

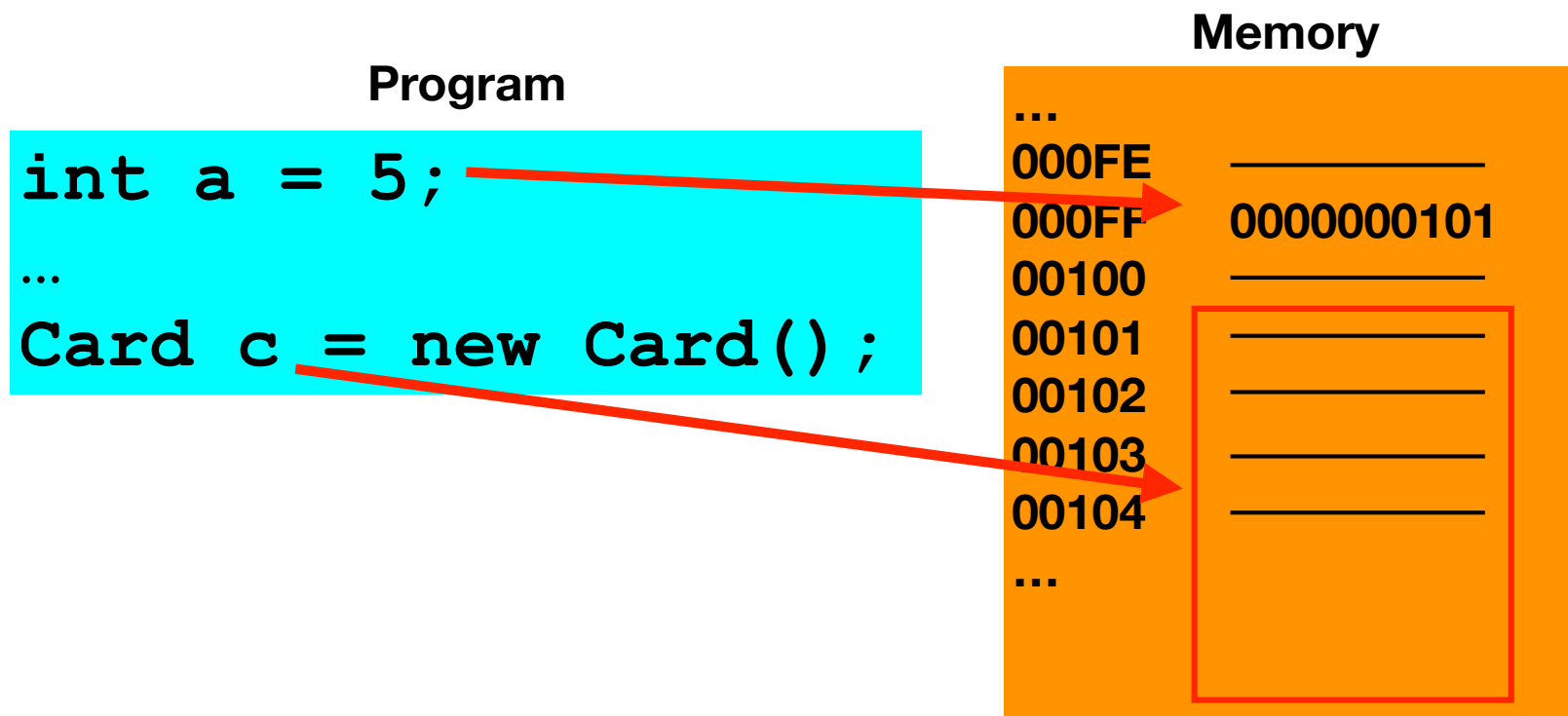


# Data Types, Variables, and Arithmetic

David Greenstein  
Monta Vista High School

# Variables

- A variable is a program's "named container" that holds a value in real memory.
- More complex values, like objects, take up a range of memory.



# Variables

In OOP, there are two different types of variables:

- **Local variables**
- **Field variables**

# Local Variables

- A **local variable** is “bound” to a block of code.
- In Java, a block of code is surrounded by braces `{}` .

Variables  
local to  
block `{}`

```
{  
    int x = 3;  
    String str = "11";  
    int z = x - 2;  
}
```

# Field Variables

- A **field variable** is also called an **instance variable**.
- In Java, a class file (source code) define an object and its fields.

