

2. Data and Expressions

- Objectives - when we have completed this set of notes, you should be familiar with:
 - character strings and escape sequences
 - variables and assignment
 - primitive data
 - if and if-else statements
 - expressions and operator precedence
 - Accepting standard input from the user
 - data conversions



Character Strings

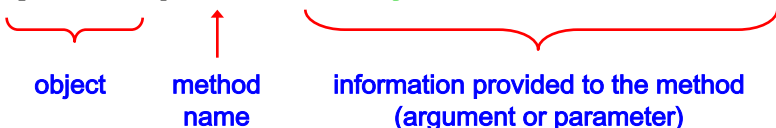
- A string of characters can be represented as a *string literal* by putting double quotes around the text:
- Examples:
 - `"This is a string literal."`
 - `"Pat Doe, 123 Main Street"`
 - `"7"`
- When your program is running, a character string is an object in Java, defined by the `String` class
- Every string literal represents a `String` object



The println Method

- Recall that the `println` method prints a character string and then advances to the next line
- The `System.out` object is an output stream corresponding to a display destination (the monitor screen)

```
System.out.println ("War Eagle from the Auburn Plains!");
```



The diagram illustrates the components of the `println` statement. A red bracket under `System.out` is labeled "object". A red arrow points to `println` and is labeled "method name". A red bracket under the string `"War Eagle from the Auburn Plains!"` is labeled "information provided to the method (argument or parameter)".



The print Method

- The `print` method for the `system.out` object is similar to the `println` method, except that it does not advance to the next line after it prints
- Therefore anything printed after a `print` statement will appear on the same line
- See [CountOff.java](#)



String Concatenation

- The *string concatenation operator* (+) appends one string to the end of another

`"Peanut butter " + "and jelly"`

- A string literal cannot be broken across two lines in a program
- It can also append a **number** to a **string**
- See [ConcatenationExample1](#)



String Concatenation

- The + operator is a binary operator applied to two operands; if at least one the operands is a String then string concatenation is done

`5 + " years"` results in `"5 years"`

- The + operator also used for addition if both operands are numeric

`5 + 10` results in `15`

- The + operator is evaluated left to right, but parentheses can be used to force the order
- See [ConcatenationExample2](#)
(Experiment with String expressions in the interactions pane in jGRASP)



Escape Sequences

- What if we wanted to print a quotation mark " (a.k.a. double quote)?
- The following line would cause a compile-time error - it would interpret the second quote as the end of the string

```
System.out.println ("I said "Hello" to you.");
```



- An *escape sequence* represents a special character
- An escape sequence begins with a backslash character (\)

```
System.out.println ("I said \"Hello\" to you.");
```



Escape Sequences

- Some Java escape sequences:

<u>Escape Sequence</u>	<u>Meaning</u>
\t	tab
\n	newline
\r	carriage return
\"	double quote
\'	single quote
\\	backslash

- \r\n are used together by println in Windows to move to the next line
- See [EscapeSeq.java](#)



Variables

- A *variable* is a name for a “location” in memory that holds a value
- There are many types of values or data...
 - integers values (e.g., -60, 0, 1, 7, 23)
 - floating point values (e.g., -5.6, 0.0, 2.4, 35.2)
 - characters values (e.g., 'j', 'P', '5')
 - boolean values (**true**, **false**)
 - references to objects
- We'll focus on `int` types (integer values) for now and then examine the other types later



Variables

- A variable must be *declared* with the type of information that it will hold or reference

type (integer) variable name

```
int total;
```

Multiple variables can be created in one declaration

```
int count, temp, result;
```



Variable Initialization

- A variable can be "initialized" to a particular value

```
int sum = 0;  
int base = 32, max = 149;
```

- When a variable is referenced in a program, its current value is used

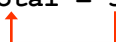
```
System.out.println("base is " + base);  
would print...  
base is 32
```





