Primitive Data Types

Variables

variable: a piece of computer memory that holds a data value

Two parts to every variable:

1. *identifier*: the name by which we refer to the variable
2. *data type*: the type of data the variable holds (e.g., string, number, boolean)

Types of Data Type

Two categories: *primitive type* and *class type*

Primitives

represents basic data types

examples:

```
char     //holds a single character
int      //holds integer values
double   //holds decimal values
boolean  //holds true/false values
```

Classes

represents more complex data

examples:

```
String  //** holds textual data
Scanner //reads input
Date    //represents day/month/year
Math    //complex mathematical ops
```

Data

"Carpe Diem"	**42** **3.14159**	`true` `false`
text	numbers	logical values

Data

"Carpe Diem"

42

3.14159

`true`

`false`

text

numbers

logical values

Primitive Data Types in Java

Integer Numeric Types (can only be whole numbers)

`byte`	1 byte	-128	through	127
`short`	2 bytes	-32678	through	32677
`int`	4 bytes	-2147483648	through	2147483647
`long`	8 bytes	-9223372036854775808	through	9223372036854775807

Decimal Numeric Types (can be whole or decimal numbers)

`float`	4 bytes	7 decimal digits of accuracy
`double`	8 bytes	15 decimal digits of accuracy

Character Type

`char`	2 bytes	any keyboard character

Logical Type

`boolean`	1 byte	`true` or `false`

Declaration & Initialization of Primitive Variables

declare a single variable

```
int age;
```

initialize a primitive variable

```
age = 29;
```

declare & initialize a single primitive variable

```
int age = 29;
```

declare & initialize multiple primitive variables **of the same type**

```
int age = 29, weight, temp = -10;
```

Declaring & Initializing Numeric Data Types

integer numeric types

```
int age = 29;
int temp = -4;
```

decimal numeric types

```
double height = 5.33;
double length = 5.0;  // note the use of the decimal!
double width = 3;  // note the lack of a decimal!
double outdoorTemp = -4.25;
double mole = 6.022E23;
double verySmallNumber = 5.6E-15;
```

Numerical Operators in Java (`int`)

Unary Prefix Operator

`-`	negation	`-6`

Binary Infix Operators

`+`	addition	`6 + 4 (= 10)`
`-`	subtraction	`6 - 4 (= 2)`
`*`	multiplication	`6 * 4 (= 24)`
`/`	division (*quotient*)	`6 / 4 (= 1)`
`%`	modulus, mod (*remainder*)	`6 % 4 (= 2)`

Unary Prefix/Postfix Operators

`++`	increment by 1
`--`	decrement by 1

Division & Modulus (Mod) for `int`

Division of two integers results in two values: the quotient and remainder

quotient describes how many times the divisor goes into the dividend

remainder describes the amount "left over" from the division

traditional math

```
19 / 4 = 4.75
       = 4 3/4
```

`int` **math**

```
19 / 4 = 4
19 % 4 = 3  // 3/4
```

Operator Precedence

Will work the same way you're familiar with from math

- work from left to right across a mathematical statement, starting with highest precedence
- mod has the same level of precedence as multiply and divide

$$2 + 19 \mathbin{/} \underline{(4 + 1)} - 5 \mathbin{\%} 3$$

$$2 + \underline{19 \mathbin{/} (5)} - 5 \mathbin{\%} 3$$

$$2 + 3 - \underline{5 \mathbin{\%} 3}$$

$$\underline{2 + 3} - 2$$

$$\underline{5 - 2}$$

$$3$$

Numerical Operators in Java (`double`)

Unary Prefix Operator

`-`	negation	`-6.2`

Binary Infix Operators

`+`	addition	`6.2 + 4.1 (= 10.3)`
`-`	subtraction	`6.2 - 4.1 (= 2.1)`
`*`	multiplication	`6.2 * 4.1 (= 25.42)`
`/`	division (*quotient*)	`6.2 / 4.1 (= 1.51…)`
`%`	modulus, mod (*remainder*)	`6.2 % 4.1 (= 2.10…)`

N.B.: you will rarely (if ever) use this with doubles!

