Chapter 2 Data and Expressions

2.1 Character Strings

A **string** is a primitive Java data type of the type `String`represented by a sequence of characters. `String` is a class, not a primitive.

Java also supports **string literals** - delimited by double quotation characters around the string value: `"this is a string literal."`

Even `""` is a string, despite having zero characters.

Java also supports multi-line strings called text blocks, indicated by three quotation marks `"""`:

System.out.println("""

  The snow is melting

  And the village is flooded

  With children

  """);

The `print` and `println` Methods

The `System.out` class provides methods for printing Strings to the console.

The only difference between `System.out.print` and `System.out.println` are that `println` adds a line break after printing the argument.

String Concatenation

It is possible to **concatenate** (join together) two strings in Java using the `+` operator.

For example: `System.out.println("I wish I was a " + "cat.");`

This must be done to break up strings that would otherwise span multiple lines, which is not supported in Java.

Numeric values that are concatenated to a string value are cast (converted) to String.

Therefor, `"I have" + 3 + "cats."` would convert the integer 3 into a String, outputting a String value: “I have 3 cats.”

It is important to understand the different contextual behaviours of the `+` operator.

- If **either** of the two operands are strings, all numeric values are cast to String before the concatenation.
- If **both** of the two operands are numeric, the output will be an arithmetic addition of the two values.

Escape Sequences

There are a number of characters that cannot be used literally in a String. To overcome this, there are several **escape sequences** to represent special characters:

Escape Sequence	Meaning
\b	backspace
\t	tab
\n	newline
\r	carriage return
\”	double quote
\’	single quote
\\	backslash

2.2 Variables and Assignment

A **variable** is a named location in memory that holds a data value.

When a variable is declared, the compiler reserves a potion of main memory large enough to hold the declared data **type**.

Once declared, you can use a variable's name (identifier) anywhere in your code to access its current value.

Accessing data from a variable this way does not affect it’s value in memory.

Declaration and Assignment Statements

Variables are created using a **declaration statement**

Variables can be declared without being assigned a value using the `=` character:

```
int input;
```

More often though, variables are **initialized** (assigned a value) as they are declared:

```
int fingers = 10
```

A declaration statement has the following components:

1. the data type (ex. int)
2. the identifier (following all identifier syntax requirements)
3. (optional) an initialization: an expression representing the value, preceded by an equals sign.
4. a semicolon.

If a type can be implicitly inferred, the keyword `var` can be used in place of an explicitly defined type.

An **assignment statement** is a statement that assigns a value to an existing variable, overwriting the previous value. For example:

int x = 10; //declaration and initialization

x = 20; //assignment statement

Java is **strongly typed**, meaning variables must be declared with a specific type, and can only hold values of that type. This helps prevent errors by ensuring type compatibility at compile time.

Constants

Constants are special variables that cannot be changed once initialized.

Constants are declared by adding the modifier `static final` :

```
final int MEANING_OF_EVERYTHING = 42;
```

A best-practice convention is to use upper case letters for constant names to distinguish them from regular variables.

Reasons to use constants:

- Giving meaning to otherwise unclear literal values.
    - Example: `MAX_LOAD` means more than the literal 250
- Facilitating program maintenance (change one number instead of hunting down instances of it throughout the program)
- Prevent accidental change at runtime.

2.3 Primitive Data Types

Java has 8 primitive data types: 2 types of floats, 4 types of integers, a character data type and a boolean data type. Everything else is represented using objects.

Integers have no fractional value, whereas **floating points** do. The numeric primitives are as follows:

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	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
`float`	32 bits	Approximately -3.4E+38 with 7 significant digits	Approximately 3.4E+38 with 7 significant digits
`double`	64 bits	Approximately -1.7E+308 with 15 significant digits	Approximately 1.7E+308 with 15 significant digits

Integers that may need more space than a long can use the BigInteger class.

Numeric Literals

numbers expressed as an integer (no decimal) are always of type `int` unless an `L` or `l` is appended to the end, which makes it type `long`

likewise, all floating point literals are of type `double` unless an `F` or `f` is appended, making it a `float`

hexidecimal numbers have the prefix `x` or `X`

octal numbers have the prefix `0` (avoid this, it is confusing)

binary numbers have the prefix `0b` or `0B`

Characters

Characters can be expressed as literals in Java with `'` single quotes, such as `a` , `J` or `;` .

In Java the primitive data type `char` represents a single character.

