



D06

PROGRAMMING with JAVA

Ch4 – Fundamental Data Types

PowerPoint presentation, created by Angel A. Juan - ajuanp@gmail.com,
for accompanying the book "Big Java", by Cay S. Horstmann (2006)

Chapter Goals

- To understand **integer** and **floating-point** numbers
- To recognize the limitations of the numeric types
- To become aware of causes for **overflow** and **roundoff** errors
- To understand the proper use of **constants**
- To write arithmetic expressions in Java
- To use the **String** type to define and manipulate character strings
- To learn how to read program input and produce formatted output

Number Types



In Java, every value is either a **reference to an object**, or it belongs to one of the 8 **primitive types**: `int`, `byte`, `short`, `long`, `double`, `float`, `char`, `boolean`

- **int**: integers, no fractional part

`1, -4, 0`

- **long**: very long integers

`-9E18 ... 9E18`

- **double**: floating-point numbers (double precision)

`0.5, -3.11111, 4.3E24, 1E-14`

- **char**: character type

`'a', '$', '@', 'Z'`

- **boolean**: two truth values, `false` and `true`

- A numeric computation **overflows** if the result falls outside the range for the number type

```
int n = 1000000;  
System.out.println(n * n); // prints -727379968
```

Primitive Types

| Type | Description | Size |
|-------|---|------------|
| int | The integer type, with range -2,147,483,648 . . . 2,147,483,647 | 4 bytes |
| byte | The type describing a single byte, with range -128 . . . 127 | 1 byte |
| short | The short integer type, with range -32768 . . . 32767 | 2 bytes |
| long | The long integer type, with range – 9,223,372,036,854,775,808 . . . -9,223,372,036,854,775,807 | 8 bytes |

| Type | Description | Size |
|---------|--|------------|
| double | The double-precision floating-point type, with a range of about $\pm 10^{308}$ and about 15 significant decimal digits | 8 bytes |
| float | The single-precision floating-point type, with a range of about $\pm 10^{38}$ and about 7 significant decimal digits | 4 bytes |
| char | The character type, representing code units in the Unicode encoding scheme | 2 bytes |
| boolean | The type with the two truth values <code>false</code> and <code>true</code> | 1 byte |

Number Types: Floating-point Types

- **Rounding errors** occur when an exact conversion between numbers is not possible:

```
double f = 4.35;  
System.out.println(100 * f); // prints 434.99999999999994
```

- The **double** type is not appropriate for financial calculations (you should use **BigDecimal** type instead)



In Java it is legal to assign an integer value to a floating-point variable, but you can not assign a floating-point expression to an integer variable:

```
double balance = 13.75;  
int dollars = balance; // Error
```

Continued...

Syntax 4.1: Cast

`(typeName) expression`

Example:

`(int) (balance * 100)`

Purpose:

To convert an expression to a different type

Big Number Objects

- If you want to compute with really large numbers, you can use **big number objects**, which are objects of the `BigInteger` and `BigDecimal` classes in the `java.math` package
- Big number objects have essentially no limits on their size and precision (but computations with them are slower)
- You can't use the familiar arithmetic operators (+, -, *, etc.) with big number objects. Instead, you have to use methods called `add()`, `subtract()`, `multiply()`, etc.:

```
BigInteger a = new BigInteger("1234567890");  
BigInteger b = new BigInteger("9876543210");  
BigInteger c = a.multiply(b);  
System.out.println(c); // Prints 1219326311126352900
```



The `BigDecimal` type carries out floating-point computation without roundoff errors

Self Check

1. Which are the most commonly used number types in Java?
2. When does the cast `(long) x` yield a different result from the call `Math.round(x)`?
3. How do you round the `double` value `x` to the nearest `int` value, assuming that you know that it is less than $2 \cdot 10^9$?

