

2. Data and Expressions

- Objectives - when we have completed this chapter, 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"

- Every 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
- 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 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 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"`

- 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 is a binary operator (i.e., takes 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 arithmetic 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 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:

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

- Use `\r\n` together to move to the next line
- See `EscapeSeq.java` (page 63)

Variables

- A *variable* is a name for a “location” in memory that holds program data
- There are many types of 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**)
- 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

data 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 right-hand side
 - Store the value in the variable on the left
(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 values:

- byte, short, **int**, long

```
int num1 = -4;
```

- Floating point values:

- float, **double**

```
double num2 = 1.2;
```

- Characters:

- Char

```
char c = 'A';
```

- Boolean values:

- Boolean

```
boolean isCold = false;
```

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 either operand is floating point, then the result is a floating point

Division and Remainder

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

14 / 3 equals 4

8 / 12 equals 0

- The remainder operator (%) returns the remainder after dividing the second operand into the first

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

- 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;  
int temp = 99;  
boolean isCold = temp < 50;
```

If Statements

- Allows a program perform a statement only under certain conditions:

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

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);
```

if-else Statements

- What if you wanted to perform one statement for true condition and a different statement 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?
- 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](#)

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?



byte



short



int



long

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

jGRASP Viewers for byte, short, int, long

The screenshot displays the jGRASP IDE interface. The main window shows a Java file named `TypesExample.java` with the following code:

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

        float x = 999;
        double y = 999;