Character Sets:

The most common character set (defining the range of characters available) is called ASCII (American Standard Code for Information Interchange).

Because ASCII only contains a limited set of 128 characters, Java uses Unicode encoding (UTF-16) which can represent over 1 million characters in the full Unicode character set. The first 128 code points of Unicode are identical to ASCII characters.

Booleans

A boolean can have only two values: `true` or `false` .

In Java, they are defined using the reserved word `boolean`

booleans have no numeric equivalent and cannot be converted two or from any other data type

2.4 Expressions

An **expression** is a combination of one or more operators and operands that usually perform a calculation.

Arithmetic Operators

The common `+` `-` `*` and `/` operators are defined for both integers and floating point numeric types.

The type of the operands of these operations affects the return value:

- If *either* operand is a floating-point type, the return value will also be a floating-point value
- if *both* operands are integers, the return value will also be an integer, and any decimal value will be lost.

The remainder operator `%` returns the remainder after dividing the second operand into the first.

The sign of the result of a remainder operation is the sign of the numerator.

Java does not have a mod operator

Operator Precedence

The order of operations followed when evaluating Java expressions are similar to those learned in algebra. Operators of the same precedence are performed left-to-right. The precedence is as follows:

- Any expression in parentheses.
- Casting
- Multiplication, division and remainder
- Addition and subtraction
- Assignment (ie storing the value calculated by the expression)

Variable Self-Assignment

Variables can appear on both sides of an assignment statement. This is often used for incrementing/changing variable values. ex:

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

The right side evaluates first (count + 1), and then the result overwrites original value.

Increment and Decrement Operators

The operators `++` and `--` simply add or subtract 1 to a numeric value. ex:

```
count++;
```

this is functionally equivalent to the statement above (`count = count + 1;`)

Increment/decrement operators can be used in two forms:

- **Postfix form**: operator after variable (`count++`, `count--`)
- **Prefix form**: operator before variable (`++count`, `--count`)

Both forms behave the same on their own, but behave differently in expressions:

- **Postfix**: Original value used first, then incremented
- **Prefix**: Increments first, then uses new value

// Postfix example (count is 15)

total = count++; // total = 15, count becomes 16

// Prefix example (count is 15)

total = ++count; // total = 16, count becomes 16

Assignment Operators

As a convenience, several assignment operators have been defined in Java that combine a basic operation with assignment.

For example:

total += 5;

This is equivalent to writing out the full expression `total = total + 5`

Assignment operators exist for all the previously mentioned arithmetic operators:

`+=` `-=` `*=` `/=` and `%=`

2.5 Data Conversions

Because Java is strongly typed, converting between types must be done carefully to avoid data loss. The following table shows valid type conversions:

From	To (Widening)	To (Narrowing)
byte	short, int, long, float, double	char
short	int, long, float, double	byte, char
char	int, long, float, double	byte, short

int	long, float, double	byte, short, char
long	float, double	byte, short, char, int
float	double	byte, short, char, int, long
double	none	byte, short, char, int, long, float

Narrowing Conversions

- Converts to a type with less storage space
- Risky - can lose information
- Java does not do these implicitly
- Example: short → byte may lose data if value too large
- Special case: byte/short → char is narrowing due to sign bit

Conversion Techniques

In Java, conversions can occur in three ways:

- assignment conversion
- promotion
- casting

Assignment conversion: the automatic conversion that occurs when assigning values of one type to a variable of another.

Assignment conversion only works for widening conversions. Attempting to assign a float value to an int variable will result in a compiler error.

Promotion: an automatic conversion that converts operands to compatible types during arithmetic operations

For example: When dividing float by int, the int is first promoted to float

Casting: A cast is a Java operator that specifies a type name in parentheses. ex: `(type)value`

When casting a floating point value to an int, the fractional value is truncated. Ex:

```
System.out.print((int) Math.Pi); //output:3
```

Note that casting does not change the variable being cast. It is a temporary type conversion in expressions

2.6 Interactive Programs

Programs can read user input during execution to compute new results each time they run.

The `Scanner` Class

The Scanner class is part of the Java API and provides methods for reading different types of input values, such as the keyboard or external files.

