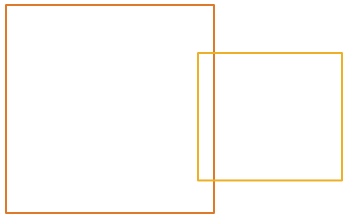
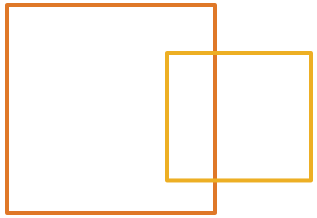
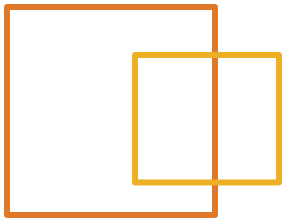
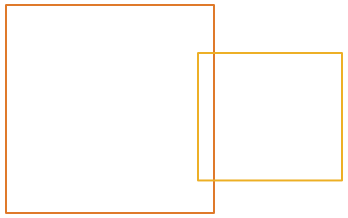
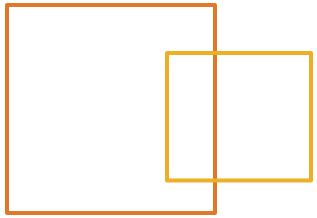
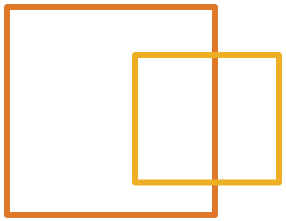





Fast Track to Java

Customized for Starbucks

Delivered by DevelopIntelligence


Variables, Operators and Data

Objectives

At the end of this module you should be able to:

- The rules for creating legal variable names in Java
- Describe and use the basic primitive data types in Java
- Use ***`String`*** data
- Determining the data type of a literal

Strong Typing in Java

- Java is a strongly typed language
  - Each variable and each expression has a type
  - Can be identified by the compiler at compile time
  - A variable's type cannot be changed
- In loosely typed languages, like JavaScript & VB

```
// JAVASCRIPT:  This is not allowed in JAVA!!!
// Declare a variable "x" with no type
var x
x = "Hi there"    // x is holding string data
x = 1234          // x is now holding numeric data
y = x + "343"     // String or numeric operation??
```

- Strong typing helps prevent errors

Data Types in Java

- There are two types
  - Reference data
  - Primitive data

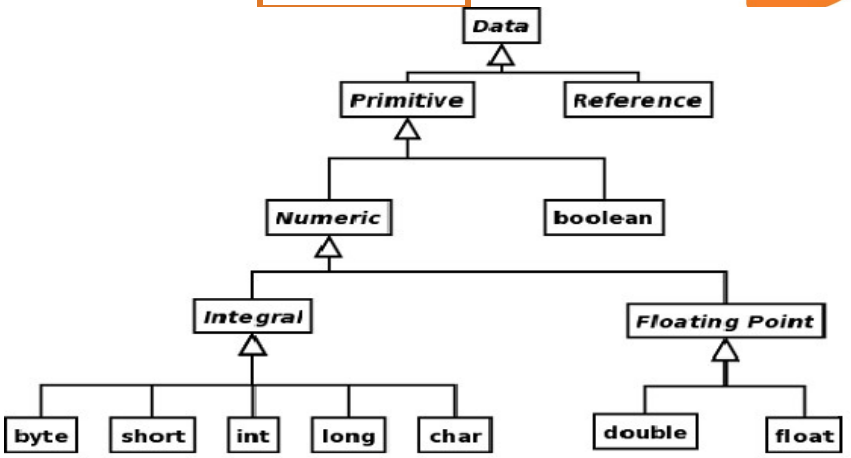
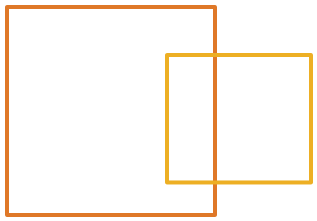
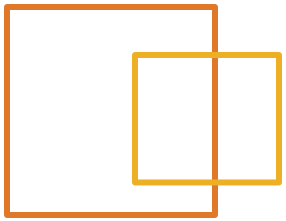