Answers

1. `int` and `double`
2. When the fractional part of `x` is ≥ 0.5
3. By using a cast: `(int) Math.round(x)`

Constants: `final`

- A **constant** is a variable which value cannot be changed once it has been set



In Java, constants are identified with the keyword **`final`**

- **Named constants** make programs easier to read and maintain (do not use **literal constants** like `3.1416` through your code, use named constants instead)
- **Convention:** use all-uppercase names for constants

```
final double QUARTER_VALUE = 0.25;
final double DIME_VALUE = 0.1;
final double NICKEL_VALUE = 0.05;
final double PENNY_VALUE = 0.01;
payment = dollars + quarters * QUARTER_VALUE + dimes * DIME_VALUE
        + nickels * NICKEL_VALUE + pennies * PENNY_VALUE;
```

Constants: public static final



If constant values are needed in several methods, declare them together with the instance fields of a class and tag them as

static and **final**

`static` means that the constant belongs to the class

Don't worry, constants can not be modified

`final` indicates that the value is a constant

- Give **static final** constants **public access** to enable other classes to use them. Methods of other classes can access a public constant by first specifying the name of the class in which it is defined, then a period (.), then its name:

```
public class Math
{
    . . .
    public static final double E = 2.7182818284590452354;
    public static final double PI = 3.14159265358979323846;
}

double circumference = Math.PI * diameter;
```

Syntax 4.2: Constant Definition

In a method:

```
final typeName variableName = expression ;
```

In a class:

```
accessSpecifier static final typeName variableName = expression;
```

Example:

```
final double NICKEL_VALUE = 0.05;  
public static final double LITERS_PER_GALLON = 3.785;
```

Purpose:

To define a constant in a method or a class

File CashRegister.java

```
01: /**
02:     A cash register totals up sales and computes change due.
03: */
04: public class CashRegister
05: {
06:     /**
07:         Constructs a cash register with no money in it.
08:     */
09:     public CashRegister()
10:     {
11:         purchase = 0;
12:         payment = 0;
13:     }
14:
```

Continued...

File CashRegister.java

```
15:    /**
16:        Records the purchase price of an item.
17:        @param amount the price of the purchased item
18:    */
19:    public void recordPurchase(double amount)
20:    {
21:        purchase = purchase + amount;
22:    }
23:
24:    /**
25:        Enters the payment received from the customer.
26:        @param dollars the number of dollars in the payment
27:        @param quarters the number of quarters in the payment
28:        @param dimes the number of dimes in the payment
29:        @param nickels the number of nickels in the payment
30:        @param pennies the number of pennies in the payment
31:    */
```

Continued...

File CashRegister.java

```
32:     public void enterPayment(int dollars, int quarters,
33:         int dimes, int nickels, int pennies)
34:     {
35:         payment = dollars + quarters * QUARTER_VALUE
36:             + dimes * DIME_VALUE
37:             + nickels * NICKEL_VALUE + pennies
38:             * PENNY_VALUE;
39:     }
40:     /**
41:      * Computes the change due and resets the machine for
42:      * the next customer.
43:      * @return the change due to the customer
44:      */
```

Continued...

File CashRegister.java

```
43:     public double giveChange()
44:     {
45:         double change = payment - purchase;
46:         purchase = 0;
47:         payment = 0;
48:         return change;
49:     }
50:
51:     public static final double QUARTER_VALUE = 0.25;
52:     public static final double DIME_VALUE = 0.1;
53:     public static final double NICKEL_VALUE = 0.05;
54:     public static final double PENNY_VALUE = 0.01;
55:
56:     private double purchase;
57:     private double payment;
58: }
```

File CashRegisterTester.java

```
01: /**
02:     This class tests the CashRegister class.
03: */
04: public class CashRegisterTester
05: {
06:     public static void main(String[] args)
07:     {
08:         CashRegister register = new CashRegister();
09:
10:         register.recordPurchase(0.75);
11:         register.recordPurchase(1.50);
12:         register.enterPayment(2, 0, 5, 0, 0);
13:         System.out.print("Change=");
14:         System.out.println(register.giveChange());
15:
```

Continued...