Fields of  
an object  
of data type  
"MyClass"



Java Class File  
(source code)

```
public class MyClass {  
    private int k;  
    private String str;  
    ...  
}
```

# Constants

## Literal Constants

Java code

```
char: '0', 'A', '\n'  
int: 3, 987, -55  
double: 0.34, 3.4e-20  
float: 4.5f  
String: "Hello", "1776"
```

String is special in Java.  
It is the only class object  
that has a literal equivalent.

---

## Symbolic Constants

Java code

```
private final int SIDE_LENGTH = 8;  
private static final int BUFFER_SIZE = 1024;  
public static final int PIXELS_PER_INCH = 6;
```

Symbolic constants  
are initialized as **final**.

# Why Symbolic Constants?

- Easy to change in one place and the change permeates throughout the program.
- Easy to assign as a literal constant, but more meaningful.
- More readable, self-documenting code.
- Removes “magic” numbers.
- Additional data type checking by the compiler.

```
private final int SIDE_LENGTH = 8;  
private static final int BUFFER_SIZE = 1024;  
public static final int PIXELS_PER_INCH = 6;
```

# Variable Data Types

In OOP, every variable has a data type.  
There are two groups of data types:

- **Primitive data types**

In the Java language, there are 8 types:

byte, short, int, long, float, double, char, boolean

**APCS A Exam only uses int, double, char, boolean**

- **Object data types**

In the Java language, an object type is called a **class definition** and is defined by the programmer or a library.



# Variable Scope

- The scope of a variable is the portions of source code that the variable is “visible”.
- In block-structured code, like Java, the scope of a variable is determined by the braces surrounding it.
- Variables can be “seen” by inner blocks, but cannot be “seen” by outer blocks.

Declared variables can be “seen” by outer and inner blocks }

```
{
  int x = 3;
  int y = 11;
  int z = x - y;
  {
    double s = 3.44;
    float t = y;
  }
  s = z;
}
```

This will cause a compiler error

Declared variables can be “seen” only by inner block }

# Variable Scope (cont.)

- In Java, **control statements** have block structure and follow scope rules.

```
if (isValid()) {  
    char a = '!';  
    int b = 303;  
    String ss = null;  
}  
b += b;
```

This will cause  
a compiler  
error

Declared  
variables  
can be “seen”  
only inside if  
block }

# Variable Scope (cont.)

- In Java, the **for-loop variable** follows scope rules.

This will cause  
a compiler  
error

```
for (int t = 0; t < 6; t++){  
    some code here ...  
}  
int s = t;
```

variable "t"  
can only be  
"seen" inside  
for-loop

If "t" is  
declared outside  
for-loop,  
no compiler  
error here

```
int t;  
for (t = 0; t < 6; t++){  
    some code here ...  
}  
int s = t;
```


# Variable Scope (cont.)

## Variable names

- can be the same if they are declared in different blocks
- each declaration creates a new storage location that is “seen” inside only that block

```
{  
  String s = ...  
}  
{  
  char s = ...  
}  
for (int s = 0; ...) {  
  
}
```

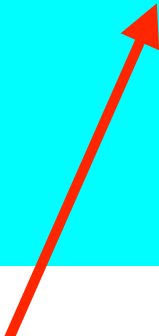
variable name “s”  
can be declared in  
different blocks  
using different types



# Variable Scope (cont.)

**Function (method) parameters** are local to that function.

```
public void addOne(int a, char b, double c){  
    some code here ...  
}
```



Variables “a”, “b”, and “c”  
are local to `addOne` method.  
These variables cannot  
be “seen” by other methods.

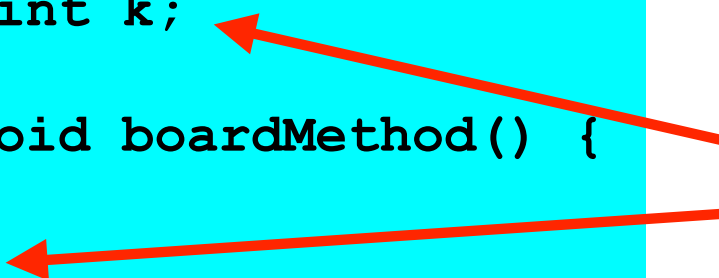
# Variable Scope (cont.)

## Class Fields

- are visible to all the functions (methods) of the object

```
public class Board {  
    private int k;  
    ...  
    public void boardMethod() {  
        ...  
        k = 5;  
        ...  
    }  
}
```

Both refer to the same storage space in memory.