Fig. 3-1: **Hierarchy of data types in Java**

- The primitive data types resemble the types in C / C++
- Data types in Java are defined by the language specification
  - They are platform independent
  - For example, the data type ***`int`*** is *always* four bytes long

Identifiers





- Identifiers are used to name
  - Classes
  - Variables
  - Method
- Identifier rules are platform independent
  - Arbitrarily long sequence of letters and digits
  - Case sensitive
  - The first character must be a letter
    - Any valid letter in the Unicode character set
    - Underscore "_" and dollar sign "***$***" are also permitted
  - Must **not** contain any white space
  - Must **not** be the same as reserved Java keywords

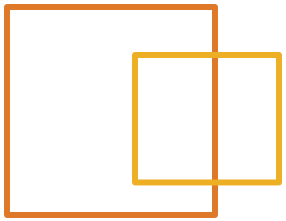
Reserved Keywords

abstract	continue	for	new	switch
assert	default	goto*	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const*	float	native	super	while
true	false	null		

*goto and const are reserved but not used

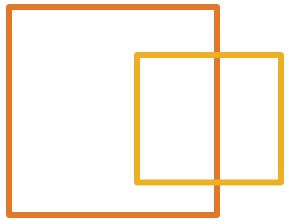
true, false, and null are but are reserved literal names

Variable Names



```
Account_Balance     // valid - remember that _ is allowed
$34             // valid - remember that $ is allowed
this                // invalid - same as reserved keyword
π               // valid - Greek letter pi is a Unicode letter
This                    // valid - different in case from 'this'
Next.item           // invalid - symbol "." not allowed
23skidoo            // invalid - must start with letter
```

Declaring a Variable

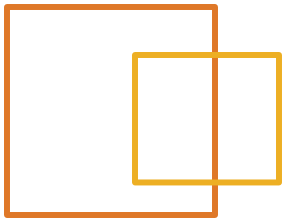


- Variables are declared in Java with the syntax

```
type name [= initializing_expression];
```

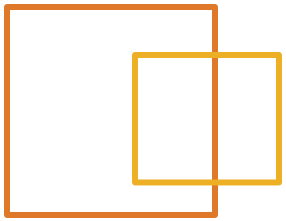
- It is good programming practice to *initialize variables when they are declared*
- A variable can also be initialized after it is declared
- Java will prevent the use of uninitialized local variables

Declaring a Variable



```
String best = "Best"; // Preferred - initialized at declaration
String okToo;           // Declared - not initialized
int x;                // Declared - not initialized

okToo = "value";     // Now okToo is initialized.
x = x + 1;       // ERROR! Use of an uninitialized variable
```

Declaring a Variable



```
boolean a,b;            // a and b are both of type boolean
boolean c = true, d = false; // initialization for both c and d
boolean e,f = true;           // WARNING! Only f is initialized!
```

```
int var1;
boolean var2 = true;
var1 = 9
```

Positioning Variable Declarations

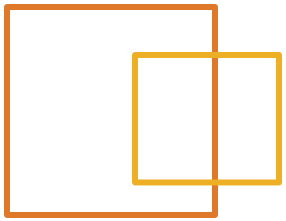
- The basic principal in Java, and in OOP in general, is to declare a variable at its point of first usage
  - This allows it to be initialized in its declaration

```
class Test {
  public static void main(String [] args) {
    int sum = 0;
    for (int counter = 0; counter < 10; counter++)
      sum = sum + counter;
      String message = "The sum is " + sum;
      System.out.println(message);
    }
  }
}
```

Boolean Data Types

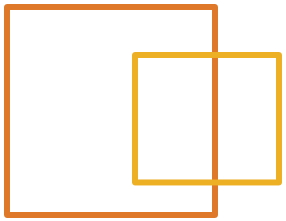
- **`boolean`** data is either **`true`** or **`false`**
- In Java, numeric and other variables are not boolean, and cannot be interpreted as such
  - (In C, C++, JavaScript and others, zero is false, any defined non-zero value is true)
- A boolean can only have the value **`true`** or **`false`**