File CashRegisterTester.java

```
16:         register.recordPurchase(2.25);
17:         register.recordPurchase(19.25);
18:         register.enterPayment(23, 2, 0, 0, 0);
19:         System.out.print("Change=");
20:         System.out.println(register.giveChange());
21:     }
22: }
```

Output

```
Change=0.25
Change=2.0
```

Self Check

4. What is the difference between the following two statements?

```
final double CM_PER_INCH = 2.54;
```

and

```
public static final double CM_PER_INCH = 2.54;
```

5. What is wrong with the following statement?

```
double circumference = 3.14 * diameter;
```

Answers

4. The first definition is used inside a method, the second inside a class
5. (1) You should use a named constant, not the "magic number" 3.14
(2) 3.14 is not an accurate representation of π

Assignment, Increment, and Decrement

- **Assignment** is not the same as mathematical equality:

```
items = items + 1;
```

- `items++` is the same as `items = items + 1`
- `items--` is the same as `items = items - 1`

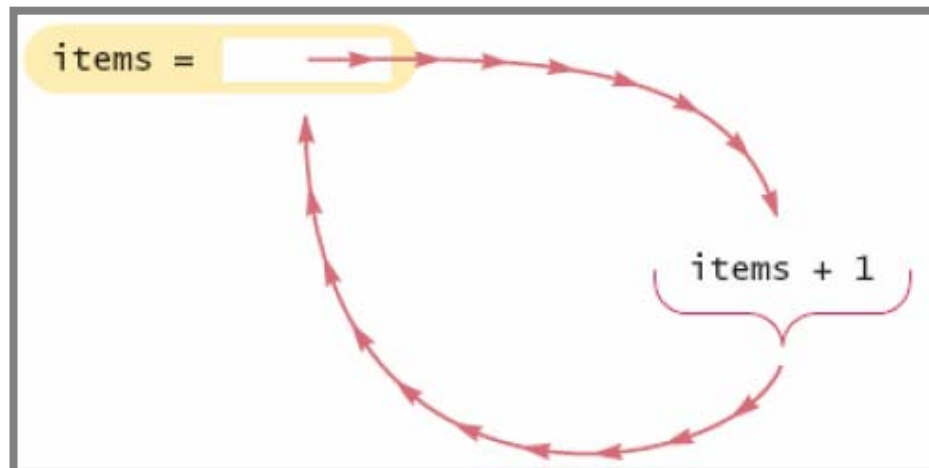


Figure 1:
Incrementing a Variable

Self Check

6. What is the meaning of the following statement?

```
balance = balance + amount;
```

7. What is the value of `n` after the following sequence of statements?

```
n--;
```

```
n++;
```

```
n--;
```

Answers

6. The statement adds the `amount` value to the `balance` variable
7. One less than it was before

Arithmetic Operations

- / is the **division** operator



If both arguments are integers, the result is always an integer, with the remainder discarded:

7.0 / 4 yields 1.75 while 7 / 4 yields 1

- Get the remainder with **% (modulo)**

7 % 4 is 3

```
final int PENNIES_PER_NICKEL = 5;
final int PENNIES_PER_DIME = 10;
final int PENNIES_PER_QUARTER = 25;
final int PENNIES_PER_DOLLAR = 100;
// Compute total value in pennies
int total = dollars * PENNIES_PER_DOLLAR + quarters
    * PENNIES_PER_QUARTER
+ nickels * PENNIES_PER_NICKEL + dimes * PENNIES_PER_DIME
    + pennies;
// Use integer division to convert to dollars, cents
int dollars = total / PENNIES_PER_DOLLAR;
int cents = total % PENNIES_PER_DOLLAR;
```

The Math class

- **Math class:** contains methods like `sqrt()` and `pow()`
- To compute x^n , you write `Math.pow(x, n)`
- However, to compute x^2 it is significantly more efficient simply to compute `x * x`
- To take the square root of a number, use the `Math.sqrt()` method; for example, `Math.sqrt(x)`

- In Java,

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

can be represented as:

```
(-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a)
```

Mathematical Methods in Java

| | |
|---|--|
| <code>Math.sqrt(x)</code> | square root |
| <code>Math.pow(x, y)</code> | power x^y |
| <code>Math.exp(x)</code> | e^x |
| <code>Math.log(x)</code> | natural log |
| <code>Math.sin(x)</code> , <code>Math.cos(x)</code> , <code>Math.tan(x)</code> | sine, cosine, tangent (x in radian) |
| <code>Math.round(x)</code> | closest integer to x |
| <code>Math.min(x, y)</code> , <code>Math.max(x, y)</code> | minimum, maximum |
| <code>Math.ceil(x)</code> , <code>Math.floor(x)</code> | smallest integer $\geq x$, largest integer $\leq x$ |
| <code>Math.abs(x)</code> | absolute value of x |
| ... | ... |

Self Check

8. What is the value of $1729 / 100$?
Of $1729 \% 100$?

9. Why doesn't the following statement compute the average of $s1$, $s2$, and $s3$?

```
double average = s1 + s2 + s3 / 3; // Error
```

10. What is the value of

```
Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2))
```

in mathematical notation?