# Variable Scope (cont.)

## Class Fields

- are visible to all the functions (methods) of the object

```
public class Board {  
    private int k;  
    ...  
    public void boardMethod() {  
        ...  
        k = 5;  
        ...  
    }  
}
```

Both refer to same storage space in memory.

```
} public class Board {  
    private int k;  
    ...  
    public void boardMethod() {  
        ...  
        int k = 5;  
        ...  
    }  
}
```

**WARNING!!!**  
Same name allowed but 2nd declaration creates a 2nd storage space.

# Variable Declarations

- Variables must be declared before they are used.
- A declaration statement can include initialization.
- By default, Java initializes numbers to **0**, booleans to **false**, and objects to **null**. Other languages, like C++, do not initialize the variable for you and you must initialize it yourself.

```
char c;  
c = 'X';  
int a = 5, b = 6;  
JFrame frame = new JFrame();  
  
str = "Hello World";  
String str;
```

Declared before used

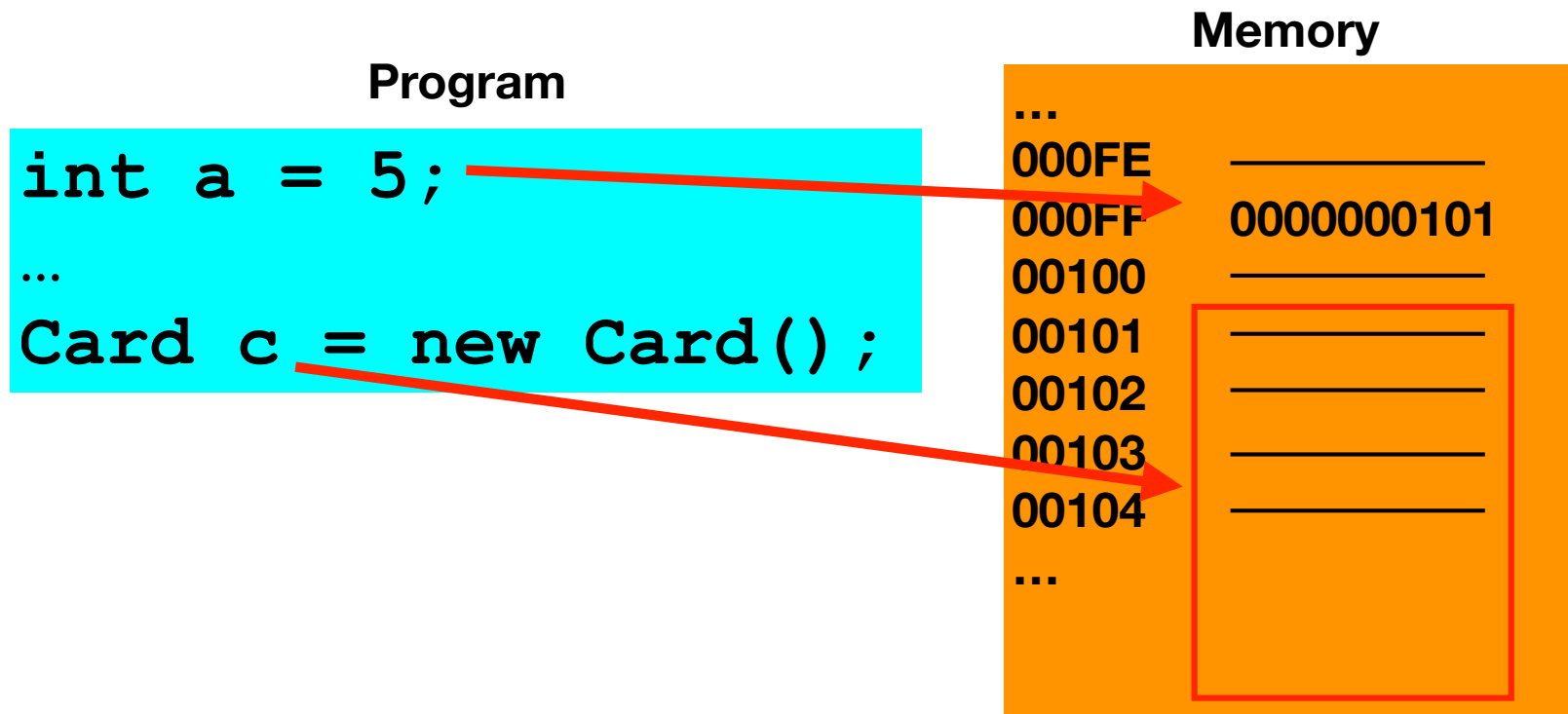
Declared and initialize

Used before  
declared.  
Compiler error!



