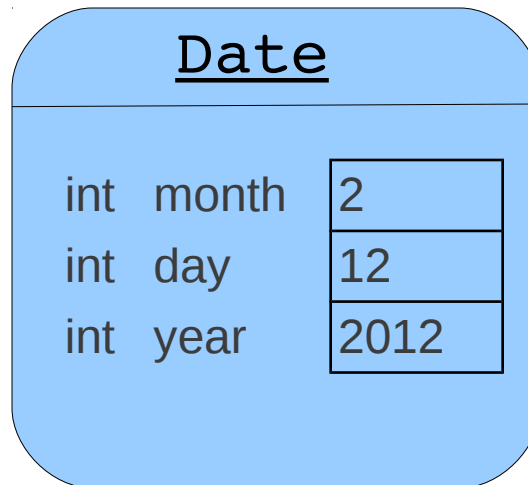
- model an object's properties such as number values – primitive data types.
- model properties like the name, address, and course schedule or birthday of a student – reference data types



# Reference values:

## Objects as properties of objects

Example: to model birthday, we first model a date with an object, via the class Date:

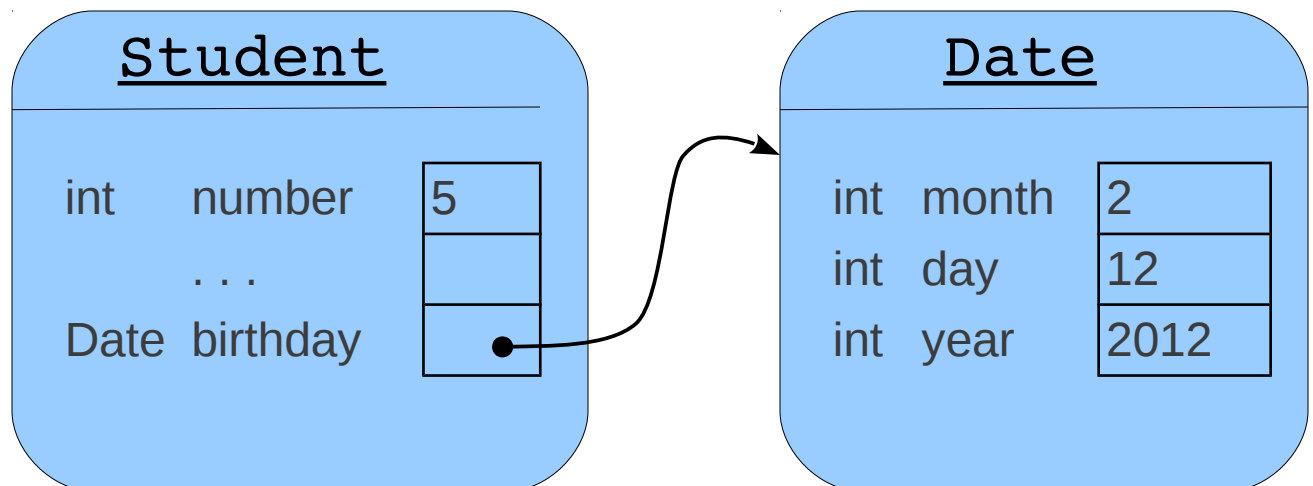


# Reference values

The student's value for property “birthday” *denotes* or *refers* to a *Date* object.

- ❑ Value is a *reference* value.
- ❑ Type is *reference-to-Date*

■ **reference value:** a value that denotes an object.



# Reference values

Remember that reference data types are references to objects.

Quick quiz: if we have

```
String x = "Hello";
```

```
String y = x;
```

```
y = null;
```

What is the value of x?

# Overview of a complete system

Example:

We build a very simple system to test a *Counter*.

- ❑ The model : only a single class Counter.
- ❑ The user interface: only a single class CounterTester.
- ❑ Data management: none.

```
//A simple tester for the class Counter.  
public class CounterTester {  
    private Counter counter;  
  
    // Create a new CounterTester.  
    public CounterTester () {  
        counter = new Counter();  
    }  
    // Run the test.  
    public void start () {  
        System.out.println("Starting count:");  
        System.out.println(counter.currentCount());  
        counter.incrementCount();  
        counter.incrementCount();  
        counter.incrementCount();  
        System.out.println("After 3 increments:");  
        System.out.println(counter.currentCount());  
        counter.reset();  
        System.out.println("After reset:");  
        System.out.println(counter.currentCount());  
    }  
}
```

# Getting it all started

We need one more class containing a method named `main`, as shown

```
/**
 * Test the class Counter.
 */
public class Test {

    /**
     * Run a Counter test.
     */
    public static void main (String[] args) {
        CounterTester tester = new CounterTester();
        tester.start();
    }
}
```

# Running a program

Depends on the computing system and environment used. We must identify class containing method `main` to the Java run-time system. For instance,

```
$ java Test
```

Running the program will produce six lines of output:

```
Starting count:  
0  
After 3 increments:  
3  
After reset:  
0
```

# Objects that “wrap” primitive values

Primitive values and objects are different.

For each primitive type Java provides classes **Boolean, Character, Byte, Short, Integer, Long, Float, Double.**

- Immutable classes
- Wrap a primitive value into an object.

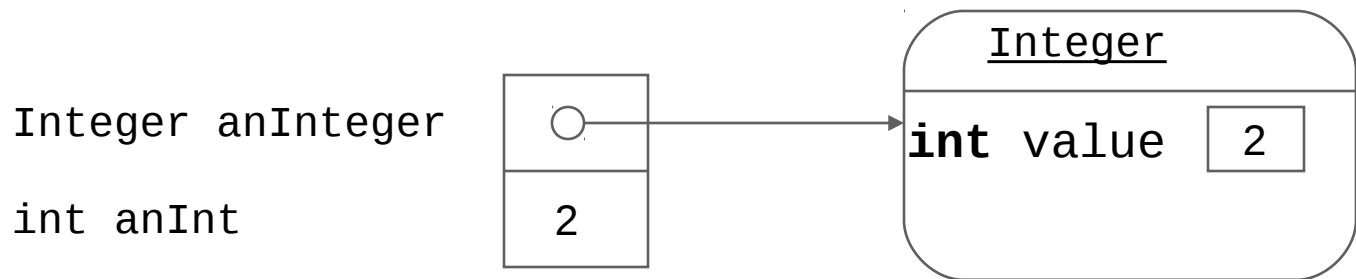


# Objects that “wrap” primitive values

Example: Integer class.

- ❑ An instance is an immutable object.
- ❑ It has a single **int** property.
- ❑ Value of property given during creation.
- ❑ Property is accessed via query: **intValue()**

```
Integer anInteger = new Integer(2);  
int anInt = anInteger.intValue();
```



# Boxing and unboxing

Automatic and implicit conversion between primitive values and the wrapper classes.

```
Integer anInteger = 3;
```

Implicitly converted to:

```
Integer anInteger = new Integer(3);
```

# Boxing and unboxing

Also,

```
int anInt = anInteger + 1;
```

Implicitly converted to:

```
int anInt = anInteger.intValue() + 1;
```

# End of Java Basics Part 1

Next lecture: Basic Object oriented Design –  
abstraction, composition

Before next class: try out above code in Eclipse.