Creating a Scanner Object

To use Scanner, create an object using:

Scanner scan = new Scanner(System.in);

This expression uses the `new` operator to create a new object, using a call to a special method called a **constructor.**

The constructor accepts an input as a parameter. In this case, `System.in` represents the input stream (typically keyboard input)

Key Scanner Methods

Unless otherwise specified, a `Scanner` uses whitespace (spaces, tabs, newlines) as default delimiters between elements in the input (aka tokens)

- `next()`: Reads next token as string
- `nextLine()`: Reads the next entire line as string
- `nextInt()`: Reads the next integer value
- `nextDouble()`: Reads the next decimal value

A **token** is the stuff in between white space in a string.

White space is defined as a space, tab or newline

The following program prompts the user to enter a line of text, and then prints it back at them:

```
import java.util.Scanner;


public class Echo {

   // Reads a character string from the user and prints it

   public static void main(String[] args) {
```

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


        System.out.println("Enter a line of text:");
        message = scan.nextLine();
        System.out.println("You entered: \\"" + message + "\\"");
    }
}
```

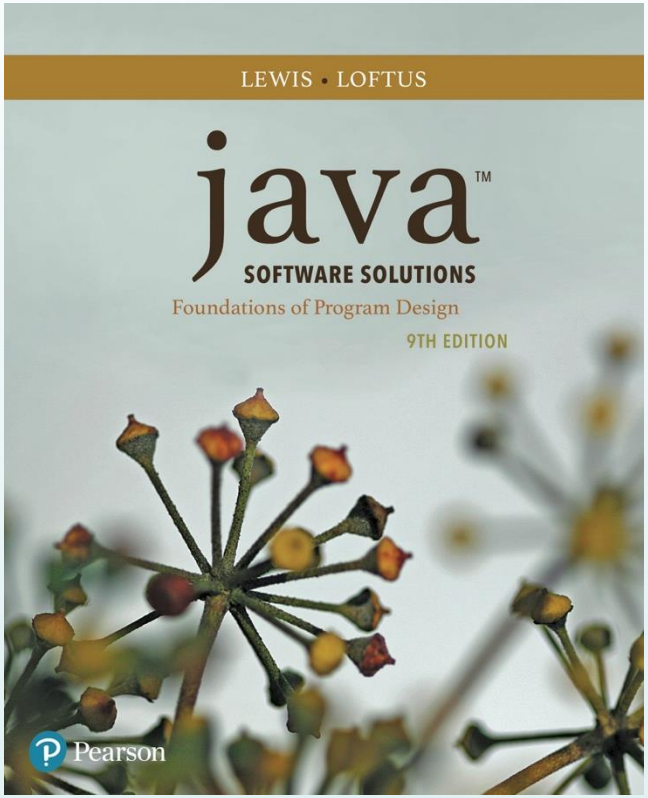
output (user input in red):

Enter a line of text: meow You entered: "meow"

Note: To use the Scanner class, we must first **import** it with the following line of code at the top of the file:

```
import java.util.Scanner;
```

Chapter 2
Data and Expressions



Java Software Solutions

Foundations of Program Design

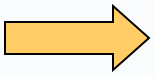
9th Edition

John Lewis
William Loftus

Data and Expressions

- Let's explore some other fundamental programming concepts
- Chapter 2 focuses on:
  - character strings
  - primitive data
  - the declaration and use of variables
  - expressions and operator precedence
  - data conversions
  - accepting input from the user

Outline



Character Strings

Variables and Assignment

Primitive Data Types

Expressions

Data Conversion

Interactive Programs

Character Strings

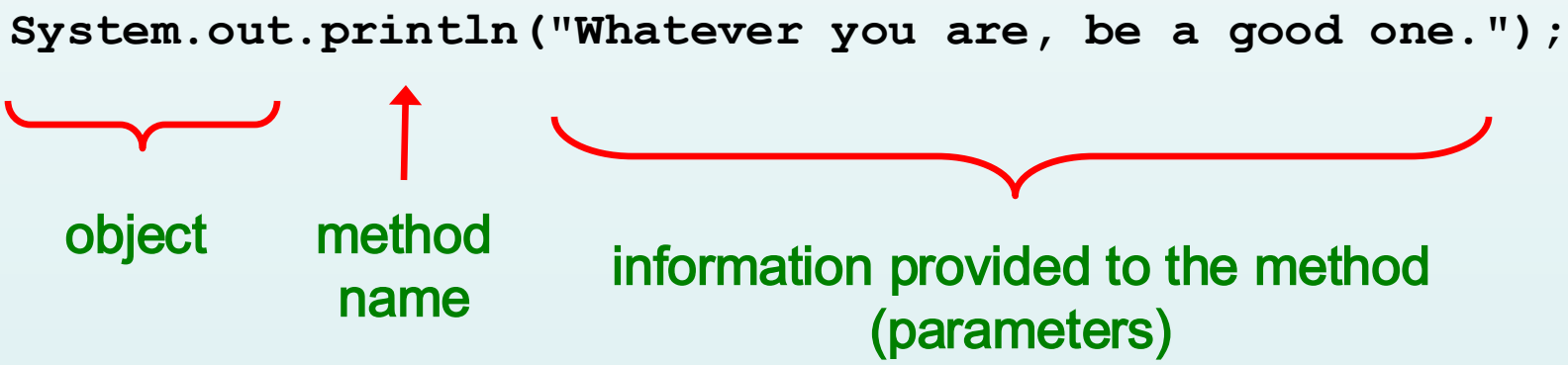
- A *string literal* is represented by putting double quotes around the text
- Examples:

```
"This is a string literal."
"123 Main Street"
"X"
```

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

The println Method

- In the `Lincoln` program from Chapter 1, we invoked the `println` method to print a character string
- The `System.out` object represents a destination (the monitor screen) to which we can send output



The print Method

- The `System.out` object provides another service as well
- The `print` method 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 `Countdown.java`

String Concatenation

- The *string concatenation operator* (+) is used to append one string to the end of another

```
"Peanut butter " + "and jelly"
```

- It can also be used to append a number to a string
- A string literal cannot be broken across two lines in a program
- See `Facts.java`

String Concatenation

- The + operator is also used for arithmetic addition
- The function that it performs depends on the type of the information on which it operates
- If both operands are strings, or if one is a string and one is a number, it performs string concatenation
- If both operands are numeric, it adds them
- The + operator is evaluated left to right, but parentheses can be used to force the order
- See `Addition.java`

Quick Check

What output is produced by the following?

```
System.out.println("X: " + 25);
System.out.println("Y: " + (15 + 50));
System.out.println("Z: " + 300 + 50);
```

Quick Check

What output is produced by the following?

```
System.out.println("X: " + 25);
System.out.println("Y: " + (15 + 50));
System.out.println("Z: " + 300 + 50);
```

```
X: 25
Y: 65
Z: 30050
```

Escape Sequences

- What if we wanted to print the quote character?
- The following line would confuse the compiler because it would interpret the second quote as the end of the string

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

- An *escape sequence* is a series of characters that 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
`\r`	carriage return
`\"`	double quote
`\'`	single quote
`\\`	backslash

- See `Roses.java`

Quick Check

Write a single `println` statement that produces the following output:

"Thank you all for coming to my home
tonight," he said mysteriously.

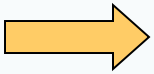
Quick Check

Write a single `println` statement that produces the following output:

"Thank you all for coming to my home
tonight," he said mysteriously.

```
System.out.println("\"Thank you all for " +
   "coming to my home\ntonight,\" he said " +
   "mysteriously.");
```

Outline



Character Strings

Variables and Assignment

Primitive Data Types

Expressions

Data Conversion

Interactive Programs

Variables

- A *variable* is a name for a location in memory that holds a value
- A *variable declaration* specifies the variable's name and the type of information that it will hold


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

Multiple variables can be created in one declaration

Variable Initialization

- A variable can be given an initial value in the declaration

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

- When a variable is referenced in a program, its current value is used
- See `PianoKeys.java`
- If the compiler can infer the variable's type from the initial value, can use var in place of the type

```
var sum = 0;
var base = 32;
var max = 149;
```

Assignment

- An *assignment statement* changes the value of a variable
- The assignment operator is the = sign