# Assignment Operator “=”

- The assignment operator (=) performs a function.
- It takes the value computed by the expression on the right and stores it into the named container on the left. (evaluated **right-to-left**)



# Assignment Operator (cont)

- The assignment operator (=) can be used several times in the same statement.
- The **assignment** is always evaluated **right-to-left**.

```
int x, y, z;  
x = y = z = 5;  
z = -(3 * x % y) / 5 - 21;
```



Right side is evaluated  
\*before\* assignment is made.

# Promoting a Data Type

- In Java, primitive numeric values on the right side of the assignment statement must fit into the data type of the variable (container) on the left side.
- If the data type on the left side of the assignment is a larger container than the type on the right, then this is called “**promoting**” a value.

```
int a = 5000; // works!  
byte b = 200; // does not work;  
           // byte range -128 to 127  
short c = 32768; // does not work  
            // short range -32768 to 32767  
  
byte s = 'x'; // works! Literal treated differently
```

# Assigning Data Types

- Most languages (like Java) allow some mixed data type “promotions” when the data types are related in some way. These must follow syntactic rules of the language.
- A data type “promotion” happens when the left side of the assignment has a larger container than the right side.
- If the left side has a smaller container than the right, the compiler complains “*possible loss of precision*”.
- Exception: Assigning literals →

```
byte a = 'a';  
short s = (int)5;
```

Valid assignments (right to left):

```
double <- float <- long <- int <- char  
double <- float <- long <- int <- short <- byte
```

# Arithmetic

- Java binary operators:  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$
- The precedence of operators and parentheses is the same as in algebra.
- $m \% n$  means  
the remainder when  $m$  is divided by  $n$   
(for example,  $17 \% 5$  is  $2$  ;  $2 \% 8$  is  $2$  )
- $\%$  has the same rank as  $/$  and  $*$
- Same-rank binary operators are performed in order from left to right
- Unary operator “ $-$ ” has higher priority than  $*$  /  $\%$

$3 * - 5 \rightarrow 3 * (-5) \rightarrow -15$

# Arithmetic

What is the order of operations?

$$3 + 2 \% (4 / 5 + 9) * 7 + 1$$

**Answer?**

# Arithmetic (cont.)

- The type of the result is determined by the types of the operands, not their values. This also applies to intermediate results in expressions.

4 + 2 → results in an **int**

4.e-2 \* .0001 → results in a **double**

3 / 6.1 → results in a **double**

'a' + 4 → results in an **int**

"a" + 4 → results in a **String**

(the last "+" is interpreted as String concatenate)

# Compound Assignment Operators

## More operators

- Compound assignment operators can be used for simple arithmetic.

```
a += b; ↔ a = a + b;  
a -= b; ↔ a = a - b;  
a *= b; ↔ a = a * b;  
a /= b; ↔ a = a / b;  
a %= b; ↔ a = a % b;
```

- Unary **increment** and **decrement** operators

```
a++; ↔ a = a + 1;  
a--; ↔ a = a - 1;
```

AP Exam: Do not use these  
inside larger expressions!

→ n = arr[a++];



# Compound Assignment Operators (cont)

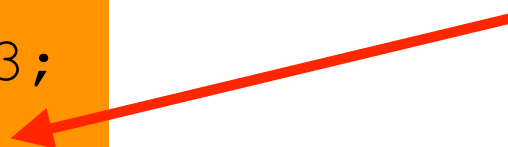
- Compound assignment operators **implicitly cast** their result before assigning.
  - The **cast** is the **data type** of the operand on the **left side** of the assignment.



Compared to binary operator “+”

```
int a = 2;  
double b = 3;  
a = a + b;
```

Loss of Precision  
Compiler Error!



# Integer Arithmetic

- In most languages (Java included) integer arithmetic **truncates** the decimal values.

```
17 / 5 gives 3  
4 / 9 gives 0
```

- The assignment statement always computes the **right side first**.

```
double x = 3 / 5;
```

Integer division produces a 0 value.  
The `double` data type on the left  
does not change the result.