Unary Prefix/Postfix Operators

`++`	increment by 1
`--`	decrement by 1

Prefix/Postfix Increment/Decrement (`int` & `double`)

```
int age = 29;

age = age + 1;
age = age - 1;

++age; //age = 30 after this line
age++; //age = 31 after this line

--age; //age = 30 after this line
age--; //age = 29 after this line

age = age--; //never do this!
age = ++age; //never do this!
```

Frequently want to increase/decrease an `int`/`double` variable by 1

We can use the increment/decrement operators as shorthand to do this

Two forms: prefix and postfix

- prefix has the operator *before* the variable
- postfix has the operator *after* the variable

Always use it by itself!

Arithmetic Shortcut Operators (`int` & `double`)

```
int x = 5;
```

Operator	Example	Equivalent To	Result
`+=`	`x += 2;`	`x = x + 2;`	`x = 7`
`-=`	`x -= 2;`	`x = x - 2;`	`x = 3`
`*=`	`x *= 2;`	`x = x * 2;`	`x = 10`
`/=`	`x /= 2;`	`x = x / 2;`	`x = 2`
`%=`	`x %= 2;`	`x = x % 2;`	`x = 1`

More Complex Operations

What if we want to…

- take the square root of a number?
- display a number in a particular format (e.g., currency)?
- generate a random number?

We can use *classes*, which represent/manipulate more complex data

Math Class

Provides a range of methods for advanced mathematical operations

- square root/powers
- logarithms
- trigonometric functions
- constant values (e, π)

Math Class

returns the result of calculating $< base >^{<exponent>}$ (e.g., 2^3)

```
Math.pow(<base>, <exponent>);
```

returns the result of calculating $\sqrt{< expression >}$ (e.g., √9)

```
Math.sqrt(<expression>);
```

returns the absolute value of < *value* >

```
Math.abs(<value>);
```

Math Class

returns the smaller value between <num1> and <num2>

```
Math.min(<num1>, <num2>);
```

returns the larger value between <num1> and <num2>

```
Math.max(<num1>, <num2>);
```

returns the value of π as a `double`

```
Math.PI;
```

`DecimalFormat` Class

Allows us to format numeric values in particular way

- currency
- specific number of decimal places

Uses a pattern String to indicate formatting

- `0`: displays a digit
- `#`: displays a digit, unless a leading zero (then omitted)
- `.` : displays a decimal
- `,` : displays a comma

Example: DecimalFormat

```java
double x = 0.329523;
DecimalFormat df1 = new DecimalFormat("0.0");
DecimalFormat df2 = new DecimalFormat("0.00");
DecimalFormat df3 = new DecimalFormat("00.00");
DecimalFormat df4 = new DecimalFormat(“#0.00");

System.out.println("X = " + df1.format(x));
System.out.println("X = " + df2.format(x));
System.out.println("X = " + df3.format(x));
System.out.println("X = " + df4.format(x));
```

```
X = 0.3
X = 0.33
X = 00.33
X = 0.33
```

Example: DecimalFormat

```java
double wage, hours;
double pay;

// Ask user for their 'wage' and 'hours' worked
// Calculate their pay for the week

pay = hours * wage;
System.out.print("Total pay for " + hours + " hours of work ");
System.out.print("is $" + pay);
```

```
Enter Wage : 20.00
Enter Hours: 51.0
Total pay for 51.0 hours of work is $1020.0
```

Example: DecimalFormat

```java
double wage, hours;
double pay;
DecimalFormat df = new DecimalFormat("$###,##0.00");

// Ask user for their 'wage' and 'hours' worked
// Calculate their pay for the week

pay = hours * wage;
System.out.print("Total pay for " + hours + " hours of work ");
System.out.print("is " + df.format(pay));
```

```
Enter Wage : 20.00
Enter Hours: 51.0
Total pay for 51.0 hours of work is $1,020.00
```