Answers

8. 17 and 29

9. Only s_3 is divided by 3. To get the correct result, use parentheses. Moreover, if s_1 , s_2 , and s_3 are integers, you must divide by 3.0 to avoid integer division:

$$(s_1 + s_2 + s_3) / 3.0$$

10.

$$\sqrt{x^2 + y^2}$$

Calling Static Methods

- The `Math` class contains a collection of `static` methods. `static` methods do not operate on an object:

```
double x = 4;  
double root = x.sqrt(); // Error
```



In Java, numbers are not objects, so you can never invoke a method on a number. Instead, you pass a number as an explicit parameter to a method:

```
Math.sqrt(x)
```

(this call makes it appear as if the `sqrt()` method is applied to an object, `Math`. However, `Math` is a class, not an object)

- `static` methods do not operate on objects, but they are still defined inside classes



Naming convention: Classes start with an uppercase letter; objects and methods start with a lowercase letter

```
System      out      println()
```

Syntax 4.3: Static Method Call

ClassName. methodName(parameters)

Example:

`Math.sqrt(4)`

Purpose:

To invoke a static method (a method that does not operate on an object) and supply its parameters

Self Check

11. Why can't you call `x.pow(y)` to compute x^y ?
12. Is the call `System.out.println(4)` a static method call?

Answers

11. `x` is a number, not an object, and you cannot invoke methods on numbers
12. No—the `println` method is called on the object `System.out`

Strings

- A **string** is a sequence of characters
- Unlike numbers, strings are objects (of the `String` class)
- String constants: `"Hello, World!"`
- String variables: `String message = "Hello, World!";`
- The number of characters in a string is called the **string length**. You can compute the length of a string with the **`length()`** method: `int n = message.length();`
- A string of length zero, containing no characters, is called the **empty string** and is written as: `" "`

Continued...

String concatenation

- Use the **+** operator to put strings together to form a longer string:

```
String name = "Dave";  
String message = "Hello, " + name;  
    // message is "Hello, Dave"
```



If one of the arguments of the **+** operator is a string, the other is converted to a string:

```
String a = "Agent";  
int n = 7;  
String bond = a + n; // bond is Agent7
```

- Concatenation** in print statements is useful to reduce the number of `System.out.print()` instructions:

```
System.out.print("The total is ");  
System.out.println(total);
```

```
System.out.println("The total is " + total);
```

Converting between Strings and Numbers

- To convert a string (containing integer or floating-points digits) to a number, use the static `parseInt()` or `parseDouble()` methods of the `Integer` or `Double` classes:

```
int n = Integer.parseInt(str);  
double x = Double.parseDouble(x);
```

- To convert a number to a string you could use the static `toString()` method of the `Integer` or `Double` classes, but also the empty string with the `+` operator:

```
String str = "" + n;  
str = Integer.toString(n);
```

Substrings

- The `substring()` method computes substrings of a string. The following call returns a string that is made up of the characters in the string `s`, starting at position `0` (start), and containing all characters up to, but not including, the position `5` (“past the end”):

```
String greeting = "Hello, World!";  
String sub = greeting.substring(0, 5); // sub is "Hello"
```



The first string position is labeled 0:

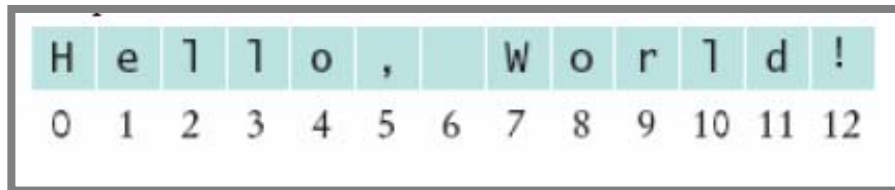
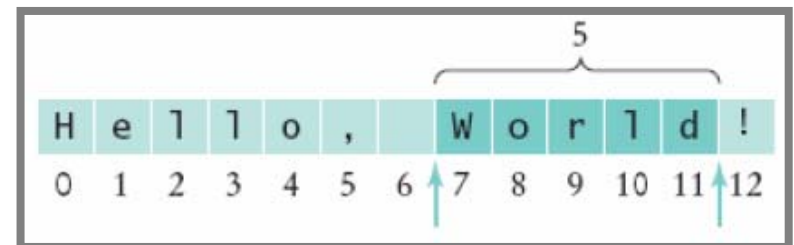