Boolean Data Types



```
// THIS DOESN'T WORK IN JAVA, but in C++ you do can this:
int x = 43;
// non-zero x is taken as a true
if (x) {
  printf("x is %d", x);
}

// In Java we must have a boolean variable or expression.
int x = 43;
boolean test = (x == 43);
// boolean variable is OK
if (test) {
  System.out.println("x is "+x);
}
// OK because result of the == test is boolean
if (x == 43) {
  System.out.println("x is " + x);
}
```

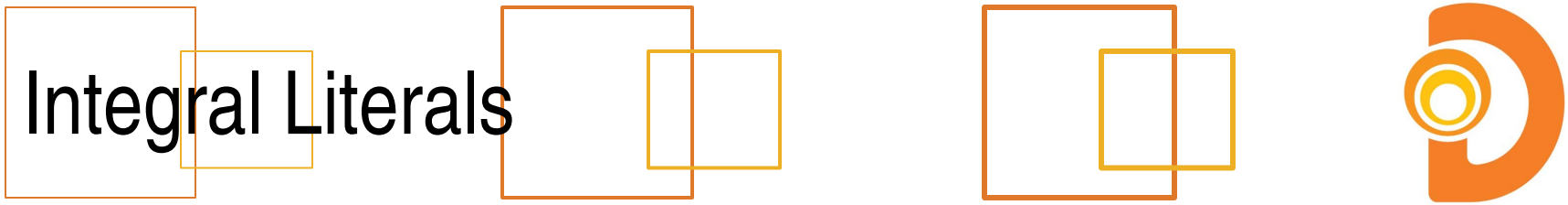
Integral Numeric Data Types



- Integral values are signed
  - char type has numeric properties, but is unsigned
- Integral values do not contain a decimal point

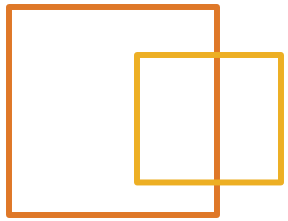
Type	Bytes	Minimum Value	Maximum Value
`byte`	1	-128	127
`short`	2	-32768	32767
`int`	4	-2147483648	214748364
`long`	8	-9223372036854775808	9223372036854775807

Integral Literals



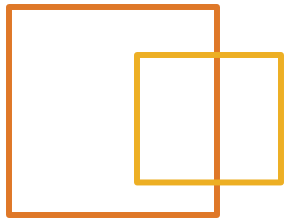
- A sequence of digits without a decimal point is assumed to be integral data
  - literals are of type ***int*** unless there is an ***L*** - upper or lowercase - immediately following the digits
  - In this case, the literal is taken to be a ***long***
  - ***7836*** and ***-98*** are ***int***
  - ***881L*** and ***-91121*** are ***long***
- Literals are interpreted in Base 10 unless
  - The literal starts with a ***0***, it is interpreted as Base 8
  - The literal starts with a ***0x*** or ***0X***, it is interpreted as Base 16

Integral Literals



```
63     // an int in base 10
-63    // a negative int in base 10
63L    // a long in base 10
063    // an int in base 8 (equal to 51 in base 10)
0631   // a long in base 8
-0631  // a negative long in base 8

091    // illegal!  Cannot have the digit 9 in base 8!

0x33   // an int in base 16 (equivalent to 51 in base 10)
0X33L  // a long in base 16
0xFF   // an int in base 16
0xff   // same as the previous line - case does not matter.
0xg1   // illegal! Can only have a-f as base 16 digits.
-0xFF  // a negative int in base 16
```

