

# Primitive Data

Dr. Chia-Ling Tsai

## Outline

- Basic data concepts
- Java primitive types
- Arithmetic in Java
- Variables (placeholders)

## Basic Data Concepts

- Bitstrings
- Representing text
- Representing other data
- Representing numbers
- Conversion

## Bitstrings

- Each bit is either 0 or 1, so it can represent two items
- A *bitstring* is a "string of bits," i.e., a list of bits, such as 11010101
- Two bits can represent *four* items
  - There are four possibilities
    - 00, 01, 10, 11
- Three bits can represent *eight* items
- Four bits can represent *16* items
- What's the formula here?
  - Each additional bit *doubles* the number of items

## Representing Text

- You have to be able to represent any *character* (letter, digit, punctuation, special “signaling” character, such as “end of a line” or “tab”)
- Different *character sets* have been designed
  - ASCII: an 8-bit character set, OK for English
  - Unicode: a 16-bit set for all languages. Java uses this one
- Exact representation doesn't matter – done by software behind the scenes

## Representing Other Data

- Data such as sound (e.g., music), still pictures, or video, usually require many bits of storage
  - Example: each single *pixel* requires storage of brightness and color information
- For data requiring too much storage, *compression* is used
  - Examples:
    - MP3 format for music
    - JPEG for pictures
  - In compressed files, usually some information is *lost*

## Representing Numbers

- Numerals
- Base 10 positional notation
- Other bases
- Digits
- Names of numbers
- Arithmetic
- Sizes of numbers
- Other numbers

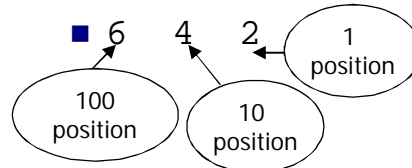
## Numerals

- The symbol used to *write* a number is called a *numeral*
  - 23 is an *Arabic* numeral and XXIII is a *Roman* numeral for the number we call "twenty-three"
  - Our standard (Arabic) numerals are *positional*: the *digits* use *base 10*
  - Roman numerals are *not* positional – they do not use a *base*
- Using a base makes possible *algorithms* for doing computations such as multiplication and division
  - You learned algorithms for multiplying and dividing Arabic numerals
  - Try them with Roman numerals!

## Base 10 Positional Notation

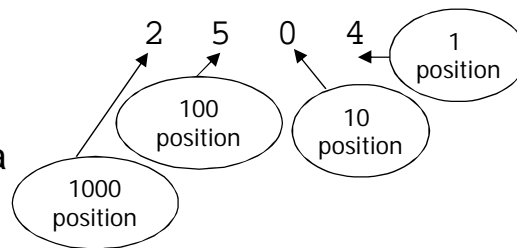
### ■ Example: 642

$$\begin{aligned} 6 \times 100 &= 600 \\ + 4 \times 10 &= 40 \\ + 2 \times 1 &= 2 \end{aligned}$$



### ■ Example: 2504

$$\begin{aligned} 2 \times 1000 &= 2000 \\ + 5 \times 100 &= 500 \\ + 0 \times 10 &= 0 \\ + 4 \times 1 &= 4 \end{aligned}$$



- Each position has a *multiplier* times a *power of 10*

## Other Bases

- Other bases can be used – any positive integer would work
  - But it would take some getting used to!
- Each position will be a power of the base
  - Example: in base 13, positions from right to left represent  $13^0$ ,  $13^1$ ,  $13^2 = 169$ ,  $13^3 = 2197$
- Example
  - How to interpret the *numeral* 642 in base 13?
  - What is the equivalent numeral in base 10?
  - $6 \times 13^2 + 4 \times 13 + 2 \times 1 = 1068$

## Digits

- You need as many digits as there are 1's in the base
  - Two for binary: 0,1
  - Eight for base eight: 0,1,2,3,4,5,6,7
  - Ten for base ten: 0,1,2,3,4,5,6,7,8,9
  - Thirteen for base thirteen: 0,1,2,3,4,5,6,7,8,9 and *three more*
    - We'll use A for 10, B for 11, and C for 12

## Names Of Numbers

- We are used to writing numbers in base 10
- Even the *names* of the numbers are based on 10
  - "six hundred forty two"
- This makes it hard to talk about other bases
- Sometimes we write  $642_{10}$  and  $642_{13}$  for "the numeral 642 interpreted in base ten" and "the numeral 642 interpreted in base thirteen"
  - So  $642_{10} = 3A5_{13}$  and  $642_{13} = 1068_{10}$
- We'll work on the conversion later

