2. Data and Expressions

- Objectives - when we have completed this chapter, you should be familiar with:
  - character strings & 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"
```

- Every character string is an object in Java, defined by the `String` class

- Every string literal represents a `String` object

The println Method

Q

- Recall that the `println` method prints a character string
- 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!");
```

object

method name

information provided to the method (a parameter)

The print Method

- The `print` method in the `system.out` object is similar to the `println` method, except that it does not advance to the next line
- 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"
```

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

String Concatenation

- The + operator also used for arithmetic addition if both operands are numeric
  `5 + 10` → `15`
- If one or both operands is a string, + performs string concatenation
  `5 + " years"` → `"5 years"`
  `"years: " + 5` → `"years: 5"`
- 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

Q

- What if we wanted to print a quote character?
- 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:

Escape Sequence	Meaning
`\b`	backspace
`\t`	tab
`\n`	newline or line feed (LF)
`\r`	carriage return (CR)
`\"`	double quote
`\'`	single quote
`\\`	backslash

- Use `\r\n` together to move to the next line
- See `EscapeSeq.java`

Variables

- A *variable* is a name for a “location” in memory that allows us to store and retrieve program data
- There are many types of data…
  - integers (-60, 7, 23, etc)
  - floating point types (-5.6, 2.4, 35.2, etc)
  - characters (`'j'`, `'P'`, `'5'`, etc)
  - boolean values (`true`, `false`)
- We’ll examine the details of the different types later; let’s focus on **`int`** types (32 bit integer values) for now.

Variables

- A variable must be *declared* with the type of information that it will hold before it can be used

data type (integer)

variable name

```
int total;
```

Multiple variables can be created in one declaration

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

Variable Initialization

- When a variable is declared, it 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;
```

- Uses the *assignment operator*: **=**
- How does it work?
  - Evaluates the expression on right side
  - Stores the value 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

Primitive Data

- There are 8 primitive data types in Java
- Integer types :
  - `byte`, `short`, **`int`**, `long`
- Floating point types:
  - `float`, **`double`**
- Character type:
  - `char`
- Boolean type:
  - `boolean`

<u>Examples</u>

```
int num1 = -4;
double num2 = 1.2;
char c = 'a';
boolean b = true;
```

Expressions

- An *expression* is a combination of one or more operators and operands
- *Arithmetic expressions* compute numeric results and make use of the *arithmetic operators*:

Addition	+
Subtraction	-
Multiplication	*
Division	/
Remainder	%

- If one of the operands in an arithmetic expression is floating point, then the result is a floating point value

Division and Remainder

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

`11 / 2`	equals	`5`
`7 / 10`	equals	`0`

- For integers, the remainder operator (%) returns the remainder after dividing the first operand by the second

`11 % 2`	equals	`1`
`7 % 10`	equals	`7`

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

Q

- 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 is appropriate when for any variable with two states (e.g., *on*, *off*)

```
boolean lightOn = true;
```

Relational Operators

- Boolean expressions can be formed 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;
int temp = 99;
boolean isCold = temp < 50;
```

If Statments

- A program can perform an operation only under certain conditions.

```
int temp = 85;
double humidity = .60;
if (humidty >= .60) {
   temp = temp + 5;
}
System.out.println("Feels like " + temp +
   " degrees.");
```

Humidity1.java

If Statments

- We can also use a boolean variable to capture the result of evaluating the boolean expression:

```
int temp = 85;
double humidity = .60;
boolean isHotter = humidity >= .60;
if (isHotter) {
    temp = temp + 5;
}
System.out.println("Feels like " + temp +
    " degrees.");
```

Humidity2.java

if-else Statments

- What if we wanted to perform a different operation under a false condition?

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

- What is the output?
- What if num1 and num2 both hold value 10?

IfElseExample.java

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

Part 2

- More on primitive types
- Character sets
- Operator precedence
- Increment and Decrement: prefix & postfix form
- Data conversion
- Reading user input (String values, etc

Numeric Primitive Data

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

Type	Storage	Min Value	Max Value
`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	$+/- 3.4 \times 10^{38}$ with 7 significant digits	
`double`	64 bits	$+/- 1.7 \times 10^{308}$ with 15 significant digits	

Numeric Primitive Data

- Suppose you want to declare an integer type
- You could use a byte value…