Mixing `int` & `double` Values

Sometimes, we might want to mix `int` & `double` values

Consider the following equation; what does it evaluate to?

```
double x = 2.5 + 9 / 2;
             2.5 +   4  ;
                ???     ;
```

N.B.: uses `int` division! Java assumes numbers without a decimal (e.g., 3 vs 3.0) are ints when not stored in a variable

Java requires both inputs of an operator to be of the same data type

achieves this through the process of *coercion*

Coercion

coercion: automatically changing a value's type to enable an operation

always coerced to the widest type necessary

There is a strict ordering on types

narrower types → *wider* types

`byte`	1 byte	-128	through	127
`short`	2 bytes	-32678	through	32677
`int`	4 bytes	-2147483648	through	2147483647
`long`	8 bytes	-9223372036854775808	through	9223372036854775807
`float`	4 bytes	7 decimal digits of accuracy		
`double`	8 bytes	15 decimal digits of accuracy		

Mixing `int` & `double` Values

```
double x = 2.5 + 9 / 2;
           2.5 +   4  ;
             6.5      ;
```

double int int

double double

double

What if we want to force double division here?

Mixing int & double Values

```
double x = 2.5 + 9 / 2.0;
          double  int  double
           2.5 +   4.5  ;
          double    double
               7.0      ;
              double
```

Casting

casting: explicitly changing the data type of a value

- can cast to a narrower or wider type
- always initiated by the programmer

```
(<dataTypeToCastTo>) <expression>;
```

```
int num;
num = (int) 5.33;  // results in num = 5
double perc;
perc = 93 / (double) 100;  // results in perc = 0.93
perc = 93 / ((double) 100);  // results in perc = 0.93
perc = (double) num / 100;  // results in perc = 0.05
```

Data

“Carpe Diem”	42 3.14159	`true` `false`
text	numbers	logical values

The char Data Type

Similar to `String`, but contains exactly one character

- uses single quotes (‘) instead of double quotes (“)

Has a few operations, but we’re only concerned with assignment for now

Will primarily use it with `String` methods

```
String exampleStr = "Hello, home!";

int index = exampleStr.indexOf('h'); //index = 7
char charPos = exampleStr.charAt(5); //charPos = ','
```

N.B.: char values

Declaring & Initializing the char Data Type

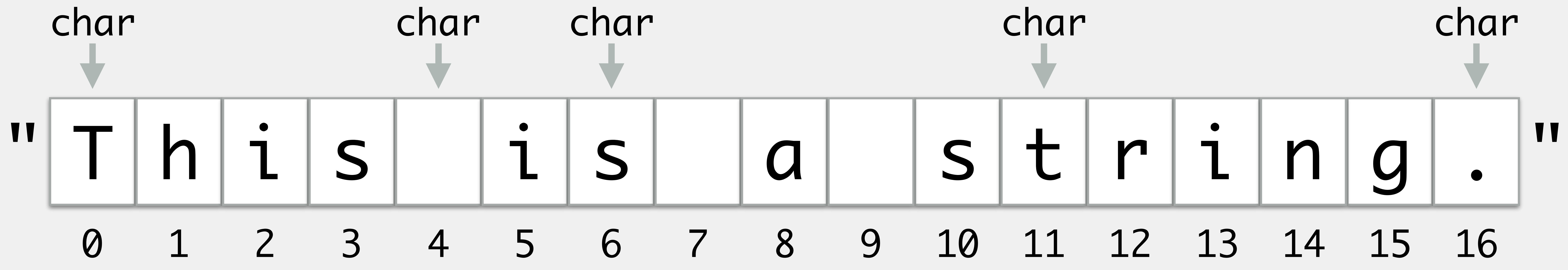
character type

```
char letterA = 'a';
char space = ' ';
char bang = '!';
```

can be an escape sequence too

```
char singleQuote = '\'';
char tab = '\t';
char lineBreak = '\n';
```