# Integer Arithmetic (cont)

- **Caution!** There is no integer overflow detected by the Java compiler or interpreter.
- It is your job to make sure the size of the integer value does not exceed the storage capacity.

```
byte 10^2 = 100  
byte 10^3 = -24  
byte 10^4 = 16  
byte 10^5 = -96  
byte 10^6 = 64
```

```
int 10^9 = 1000000000  
int 10^10 = 1410065408  
int 10^11 = 1215752192  
int 10^12 = -727379968  
int 10^13 = 1316134912
```

**Overflow Numbers  
(garbage)**

# Casting

- You can force a literal or variable to another compatible data type using **casting**.

```
int a = 3.4;
```

← ERROR! Loss of precision

```
int a = (int) 3.4;
```

← Literal double 3.4 is converted to int 3 before assignment. ("Down casting")

- Casting must follow **syntactic rules** of compatibility of the language.

```
int a = (int) "Hello";
```

← ERROR! Inconvertible type

```
String s = (String) 5;
```

# Casting (cont)

- Casting can be useful when used correctly.

```
int rand = (int) (Math.random() * 10) + 1;
```

Random integer  
from 1 to 10

Random double  
from 0.0 to 9.9

- Example of casting improperly. (forgot parentheses)

```
int rand = (int) Math.random() * 10 + 1;
```

Always equals 1

Always equals 0

# Casting (cont)

- Cast operators have higher precedence than binary arithmetic operators (+, -, \*, /, %)

```
int t = (int) 9.12 * 5;
```

Cast happens  
before multiplication

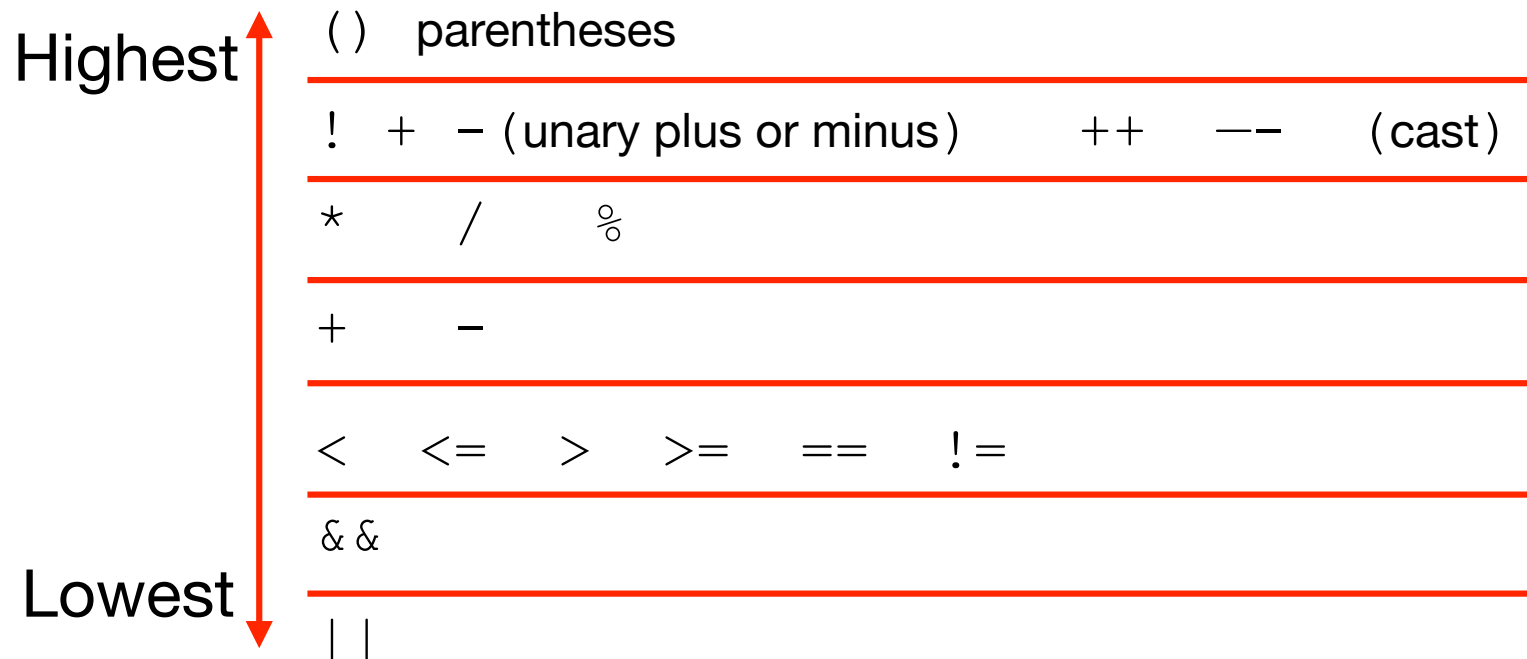
- A cast operator is a unary operator.

```
long big = (int) (11 / 3.1) + - 2500000;
```

Cast unary  
operator

Minus unary  
operator

# Operator Precedence



## Example:

```
double a = (((double)b) + (((count * 9) / 23) + gif));  
double a = (double)b + count * 9 / 23 + gif;
```

Easier to read



**Questions?**