        char c = 'A';
        boolean bn = true;
    }
}
```

On the left, the **Variables** pane lists the current state of variables:

- static : (TypesExample)
- Arguments
- args[0] : id = 53 : java
- Locals
 - b = 15 : byte
 - s = 15 : short
 - i = 15 : int
 - j = 15 : long
 - x = 999.0 : float
 - y = 999.0 : double
 - c = A = 'A' = 65 : char
 - bn = true : boolean

Four **Viewer** windows are open, each showing the value of a specific variable in different formats (Decimal, Hex, Octal, Binary):

- Viewer (by name): b** (byte): Decimal: 15, Hex: 0xF, Octal: 017, Binary: 0000 1111
- Viewer (by name): s** (short): Decimal: 15, Hex: 0xF, Octal: 017, Binary: 0000 0000 0000 1111
- Viewer (by name): i** (int): Decimal: 15, Hex: 0xF, Octal: 017, Binary: 0000 0000 0000 0000 0000 0000 0000 1111
- Viewer (by name): j** (long): Decimal: 15, Hex: 0xF, Octal: 017, Binary: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111

The bottom status bar indicates the program is in a debugging state: **Status: debugging user program**, with the cursor at **Line:22 Col:9**.

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, ...

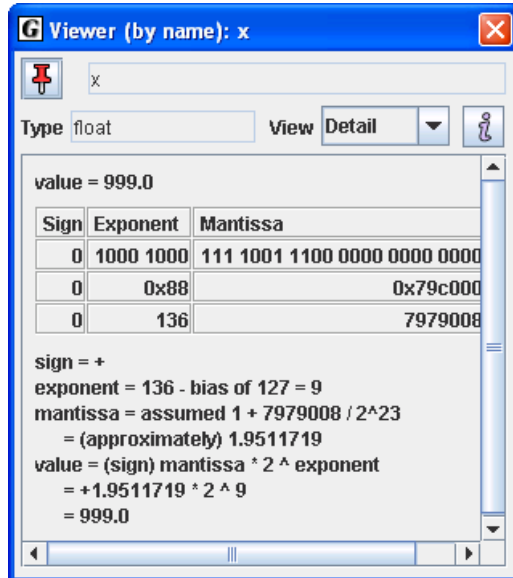
special symbols

&, |, \, ...

control characters

carriage return, tab, ...

jGRSAP Viewers for float, double, char, boolean



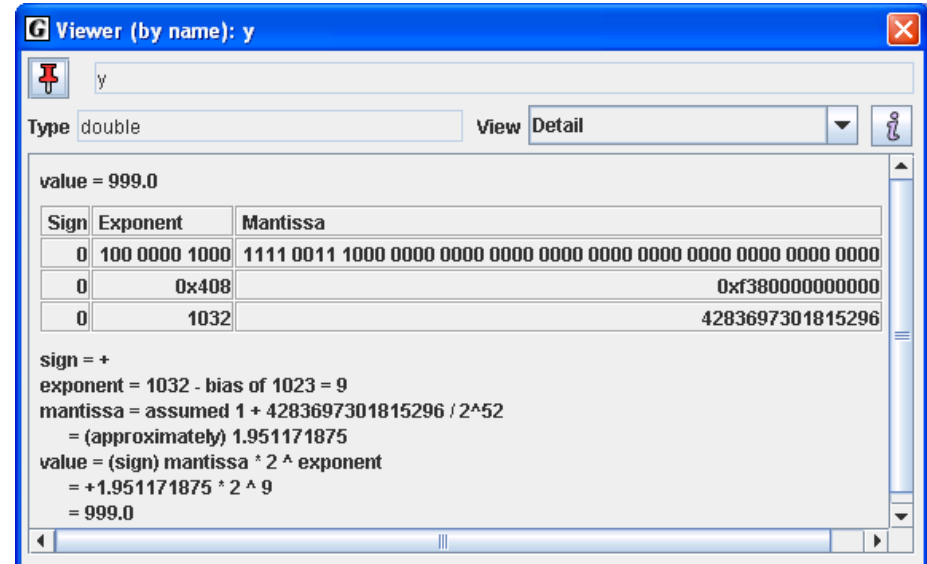
G Viewer (by name): x

Type: float View: Detail

value = 999.0

Sign	Exponent	Mantissa
0	1000 1000	111 1001 1100 0000 0000 0000
0	0x88	0x79c000
0	136	7979008

sign = +
exponent = 136 - bias of 127 = 9
mantissa = assumed 1 + 7979008 / 2²³
= (approximately) 1.9511719
value = (sign) mantissa * 2^{exponent}
= +1.9511719 * 2⁹
= 999.0



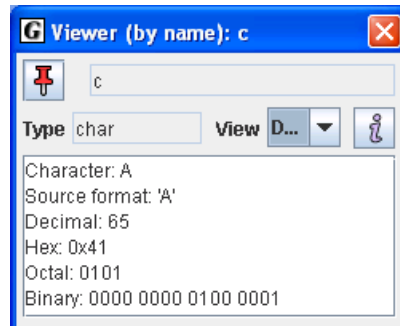
G Viewer (by name): y

Type: double View: Detail

value = 999.0

Sign	Exponent	Mantissa
0	100 0000 1000	1111 0011 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0	0x408	0xf3800000000000
0	1032	4283697301815296

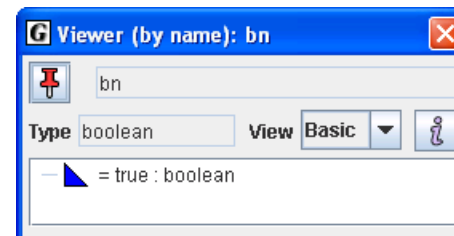
sign = +
exponent = 1032 - bias of 1023 = 9
mantissa = assumed 1 + 4283697301815296 / 2⁵²
= (approximately) 1.951171875
value = (sign) mantissa * 2^{exponent}
= +1.951171875 * 2⁹
= 999.0



G Viewer (by name): c


Type: char View: D...

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

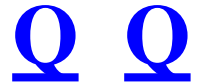


G Viewer (by name): bn

Type: boolean View: Basic

—  = true : boolean

Operator Precedence



- 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



- 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>
+=	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 original variable
- Therefore

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

is equivalent to

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


Data Conversion

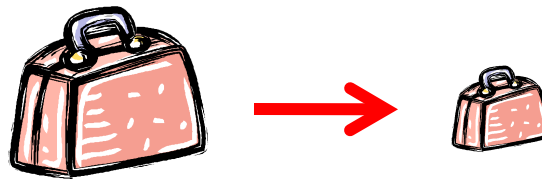
- Sometimes it is convenient to convert data from one type to another
- For example, we may want to treat an integer as a floating point value
- Conversions must be handled carefully to avoid losing information

Data Conversion

- *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 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
 - 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

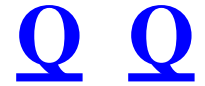
Assignment Conversion

- *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`

```
money = dollars;
```

- Only allows widening conversions
- The value or type of `dollars` did not change

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;
```

Casting

- *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 in front of the value being converted
- 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. 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](#)

Scanning a String

- 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](#)