Strings



Strings are a collection of char values concatenated together

Data

“Carpe Diem”

text

42

3.14159

numbers

`true`

`false`

logical values

Logical Data

Can express exactly one of two values: true or false

- in programming, we also think of these as 1 (true) and 0 (false)

Operators are used to express logical ideas that can be evaluated

- && (and)
- || (or)
- ! (not)

`&&` (`and`)

Can express whether or not two statements are true

- it is raining **and** it is cold
- I attend UWL **and** I am a science major

If one or both of the statements are false, then the entire expression is false

Evaluation:

```
0 && 0 is 0
0 && 1 is 0
1 && 0 is 0
1 && 1 is 1
```

Truth Tables

Truth table: a table where each row corresponds to one combination of inputs, columns for statements give the input values, and subsequent columns give the truth value for the results of individual operators

P	Q	P && Q
0	0	0
0	1	0
1	0	0
1	1	1

&& (and)

P	Q	P && Q
0	0	0
0	1	0
1	0	0
1	1	1

0 = false

1 = true

|| (or)

Can express whether one or both of two statements are true

- it is raining **or** it is cold
- I attend UWL **or** I am a science major

If one **or both** of the statements are true, then the entire expression is true

Nuances of ||

In English, we use “or” to present two mutually exclusive possibilities

- e.g., “Did you have pizza or spaghetti for dinner?”
- possible answers: pizza, spaghetti, neither, both (maybe?)

Logically, the answer could be “yes” or “no”

- no: you had neither
- yes: you had spaghetti, or pizza, or both

Spectrum of possible answers does not work with our logical value system

- we instead work with true (yes) or false (no)

|| (or)

P	Q	P || Q
0	0	0
0	1	1
1	0	1
1	1	1

0 = false

1 = true

! (not)

Can express the opposite value of a single statement

- it is **not** raining
- I am **not** a science major

If the statement is true, the expression is false, and vice versa

! (not)

P	!P
0	1
1	0

0 = false

1 = true

Expressing More Complex Ideas

Often want to express more complex ideas

- “Show up to lab or don’t show up to lab and submit exercise three”

Want to know the outcome of every possible scenario (set of inputs)

Can combine statements into larger expressions

```
goToLab || (!goToLab && submitEx3)
```

How to evaluate possible outcomes?

- use truth tables
- one statement at a time

Example: Truth Table for Complex Expressions

goToLab || (!goToLab && submitEx3)

goToLab	submitEx3

Create one column per variable

- list in alphabetical order

For N variables, you will have 2^N additional rows

- in this case, $2^2 = 4$

Fill rows with every combination of 0s and 1s

- easiest way? count in binary
- i.e., count using only 0s and 1s

Counting

Decimal			Binary	
0	8	16	0	1000
1	9	17	1	1001
2	10	18	10	1010
3	11	19	11	1011
4	12	20	100	1100
5	13	21	101	1101
6	14	22	110	1110
7	15	23	111	1111

Counting

Decimal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Example: Truth Table for Complex Expressions

goToLab || (!goToLab && submitEx3)

goToLab	submitEx3
0	0
0	1
1	0
1	1

Create one column per variable

- list in alphabetical order

For N variables, you will have 2^N additional rows

- in this case, $2^2 = 4$

Fill rows with every combination of 0s and 1s

- easiest way? count in binary
- i.e., count using only 0s and 1s

Precedence for Logical Operators

<table>
<tr><th>Description</th><th>Operator(s)</th></tr>
<tr><td>precedence</td><td>()</td></tr>
<tr><td>negation</td><td>!</td></tr>
<tr><td>logical AND</td><td>&&</td></tr>
<tr><td>logical OR</td><td>||</td></tr>
</table>

<table>
<tr><th>P</th><th>Q</th><th>+
P || Q</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>

<table>
<tr><th>P</th><th>Q</th><th>*
P && Q</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>

It matters!