```
byte scheduledCourses;
```

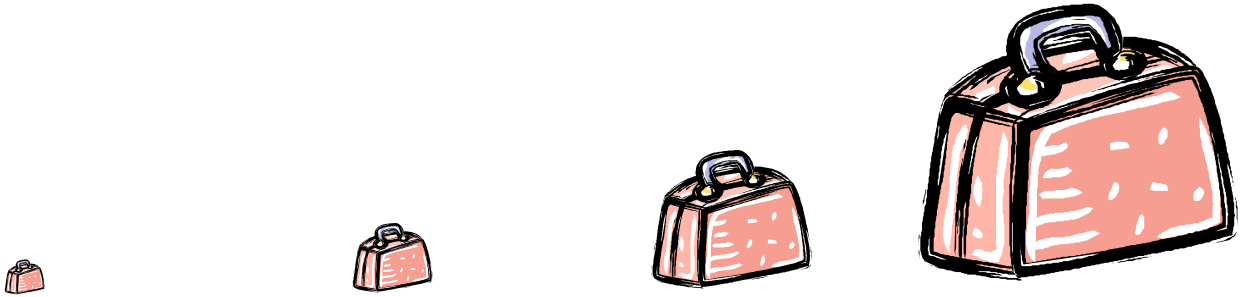
  - Takes up only a small space (8 bits)
  - However, it can only be between **-127** and **127**
- Or an int value

```
int storeInventory;
```

  - Now you can go all the way to **2,147,483,647**
    - Range is approximately **± 2 billion**
  - 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?



`byte` `short` `int` `long`

- Since your computer probably has plenty of space for our programs, **int** and **double** numeric types will be used most often

jGRASP Viewers for byte, short, int, long

File: TypesExample.java C:\Users\crossjh\Documents\courses\comp1210\2011-08-Fall\Web\Lecture_Notes\02\Examples - jGRASP CSD (Java)

File Edit View Build Debug Project Settings Tools Window Help

Threads

Call Stack

[1] TypesExample.main (Type

Variables Eval

Arguments

args --> (obj 379 : java

Locals

b = 15 : byte

s = 15 : short

i = 15 : int

j = 15 : long

a = 15.0 : float

x = 999.0 : float

y = 999.0 : double

z = 0.1 : double

c = 'A' : 65 : char

bn = true : boolean

Browse Debug Find

Workbench

```
public static void main(String[]
{
    byte b = 15;
    short s = 15;
    int i = 15;
    long j = 15;

    float a = 15;
    float x = 999;
    double y = 999;
    double z = 0.1;

    char c = 'A';

    boolean bn = true;
```

TypesExample.java

Compile Messages jGRASP Messages Run I/O Interactions

End

Clear

Help