Figure 3: String Positions

Figure 4:
Extracting a Substring



- Substring length is “past the end” - start

Escape Sequences

- The **backslash character**, `\`, is used as an **escape character**. A sequence like `\"` is called an **escape sequence**. The backslash does not denote itself; instead, it is used to encode other characters that would otherwise be difficult to include in a string
- If you actually want to print a backslash, you must enter two `\\` in a row:

```
System.out.println("The file is in C:\\Temp\\File.txt");
```
- Another escape sequence occasionally used is `\n`, which denotes a **newline** or line feed character
- Java uses the **Unicode** encoding scheme; you can include any international character inside a string by writing `\u` followed by its Unicode encoding):

```
System.out.println("All the way to San Jos\u00E9!");
```

Strings and the `char` type

- Strings are sequences of Unicode characters. **Character constants** look like strings constants, except that character constants are delimited by single quotes: `'H'` is a character, `"H"` is a string containing a single character
- You can use escape sequences inside character constants:

```
System.out.println('\u00E9');
```



Characters have numeric values (character `'H'` is actually encoded as the ASCII number 72)

- The **`charAt()`** method of the `String` class returns a code unit from a string (however, if you use `char` variables, your programs may fail with some strings that contain international or symbolic characters):

```
String greeting = "Hello";  
char ch = greeting.charAt(0);
```

Self Check

13. Assuming the `String` variable `s` holds the value `"Agent"`, what is the effect of the assignment `s = s + s.length()`?
14. Assuming the `String` variable `river` holds the value `"Mississippi"`, what is the value of `river.substring(1, 2)`? Of `river.substring(2, river.length() - 3)`?

Answers

13. `s` is set to the string `Agent5`
14. The strings `"i"` and `"ssissi"`

Reading Input from a Keyboard

- When Java was first designed, not much attention was given to reading keyboard input → `System.in` has a minimal set of features—it can only read one byte at a time



Finally, in Java 5.0, a `Scanner` class was added to read keyboard input in a convenient manner. To construct a `Scanner` object, simply pass the `System.in` object to the `Scanner` constructor (you can create a scanner out of any input stream – such as a file):

```
import java.util.Scanner;

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

- Once you have a scanner, you use the `nextInt()` or `nextDouble()` methods to read the next integer or floating-point number:

```
System.out.print("Enter quantity: ");
int quantity = in.nextInt();
```

Continued...

Reading Input from a Keyboard

- When the `nextInt()` or `nextDouble()` method is called, the program waits until the user types a number and hits the Enter key (you should always provide instructions for the user, i.e. a **prompt**, before calling a Scanner method)
- The **`nextLine()`** method returns the next line of input (until the user hits the Enter key) as a `String` object:

```
System.out.print("Enter city: ");  
String city = in.nextLine();
```

- The **`next()`** method returns the next word, terminated by any **white space** (a space, the end of a line, or a tab):

```
System.out.print("Enter postal code: ");  
String postal = in.next();
```

File InputTester.java

```
01: import java.util.Scanner;
02:
03: /**
04:     This class tests console input.
05: */
06: public class InputTester
07: {
08:     public static void main(String[] args)
09:     {
10:         Scanner in = new Scanner(System.in);
11:
12:         CashRegister register = new CashRegister();
13:
14:         System.out.print("Enter price: ");
15:         double price = in.nextDouble();
16:         register.recordPurchase(price);
17:
```

Continued...

File InputTester.java