Floating Point Data Types

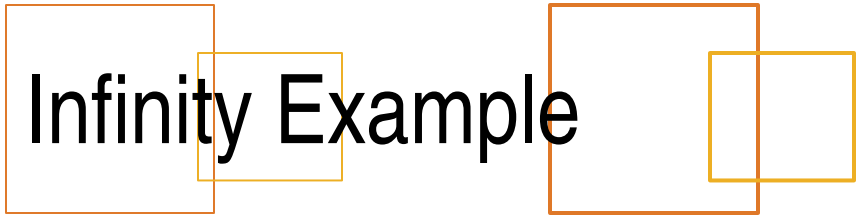


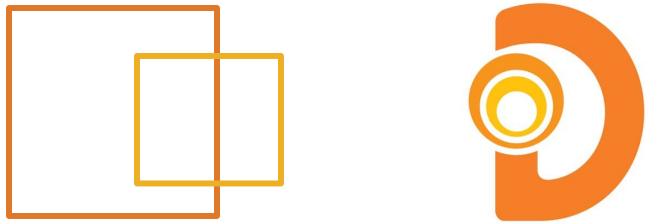
- Floating point are numerical values with fractional parts.
- Two kinds of floating point numbers
- ***float***: 4 bytes long
  - Largest ***float*** is 3.4028234 E +38
  - About 6 or 7 significant digits
- ***double***: is 8 bytes long
  - Largest ***double*** in magnitude is 1.79769313486231570 E +308
  - About 15 significant digits

Infinities, Negative Zeros and Non-numbers

- Double.POSITIVE_INFINITY
- Double.NEGATIVE_INFINITY
- Double.isInfinite(infinity)
- Double.NaN
  - double a = Double.NaN, b = Double.NaN;
  - a != b;
  - But a.equals(b) == true to allow use in hash structures
- -0.0 // negative zero
- Also Float.POSITIVE_INFINITY etc.

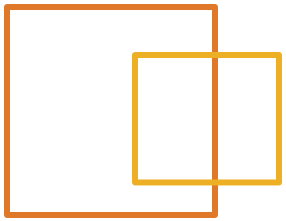
Infinity Example





```
class InfinityTest {
  public static void main(String [] args) {
    // Set up bigd as a large double
    double bigd = 1e306;

    // loop - we should see bigd overflow about the third iteration
    for (int i=1;
         (i<100) && (bigd<Double.POSITIVE_INFINITY);
         i++){
      System.out.println("Iteration="+ i +": bigd="+bigd);
      bigd = bigd * 10.0;
    }//end for loop
  }//end main
}//end class
```

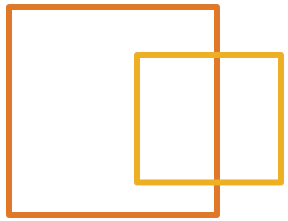
Floating Point Literals



Floating point literals

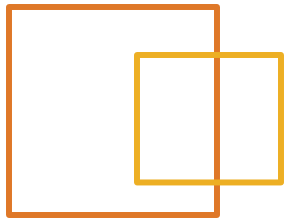
```
38.0        // double
38.0f       // float
38.98D      // double
1.78e23 // double
1.78e23f    // float

-789.983    // double
-1.89e-17F  // float
```

Character Data



- A kind of integral data
  - The type name is ***`char`***
  - Takes on values from 0 to 65535
- Java supports the Unicode standard - each character is stored as a two-byte representation
- ASCII is a subset of Unicode - Java handles the ASCII/Unicode conversions behind the scenes

Character Literals



- Character literals usually represent a single Unicode character in single quotes
- Character literals can also be the numeric code for Unicode characters
  - Unicode escape sequence notation
  - '**\u*dddd***' where ***dddd*** is the hexidecimal representation of the Unicode character
- Certain common non-printable characters, as well as the single and double quote and backslash, have special escape sequences that are recommended for use instead of the corresponding Unicode escape sequence

Character Literals

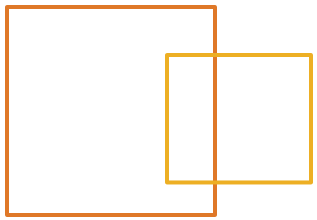
Character literals

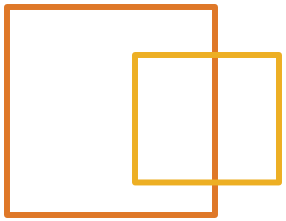
```
'a'     '7'     'ξ'     '©'     ' '
'\''        '\\'        '\u0F34'    '\u0004'    '\ffd1'
'_'

'ab'        // Not a char literal - two characters between quotes
'\ug189'    // Not a char literal - illegal Unicode code.
```

Unicode Escape Sequences

```
    '\b'            /* \u0008: backspace BS */
    '\t'            /* \u0009: horizontal tab HT */
    '\n'            /* \u000a: linefeed LF */
    '\f'            /* \u000c: form feed FF */
    '\r'            /* \u000d: carriage return CR */
    '\"'            /* \u0022: double quote " */
    '\''            /* \u0027: single quote ' */
    '\\'            /* \u005c: backslash \ */
```

