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;
```