```
18:         System.out.print("Enter dollars: ");
19:         int dollars = in.nextInt();
20:         System.out.print("Enter quarters: ");
21:         int quarters = in.nextInt();
22:         System.out.print("Enter dimes: ");
23:         int dimes = in.nextInt();
24:         System.out.print("Enter nickels: ");
25:         int nickels = in.nextInt();
26:         System.out.print("Enter pennies: ");
27:         int pennies = in.nextInt();
28:         register.enterPayment(dollars, quarters, dimes,
                nickels, pennies);
29:
30:         System.out.print("Your change is ");
31:         System.out.println(register.giveChange());
32:     }
33: }
```

Output

```
Enter price: 7.55
Enter dollars: 10
Enter quarters: 2
Enter dimes: 1
Enter nickels: 0
Enter pennies: 0
Your change is 3.05
```

Reading Input from a Dialog Box

- Prior to Java 5.0, the easiest way to read input was to create a separate **pop-up window** for each input. To do so, you have to call the static `showInputDialog()` method of the `JOptionPane` class, and supply the string that prompts the input from the user:

```
import javax.swing.JOptionPane;  
String input = JOptionPane.showInputDialog("Enter price:");
```

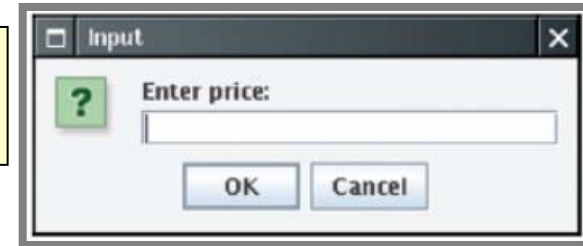


Figure 8: An Input Dialog Box

- That method returns a `String` object. If you need a number, use the `Integer.parseInt()` and `Double.parseDouble()` methods to convert the string to a number:

```
double price = Double.parseDouble(input);
```



Finally, whenever you call `JOptionPane.showInputDialog()` in your programs, you need to add the following line to the end of your `main()` method to finish the user interface thread created by the method:

```
System.exit(0);
```

Formatting Numbers

- The `printf()` method of the `PrintStream` class allows to give special format to the numbers to be printed
- Its first parameter is a **format string** containing characters to be printed and **format specifiers** (codes that start with a `%` character and end with a letter that indicates the format type). The remaining parameters of `printf()` are the values to be formatted:

```
System.out.printf("Total:%5.2f", total);
```

Prints the string `Total:` followed by a floating-point number, `total`, with a width of 5 and a precision of 2

- The `format()` method of the `String` class is similar to the `printf()` method. However, it returns a string instead of producing output:

```
String message = String.format("Total:%5.2f", total);
```

Self Check

15. Why can't input be read directly from `System.in`?
16. Suppose `in` is a `Scanner` object that reads from `System.in`, and your program calls
`String name = in.next();`
What is the value of `name` if the user enters `John Q. Public`?

Answers

15. The class only has a method to read a single byte. It would be very tedious to form characters, strings, and numbers from those bytes.
16. The value is "John". The `next` method reads the next *word*.

Chapter Summary

- Java has eight **primitive types**, including four integer types and two floating-point types
- A numeric computation **overflows** if the result falls outside the range for the number type
- **Rounding errors** occur when an exact conversion between numbers is not possible
- You use a **cast** (`typeName`) to convert a value to a different type
- Use the **`Math.round()`** method to round a floating-point number to the nearest integer
- A **`final`** variable is a constant. Once its value has been set, it cannot be changed
- Use **named constants** to make your programs easier to read and maintain

Continued...

Chapter Summary

- Assignment to a variable (`=`) is not the same as mathematical equality
- The `++` and `--` operators increment and decrement a variable
- If both arguments of the `/` operator are integers, the result is an integer and the remainder is discarded
- The `%` operator computes the remainder of a division
- The `Math` class contains methods `sqrt()` and `pow()` to compute square roots and powers
- A `static method` does not operate on an object
- A `string` is a sequence of characters. Strings are objects of the `String` class

Continued...

Chapter Summary

- Strings can be **concatenated**, that is, put end to end to yield a new longer string. String concatenation is denoted by the **+** operator
- Whenever one of the arguments of the **+** operator is a string, the other argument is converted to a string
- If a string contains the digits of a number, you use the **Integer.parseInt()** or **Double.parseDouble()** method to obtain the number value
- Use the **substring()** method to extract a part of a string
- String positions are counted starting with **0**
- Use the **Scanner** class to read keyboard input in a console window