Assignment

- An *assignment statement* changes value of variable

```
total = 55;
```



- The *assignment operator* =
- How does it work?
 - Evaluate the expression on the right side
 - Store the result in the variable on the left side (previous value is overwritten)
- Java is *strongly typed*: variable type and expression type must be compatible!
- See [VariablesExample.java](#) (Run in Canvas , )



Primitive Data

- There are 8 primitive data types in Java
- Integer types:
 - byte, short, **int**, long **int** age = 19;
- Floating point types:
 - float, **double** **double** avg = 94.8;
- Character type:
 - char **char** letter = 'A';
- Boolean type:
 - boolean **boolean** isCold = **false**;



Expressions

- An *expression* is a construct made up of one or more variables, operators, and method invocations that evaluates to a single value
- *Arithmetic expressions* compute numeric results and make use of the *arithmetic operators*:

Addition	+
Subtraction	-
Multiplication	*
Division	/
Remainder (Modulus)	%
- If either operand is a floating point value, then the result is a floating point value



Division and Remainder

- If both operands to the division operator (/) are integer types, the result is an integer (the fractional part is discarded)

14 / 3 equals 4

8 / 12 equals 0

- The remainder (or mod) operator (%) returns the remainder after dividing the first operand by the second

14 % 3 equals 2

8 % 12 equals 8

[RemainderCheck.java](#)



Assignment Revisited

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

First, one is added to the original value of count

count = count + 1;



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



Increment and Decrement

- The increment and decrement operators use only one operand
- The *increment operator* (++) adds one to its operand
- The *decrement operator* (--) subtracts one from its operand
- The statement

```
count++;
```

is functionally equivalent to

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



Assignment Operators

- Often we perform an operation on a variable, and then store the result back into that variable
- Java provides *assignment operators* to simplify that process
- For example, the statement

```
num += count;
```

is equivalent to

```
num = num + count;
```



Characters

- A `char` variable stores a single character
- *Character literals* are in single quotes:
`'a' 'x' '7' '$' ',' '\n'`
- Example declarations:
`char topGrade = 'A';`
`char terminator = ';', separator = ' ';`
- A primitive character variable holds only one character, while a `String` object holds multiple characters



Boolean

Q3

- A `boolean` value represents a true or false condition
- The reserved words `true` and `false` are the only valid values for a boolean type
`boolean done = false;`
- A `boolean` variable can also be used to represent any two states, such as a light bulb being *on* or *off*



Relational Operators

- Boolean values can be calculated using relational operators

Operator	Meaning
==	Equal
!=	Not equal
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal

- Example:

```
boolean greater = 89 > 50; // greater set to true
int temp = 99;
boolean isCold = temp < 50; // isCold set to false
```



if Statements

- Allows a program to execute a statement or block { } only under certain conditions:

```
int temp = 39;
if (temp < 50) {
    System.out.println("It's cold!");
}
System.out.println("Temp = " + temp);
```

[IfExample.java](#)



if Statements

- You can also use a boolean variable:

```
int temp = 39;
boolean isCold = temp < 50;
if (isCold) {
    System.out.println("It's cold!");
}
System.out.println("Temp = " + temp);
```

[IfExample2.java](#)



if-else Statements

- What if you wanted to execute one statement or block { } for a true condition and a different statement or block { } for a false condition?

```
int num1 = 9, num2 = 7;
if (num1 < num2) {
    System.out.println(num1 + " is < " + num2);
}
else {
    System.out.println(num1 + " is >= " + num2);
}
System.out.println("Done!");
```

- What is the output? [IfElseExample.java](#)
- What if num1 and num2 both hold value 10?