```
total = 55;
```

- The value that was in `total` is overwritten
- You can only assign a value to a variable that is consistent with the variable's declared type
- See `Geometry.java`

Constants

- A *constant* is an identifier that is similar to a variable except that it holds the same value during its entire existence
- As the name implies, it is constant, not variable
- The compiler will issue an error if you try to change the value of a constant
- In Java, we use the `static final` modifier to declare a constant

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

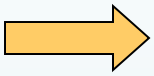
Constants

- Constants are useful for three important reasons
- First, they give meaning to otherwise unclear literal values
  - Example: `MAX_LOAD` means more than the literal 250
- Second, they facilitate program maintenance
  - If a constant is used in multiple places, its value need only be set in one place
- Third, they formally establish that a value should not change, avoiding inadvertent errors by other programmers

Outline

Character Strings

Variables and Assignment



Primitive Data Types

Expressions

Data Conversion

Interactive Programs

Primitive Data

- There are eight primitive data types in Java
- Four of them represent integers:
  - `byte`, `short`, `int`, `long`
- Two of them represent floating point numbers:
  - `float`, `double`
- One of them represents characters:
  - `char`
- And one of them represents boolean values:
  - `boolean`

Numeric Primitive Data

- The difference between the numeric primitive types is their size and the values they can store:

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	

Numerical Literals

- Numbers consisting of an optional sign followed by digits have type int
  - use a suffix of L or l for type long
- Numbers with a decimal point have type double
  - use a suffix of F or f for type float
- Hexadecimal numbers have prefix 0x or 0X
  - Octal numbers have prefix 0 (confusing, avoid)
  - Binary numbers have prefix 0b or 0B
- Floating point numbers can use scientific notation using E or e followed by the exponent
- Underscore _ can be used in the middle of a number as a separator for readability

Characters

- A `char` variable stores a single character
- Character literals are delimited by single quotes:

```
'a'   'X'   '7'   '$'   ','   '\n'
```

- Example declarations:

```
char topGrade = 'A';
char terminator = ';', separator = ' ';
```

- Note the difference between a primitive character variable, which holds only one character, and a `String` object, which can hold multiple characters

Character Sets

- A *character set* is an ordered list of characters, with each character corresponding to 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

Characters

- 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, ...

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;
```

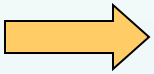
- A `boolean` variable can also be used to represent any two states, such as a light bulb being on or off

Outline

Character Strings

Variables and Assignment

Primitive Data Types



Expressions

Data Conversion

Interactive Programs

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 or both operands are floating point values, 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)

```
14 / 3   equals   4
8 / 12   equals   0
```

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

```
14 % 3   equals   2
8 % 12   equals   8
```

Division, Remainder and Negatives

- Integer division always truncates towards zero

  -14 / 3 equals -4 14 / -3 equals -4

  -14 / -3 equals 4

  -8 / 12 equals 0 8 / -12 equals 0

- Integer remainder follows the rule a = b * q + r
  - where q = a / b and r = a % b
- So the sign of `a % b` is the sign of `a`

  -14 % 3 equals -2 14 % -3 equals 2

  -14 % -3 equals -2

  -8 % 12 equals -8 8 % -12 equals 8

- Java does not have a mod operator

Quick Check

What are the results of the following expressions?

```
    12 / 2
12.0 / 2.0
    10 / 4
  10 / 4.0
    4 / 10
  4.0 / 10
    12 % 3
    10 % 3
    3 % 10
```

Quick Check

What are the results of the following expressions?

```
   12 / 2  =  6
12.0 / 2.0 =  6.0
   10 / 4  =  2
  10 / 4.0 =  2.5
   4 / 10  =  0
  4.0 / 10 =  0.4
   12 % 3  =  0
   10 % 3  =  1
   3 % 10  =  3
```

Operator Precedence

- Operators can be combined into larger expressions

```
result  =  total + count / max - offset;
```

- Operators have a well-defined 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

Quick Check

In what order are the operators evaluated in the following expressions?

```
a + b + c + d + e
```

```
a + b * c - d / e
```

```
a / (b + c) - d % e
```

```
a / (b * (c + (d - e)))
```

Quick Check

In what order are the operators evaluated in the following expressions?

```
a + b + c + d + e
  1   2   3   4
```

```
a + b * c - d / e
  3   1   4   2
```

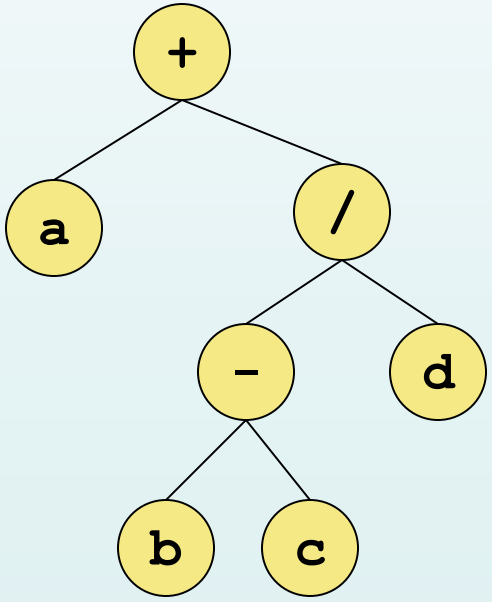
```
a / (b + c) - d % e
  2    1    4   3
```

```
a / (b * (c + (d - e)))
  4    3    2    1
```