```
----jGRASP exec: java -
----jGRASP: connected t
```

Status: debugging user program Line:1 Col:1 Code:4

Viewer (by nam...

b

Type byte View

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

Viewer (by name): s

s

Type short View

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

Viewer (by name): i

i

Type int View Detail

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

Viewer (by name): j

j

Type long View Detail

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

TypesExample.java

Character Sets

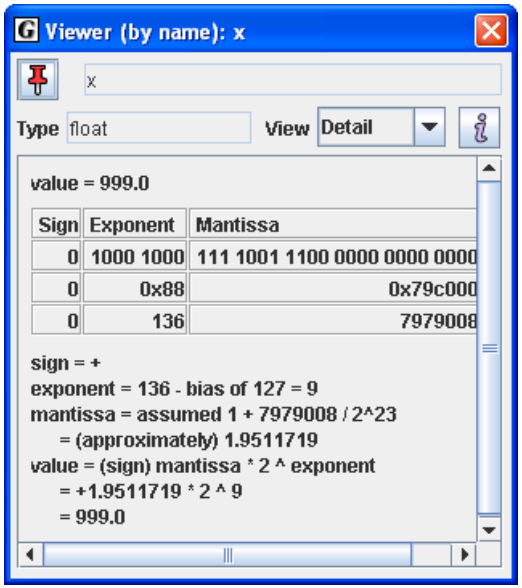
- A *character set* maps each of its characters to a unique number which imposes an order on the characters
  - A `char` variable in Java can store any character from the *Unicode character set*
  - The Unicode character set uses sixteen bits per character, allowing for 65,536 unique characters
  - It is an international character set, containing symbols and characters from many world languages
  - ASCII is (255 characters) is a subset of Unicode

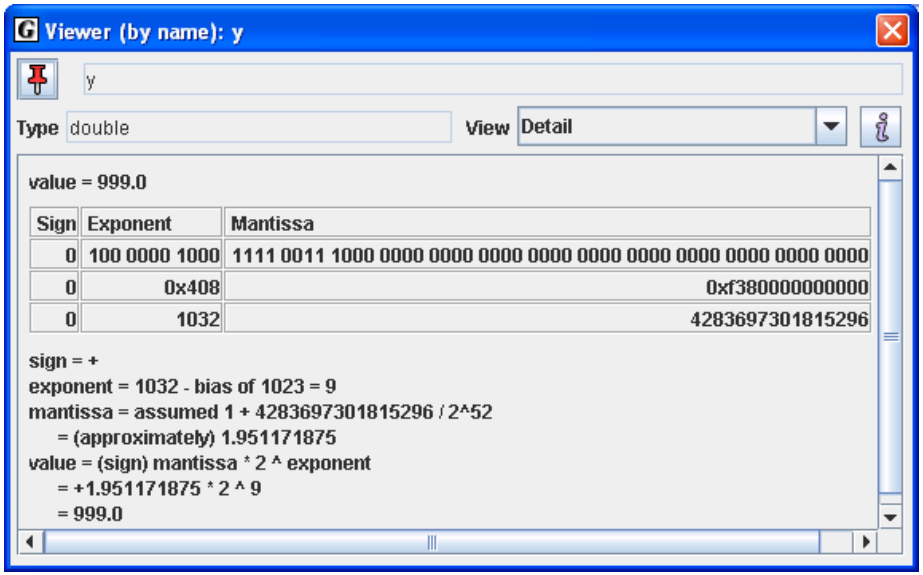
Character Sets

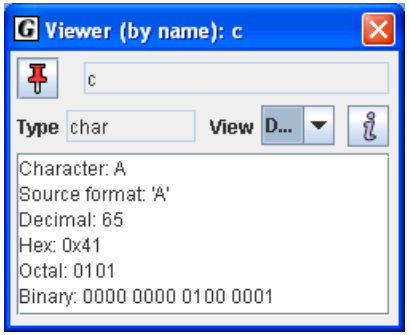
- The ***ASCII*** *character set* is older and smaller than Unicode (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange)
- The ASCII characters are a subset of the Unicode character set, including:

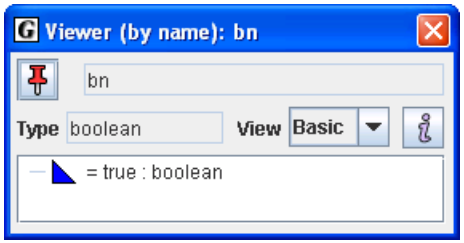
uppercase letters	A, B, C, …
lowercase letters	a, b, c, …
punctuation	period, semi-colon, …
digits	0, 1, 2, …
special symbols	&, |, \, …
control characters	carriage return, tab, ...

jGRSAP Viewers for float, double, char, boolean









Operator Precedence

Q Q

- 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

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

Increment and Decrement

Q

- 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:

Operator	Example	Equivalent To
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y

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 variable on the left; finally this result is assigned to the variable on the left
- Therefore

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

is equivalent to

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

Data Conversion

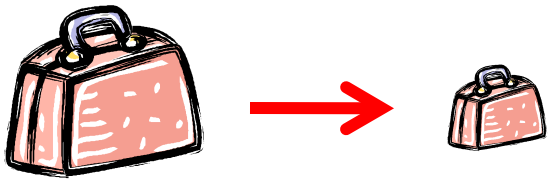
- Sometimes it is convenient to convert data from one type to another
- For example, we may want to treat an integer as a floating point value
- Conversions must be handled carefully to avoid losing information

Data Conversion

- *Widening conversions* go from a smaller to larger data type.
  - 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 as high as 2,147,483,647) ☺
  - If an **int** is converted to a **double**, there is no loss of precision
- *Narrowing conversions* go from a large data type to a smaller one.
  - If the an **int** value was 700 (larger than the max **byte** value of 127), information would be lost when converted to an int
  - If your grade of 89.8 (a **double**) was converted to an **int** type, the new value would be 89 (a 'B'!) ☹

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

Q

- *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 <u>value</u> in `dollars` to a **`double`**

```
money = dollars;
```

- The type and value of `dollars` did not change
- Only widening conversions can occur implicitly during assignment

Data Conversion

- *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;
```

Q Q

Casting

Q

- *Casting* allows narrowing conversions and widening conversions, so be careful!
- It is also easy to detect in code
- To cast, the type is in parentheses <u>in front of the value being converted</u>
- For example, if `total` and `count` are integers, the value of total would be converted to a floating point to avoid integer division:

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

Constants

- A *constant* is similar to a variable, but its initial value cannot be changed
- In Java, we use the `final` modifier to prevent the initial value from changing:

```
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
2. Constants facilitate program maintenance:
   a constant used in multiple places only needs to be updated at its declaration
3. Constants 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

Scanning a String

Q

- 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