Interactive Programs Using Standard Input

- Programs generally need user input
- The `Scanner` class provides methods for reading input values of various types
- A `Scanner` object can be set up to read input from various sources (including keyboard input)
- Keyboard input is represented by the `System.in` object





Numerical Input Example

- The following line creates a `Scanner` object that reads from the keyboard:

```
Scanner scan = new Scanner(System.in);
```

- The `new` operator creates the `Scanner` object
- Once created, the `Scanner` object can be used to get user input. For example, `nextInt` retrieves an integer value:

```
int numberItems = scan.nextInt();
```

- See [Difference.java](#) (also Run in Canvas  )



Part 2

- More on primitive types
- Character sets
- Operator precedence
- Increment and Decrement: prefix and postfix form
- Data conversion
- Reading user input



Numeric Primitive Data

- Why have multiple types for integer and floating point values? They are different sizes in memory, which dictate the range of possible values

<u>Type</u>	<u>Storage</u>	<u>Min Value</u>	<u>Max Value</u>
byte	8 bits	-128	127
short	16 bits	-32,768	32,767
int	32 bits	-2,147,483,648	2,147,483,647
long	64 bits	$< -9 \times 10^{18}$	$> 9 \times 10^{18}$
float	32 bits	$\pm 3.4 \times 10^{38}$ with 7 significant digits	
double	64 bits	$\pm 1.7 \times 10^{308}$ with 15 significant digits	



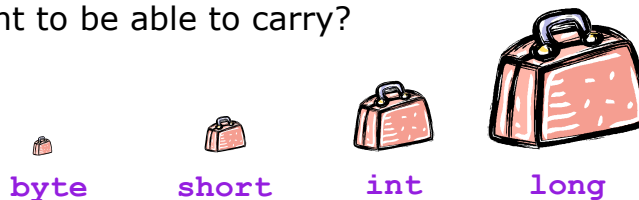
Numeric Primitive Data

- Suppose you want to declare a variable to hold an integer value
- You could use a byte value...
`byte` `scheduledCourses;`
 - Takes up only a small space (8 bits)
 - However, it can only be between -127 and 127
- Or an int value
`int` `storeInventory;`
 - Now you can go all the way to 2,147,483,647!
 - However, reserves much more space (32 bits)



Numeric Primitive Data

- **Think of it as picking out a suitcase.** How much space do you have? How much do you want to be able to carry?

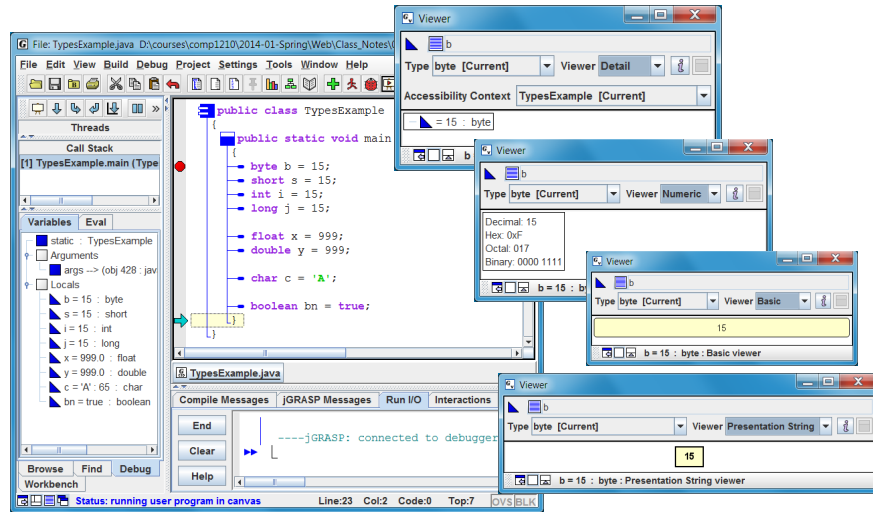


- Your computer/phone/etc has plenty of space, so use **int** and **double** values "just in case"