Strings

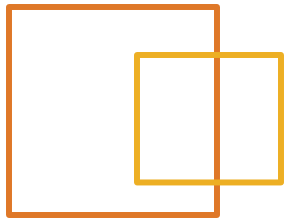




- There is no string primitive data type in Java
  - ***`String`***s are actually a reference data type that is implemented in the Java SE APIs
  - Java allows ***`String`*** data to be used syntactically as if it were a primitive data type in many cases
  - Intended to make working with character strings more "programmer friendly”

- ***`String`*** literals are sequences of Unicode Characters
  - Enclosed in double quotes
  - ***`char`*** escape sequences are valid for ***`String`***

Chars and Strings



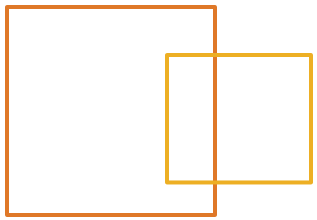
Using Character data

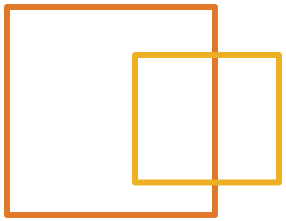
```
char a = 'a';       // single character
char b = 'b';       // single character
char nl = '\n';     // escape code for newline
char x = '\u7878'; // Unicode escape sequence
a + b;           // the result is an int.
```

Strings

```
String s = "This is a string";
s = "This is a string with a backspace \b in it";
s = "This is a string with a \" double quote inside";
String t = s;
t = ""; // this is the empty string
t = 'a';    // illegal! 'a' is not a string.
```

Arrays

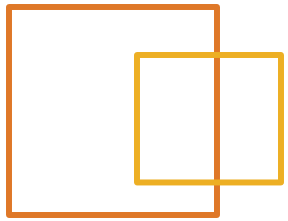




- An array is a data structure that holds multiple values of the same type
  - The values of an array are called the array *elements*
  - They are accessed by index or their numerical position from the start of the array
  - In Java, all arrays are zero-based which means that the index of the first position is 0
  - Initialized arrays have an intrinsic attribute describing the size - ***`length`***
- The easiest way declare and initialize an array:

```
data_type [] array_name = { list, of, initial,
  values };
```

Creating Arrays Example



Creating arrays

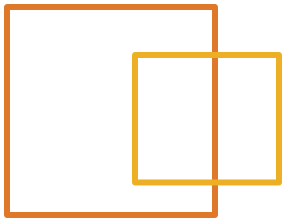
```
class ArrayTest {
  public static void main(String [] args) {
    int [] bob = {9,78,-3,0,89 };
    String [] a = {"black", "brown", "white",
                   "green", "blue", "brown"};
  }
}
```

The Array a

black	brown	white	green	blue	brown
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]

Fig. 4-1: **Array from Example 2-21**

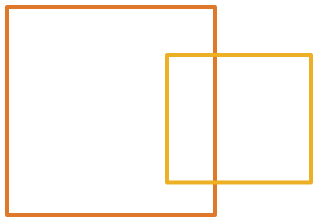
Using Arrays Example

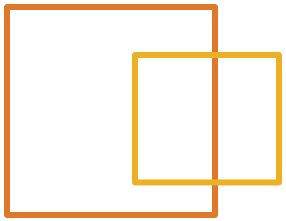


Using arrays

```
class ArrayTest2 {
  public static void main(String [] args) {
    int [] bob = {9,78,-3,0,89 };
    String [] a = {"black", "brown", "white", "green",
                                          "blue", "brown"};
    int index = 0;
    while (index < a.length) {
      System.out.println("a["+index+"] ->"+a[index]);
      index++;
    }
    index = 0;
    while (index < bob.length) {
      if (index == 3 || index == 2) bob[index]=9999;
      System.out.println("bob["+index+"] ->"+bob[index]);
      index++;
    }
  }
}
```

Summary





We covered

- The rules for creating legal variable names in Java
- Describe and use the basic primitive data types in Java
- Use String data
- Determining the data type of a literal