Work out **P && Q || R** two ways: performing || first and performing && first

Example: Truth Table for Complex Expressions

goToLab || (!goToLab && submitEx3)

goToLab	submitEx3	!goToLab	!goToLab && submitEx3	goToLab || (!goToLab && submitExercise3)
0	0	1	0	0
0	1	1	1	1
1	0	0	0	1
1	1	0	0	1

P	Q	P && Q
0	0	0
0	1	0
1	0	0
1	1	1

P	Q	P || Q
0	0	0
0	1	1
1	0	1
1	1	1

The boolean Data Type

Can only contain one of two values: `true` or `false`

Declaration/initialization/assignment work the same as `int`, `double`, `char`

Uses the logical operators (i.e., !, ||, &&)

```
boolean entree = true;
boolean salad = false;
boolean soup = true;

boolean validOrder = entree && (salad || soup);
```

What is `validOrder` set to?

`true`

`boolean` Operators

Uses the logical operators (i.e., !, ||, &&)

Also uses *relational* and *equality operators*

Description	Operator(s)
precedence	()
negation	!
relational	< > <= >=
equality	== !=
logical AND	&&
logical OR	||

Relational and Equality Operators

< (less than)

8 < 3 (false), 3 < 8 (true)

> (greater than)

8 > 3 (true), 3 > 8 (false)

<= (less than or equal to)

6 <= 6 (true), 6 <= 7 (true)

7 <= 6 (false)

>= (greater than or equal to)

6 >= 6 (true), 6 >= 7 (true)

7 >= 6 (false)

== (equality)

6 == 6 (true), 8 == 3 (false)

!= (inequality)

6 != 6 (false), 8 != 3 (true)

Example: `boolean` Expressions

Description	Operator(s)
precedence	()
negation	!
relational	< > <= >=
equality	== !=
logical AND	&&
logical OR	||

```
boolean x = 2.5 > 3 || !(4 != 5.1);
            2.5 > 3 || !(   true   );
         double   int  int boolean double
            2.5 > 3 ||      false   ;
         double   int      boolean
              false  ||     false   ;
                     false          ;
```

Operator Precedence

We can mix types, operators in a single expression

<table>
<tr><th>Description</th><th>Operator(s)</th></tr>
<tr><td>precedence</td><td>()</td></tr>
<tr><td>prefix</td><td>! -</td></tr>
<tr><td>multiplicative</td><td>* / %</td></tr>
<tr><td>additive</td><td>+ -</td></tr>
<tr><td>relational</td><td>< > <= >=</td></tr>
<tr><td>equality</td><td>== !=</td></tr>
<tr><td>logical AND</td><td>&&</td></tr>
<tr><td>logical OR</td><td>||</td></tr>
</table>

```
boolean x = 2.5 + 4 > 3 || !(4 % 2 != 5.1);
            2.5 + 4 > 3 || !(  0   != 5.1);
            2.5 + 4 > 3 || !(    true    );
            2.5 + 4 > 3 ||      false     ;
              6.5   > 3 ||      false     ;
                 true   ||      false     ;
                       true               ;
```

Short-Circuit Evaluation

Two situations where evaluation of && and || will be terminated early

```
false && …
```

```
true || …
```

Java will always compute the lefthand side of an operator first

```
2.5 + 4 > 3 || !(4 % 2 != 5.1)
  6.5   > 3 || !(4 % 2 != 5.1)
     true   || !(4 % 2 != 5.1)
           true
```

Short-Circuit Evaluation

Two situations where evaluation of && and || will be terminated early

```
false && …
true || …
```

Java will always compute the lefthand side of an operator first

```
2.5 + 4 > 3 || !(4 % 2 != 5.1)
  6.5   > 3 || !(4 % 2 != 5.1)
     true   || !(4 % 2 != 5.1)
           true
```

```
int num = ...; //user input
boolean divByNum;
divByNum = 2 >= 1 / num;
divByNum = num != 0 && 2 >= 1 / num;
```

entire expression evaluates to false if num != 0 is false (i.e., if num is 0)