[TypesExample.java](#) (set breakpoint; Debug ; single step ; open viewers by dragging variables from Debug tab)



jGRASP Viewers for byte



jGRASP Numeric Viewers for Primitive Types (except boolean) in a Canvas Window

byte b

Decimal: 15
Hex: 0xF
Octal: 017
Binary: 0000 1111

short s

Decimal: 15
Hex: 0xF
Octal: 017
Binary: 0000 0000 0000 1111

int i

Decimal: 15
Hex: 0xF
Octal: 017
Binary: 0000 0000 0000 0000 0000 0000 1111

long j

Decimal: 15
Hex: 0xF
Octal: 017
Binary: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111

float x

value = 15.0

Sign	Exponent	Mantissa
0	1000 0010 111 0000 0000 0000 0000 0000	0x700000
0	0x82	0x700000
0	130	7340032

sign = +
exponent = 130 - bias of 127 = 3
mantissa = assumed 1 + 7340032 / 2²³
= (approximately) 1.875
value = (sign) mantissa * 2^{exponent}
= +1.875 * 2³
= 15.0

double y

value = 15.0

Sign	Exponent	Mantissa
0	100 0000 0010 1110 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	0xe000000000000000
0	0x402	0xe000000000000000
0	1026	3940649673949184

sign = +
exponent = 1026 - bias of 1023 = 3
mantissa = assumed 1 + 3940649673949184 / 2⁵²
= (approximately) 1.875
value = (sign) mantissa * 2^{exponent}
= +1.875 * 2³
= 15.0

char c

Character: A
Source format: 'A'
Decimal: 65
Hex: 0x41
Octal: 0101
Binary: 0000 0000 0100 0001

boolean bn

bn = true : boolean

Open [TypesExample.java](#)
Run in Canvas then
Play or single step



Character Sets

- A *character set* is an ordered list of characters, and character represents a unique number
 - A `char` variable in Java can store any character from the *Unicode character set*
 - The Unicode character set uses sixteen bits per character, allowing for 65,536 unique characters
 - It is an international character set, containing symbols and characters from many world languages
- *Experiment with String expressions in the interactions pane in jGRASP*



Character Sets

- The *ASCII character set* is older and smaller than Unicode, but is still quite popular
- The ASCII characters are a subset of the Unicode character set, including:

uppercase letters	A, B, C, ...
lowercase letters	a, b, c, ...
punctuation	period, semi-colon, ...
digits	0, 1, 2, ...
special symbols	&, , \, ...
control characters	carriage return, tab, ...



Operator Precedence [Q4](#) [Q5](#)

- Operators can be combined into complex expressions

```
result = total + count / max - offset;
```
- Operators have a precedence which determines the order in which they are evaluated
- Multiplication, division, and remainder are evaluated before addition, subtraction, and string concatenation
- Arithmetic operators with the same precedence are evaluated from left to right, but parentheses can 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



Assignment Revisited

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

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

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

4 1 3 2



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

Q6



Increment and Decrement Q7

- The increment and decrement operators can be applied in *postfix form*:
`count++` uses old value in the expression, then increments
- or *prefix form*:
`++count` increments then uses new value in the expression
- When used as part of a larger expression, the two forms can have different effects
 - Use the increment and decrement operators with care

[IncrementOperatorExample](#)



Assignment Operators

- There are many assignment operators in Java, including the following:

<u>Operator</u>	<u>Example</u>	<u>Equivalent To</u>
<code>+=</code>	<code>x += y</code>	<code>x = x + y</code>
<code>-=</code>	<code>x -= y</code>	<code>x = x - y</code>
<code>*=</code>	<code>x *= y</code>	<code>x = x * y</code>
<code>/=</code>	<code>x /= y</code>	<code>x = x / y</code>
<code>%=</code>	<code>x %= y</code>	<code>x = x % y</code>



Assignment Operators

- The right hand side of an assignment operator can be a complex expression
- The entire right-hand expression is evaluated first, then the result is combined with the original variable
- Therefore

```
result /= (total-MIN) % num;
```

is equivalent to

```
result = result / ((total-MIN) % num);
```



Data Conversion

- Sometimes it is necessary to convert data from one type to another
- For example, we may want to treat an integer as a floating point value
- Conversions must be handled carefully to avoid losing information