Expression Trees

- The evaluation of a particular expression can be shown using an *expression tree*
- The operators lower in the tree have higher precedence for that expression

```
a + (b - c) / d
```



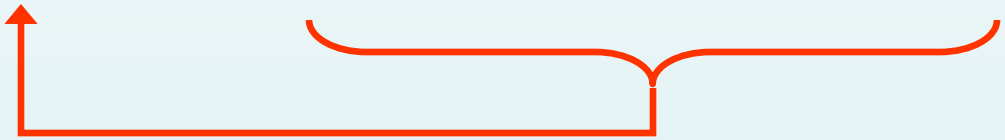
Assignment Revisited

- 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

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 statement

```
count++;
```

  is functionally equivalent to

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

Increment and Decrement

- The increment and decrement operators can be applied in *postfix form*:

```
count++
```

- or *prefix form*:

```
++count
```

- When used as part of a larger expression, the two forms can have different effects
- Because of their subtleties, the increment and decrement operators should be used with care

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;
```

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 original variable
- Therefore

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

is equivalent to

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

Assignment Operators

- The behavior of some assignment operators depends on the types of the operands
- If the operands to the += operator are strings, the assignment operator performs string concatenation
- The behavior of an assignment operator (+=) is always consistent with the behavior of the corresponding operator (+)
- The result is converted to the type of the LHS before assignment – even if narrowing is required

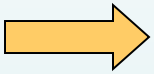
Outline

Character Strings

Variables and Assignment

Primitive Data Types

Expressions



Data Conversion

Interactive Programs

Data Conversion

- Sometimes it is convenient to convert data from one type to another
- For example, in a particular situation we may want to treat an integer as a floating point value
- These conversions do not change the type of a variable or the value that's stored in it – they only convert a value as part of a computation

Data Conversion

- *Widening conversions* are safest because they tend to go from a small data type to a larger one (such as a `short` to an `int`)
- *Narrowing conversions* can lose information because they tend to go from a large data type to a smaller one (such as an `int` to a `short`)
- In Java, data conversions can occur in three ways:
  - assignment conversion
  - promotion
  - casting

Data Conversion

Widening Conversions

From	To
byte	short, int, long, float, or double
short	int, long, float, or double
char	int, long, float, or double
int	long, float, or double
long	float or double
float	double

Narrowing Conversions

From	To
byte	char
short	byte or char
char	byte or short
int	byte, short, or char
long	byte, short, char, or int
float	byte, short, char, int, or long
double	byte, short, char, int, long, or float

Assignment Conversion

- *Assignment conversion* occurs when a value of one type is assigned to a variable of another
- Example:

```
int dollars = 20;
double money = dollars;
```

- Only widening conversions can happen via assignment
- Note that the value or type of `dollars` did not change

Promotion

- *Promotion* happens automatically when operators in expressions convert their operands
- Example:

```
int count = 12;
double sum = 490.27;
result = sum / count;
```

- The value of `count` is converted to a floating point value to perform the division calculation

Casting

- *Casting* is the most powerful, and dangerous, technique for conversion
- Both widening and narrowing conversions can be accomplished by explicitly casting a value
- To cast, the type is put in parentheses in front of the value being converted

```
int total = 50;
float result = (float) total / 6;
```

- Without the cast, the fractional part of the answer would be lost

Outline

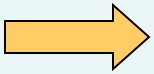
Character Strings

Variables and Assignment

Primitive Data Types

Expressions

Data Conversion



Interactive Programs

Interactive Programs

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

Reading Input

- 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 invoke various input methods, such as:

```
answer = scan.nextLine();
```

Reading Input

- The `Scanner` class is part of the `java.util` class library, and must be imported into a program to be used
- The `nextLine` method reads all of the input until the end of the line is found
- See `Echo.java`
- The details of 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 `GasMileage.java`

Summary

- Chapter 2 focused on:
  - character strings
  - primitive data
  - the declaration and use of variables
  - expressions and operator precedence
  - data conversions
  - accepting input from the user