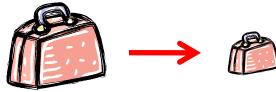
Data Conversion

- *Narrowing conversions* go from a large data type to a smaller one or from a floating point type to an integer type which has less detail
 - If the an int value was 700 (larger than the max byte value of 127), information would be lost when converted to an byte
 - If your grade of 89.8 (a double) was converted to an int type, the new value would be 89 (a 'B'!) ☹
- *Widening conversions* go from a smaller to larger data type or from an integer type to a floating point type which has more detail
 - If a 'byte' with value 95 was converted to an 'int' type, the new value would still be 95 (your new grade could now go to 2,147,483,647!) ☺



Data Conversion

- Think about the suitcase example...
 - Narrowing conversion : you may lose data going from a larger data type to a smaller data type



Not ok if the larger one was full!

- In Java, data conversions can occur in three ways:
 - assignment conversion
 - promotion
 - casting



Assignment Conversion

Q8

- *Assignment conversion*: a value of one type is assigned to a variable of another; example:
 - Variable `money` is a `double` type. Variable `dollars` is an `int` type.
 - The assignment below converts the value in `dollars` to a `double` as it assigns it to `money`

```
money = dollars;
```

- The type and value of `dollars` did not change
- Allows only widening conversions



Data Conversion

[Q9](#) [Q10](#)

- *Promotion* happens when operators in expressions convert their operands

- For example:

`sum` is a **double** (as is `result`)

`count` is an **int**

The value of `count` is converted to a floating point value to perform the following calculation:

```
result = sum / count;
```



Casting

[Q11](#)

- *Casting* allows narrowing conversions and widening conversions, so be careful!
- It is also easy to detect in code
- To cast, the type (in parentheses) is placed in front of the value being converted
- For example, if `total` and `count` are integers, the value of `total` could be converted to a **double** with a cast to avoid integer division:

```
result = (double) total / count;
```



Constants

- A *constant* is similar to a variable, but it is placed at the class level (e.g., above the main method), written in all CAPS with underscores, and its initial value cannot be changed
- The `static` modifier allows it to be shared among all methods in the class; the `final` modifier prevents the initial value from changing

```
static final int MIN_HEIGHT = 69;
```

- The compiler will issue an error if you try to change the value of a constant



Constants

- Constants are useful for three important reasons...
 1. Constants improve code readability
 - For example, `MAX_LOAD` means more than the literal 250 (a.k.a., a magic number)
 2. Second, they facilitate program maintenance
 - If a constant is used in multiple places, its value need only be updated in one place
 3. Third, they prevent a value from changing, avoiding inadvertent errors by other programmers
- Constants will be revisited in Chapter 4



Reading Input

- The `Scanner` class is part of the `java.util` class library, and must be imported into a program to be used:

```
import java.util.Scanner;
```
- See [ReadLineExample](#)
- The `nextLine` method reads all of the input until the end of the line is found
- Object creation and class libraries are discussed further in Chapter 3



Input Tokens

- Unless specified otherwise, *white space* is used to separate the elements (called *tokens*) of the input
- White space includes space characters, tabs, new line characters
- The `next` method of the `Scanner` class reads the next input token and returns it as a string
- Methods such as `nextInt` and `nextDouble` read data of particular types
- See [DinnerForGroup](#) (Run in Canvas 🖥️; ▶)





Scanning a String

[Q12](#)

- A Scanner object can be created to scan any String, breaking it into tokens
- Suppose we want to separate a phrase into words and print each word on a separate line

```
Scanner scan = new Scanner("this is a test");  
System.out.println(scan.next());  
System.out.println(scan.next());  
...
```

[StringScan.java](#) (Run in Canvas ; single step )