## Hexadecimal

- Decimal and binary are most important for computing – the first for us, the second for the computer
- When we want to look at the actual representation inside a computer, *hexadecimal* (base 16) is more convenient than binary
- Hexadecimal notation allows us to represent 4-bit bitstrings by a single symbol
  - Example:  $001110110001_2 = 3B1_{16}$
- Since we don't have to look "inside" memory, that's all we'll say about hexadecimal

## Arithmetic

- The algorithms you learned for arithmetic *may be used no matter what the base*
  - Example: addition algorithm
    - Add column by column, from right to left. If answer has more than one digit, *carry* to next column
    - Base 10 example
$$\begin{array}{r} 11 \\ 265 \\ + 477 \\ \hline 742 \end{array}$$
    - Base 2 example
$$\begin{array}{r} 111 \\ 111101 \\ + 110110 \\ \hline 1110011 \end{array}$$
- Since bits are used in computers, numbers are stored in, and arithmetic is carried out in, base 2 – the *binary system*

## Sizes Of Numbers

- In arithmetic, there is no “last” number
  - No matter how large a number you choose, one more than that is also a number
- In a computer, you have a certain amount of “room” for number storage
  - Each number can occupy a certain number of *bits*
  - Once you’ve used up storage, any larger number will *not fit*
    - This is called *overflow*
  - Example
    - If there are 8 bits available, the largest number is the one represented by 11111111 ( $511_{10}$ )
    - The *next* number ( $512_{10}$ ) looks like this: 00000000!
      - The first digit to the left is lost, because of overflow

## Other Numbers

- To represent a negative number, use one bit for a sign
- This is *almost* what is done in most computers
  - Leftmost bit is for the sign (1 for -, 0 for +)
  - But *two’s complement* notation is used (details in homework)
- Leaving room for a sign means you can store only half as many numbers
  - In 8 bits, the largest number is 01111111 ( $255_{10}$ )
  - The number 11111111 is *negative* ( $-255_{10}$ )
  - What if we add 1 to the largest number?
- Decimals just make for more complications
  - Later ...

## Conversion

- From another base to decimal is easy
- What about the other way?
- Here is an *algorithm*:
  - Set the dividend equal to the original numeral
  - Repeat as long as the dividend is **not** zero
    - Divide the dividend by the new base
    - *Remember the remainder*
    - Set the dividend equal to the quotient
  - Form the numeral in the new base by writing the remainders in reverse order
- ...but we don't have to do this (except in homework!)

## Example

- Convert 44 to base 2
  - Dividend = 44
  - Divide by 2: quotient = 22, remainder = 0
  - Set dividend = 22. Repeat
  - Divide by 2: quotient = 11, remainder = 0
  - Set dividend = 11. Repeat
  - Divide by 2: quotient = 5, remainder = 1
  - Set dividend = 5. Repeat
  - Divide by 2: quotient = 2, remainder = 1
  - Set dividend = 2. Repeat
  - Divide by 2: quotient = 1, remainder = 0
  - Set dividend = 1. Repeat
  - Divide by 2: quotient = 0, remainder = 1
  - Set dividend = 0. Stop
- Base 2 numeral is 101100

## Java Primitive Types

- Different kinds of numbers
- Integer types
- Range of decimal numbers
- Floating-point numbers
- Writing floating-point numbers
- Floating-point types
- Other primitive types

## Different Kinds Of Numbers

- Because numbers must be stored in a *limited* amount of space, storage differs depending on
  - Whether the number is an integer or a real
  - How large – and in the case of decimals, how precise – we require our represented numbers to be
  - Programming languages
  - Machine/platform types
- Computer arithmetic distinguishes among
  - Whole numbers (integers)
  - Floating-point numbers



## Floating-point Numbers

- Idea is to trade *accuracy* for *range*
- Must remember three things about each number
  - Sign (+ or -)
  - *Significant digits*
    - Ignore leading and trailing zeros
    - Examples: 67 for Planck's Constant and 588 for size of Milky Way
  - *Exponent*
    - Number of places (- for left, + for right) to move the decimal point from *after* the first significant digit
    - Examples: -34 for Planck's Constant and +17 for size of Milky Way

## Writing Floating-point Numbers

- You have to express the three important parts: sign, significant digits, and exponent
- Significant digits written with decimal after first digit
  - This is called the *mantissa* of the number
- Exponent written after the letter E
- Examples
  - Planck's Constant:  $6.7E-34$
  - Size of Milky Way:  $5.88E17$
- Note: scientists write  $6.7 \cdot 10^{-34}$  and  $5.88 \cdot 10^{17}$ . This is *scientific notation*.
  - Easier to use E in typing, though – no superscripts

## Java Floating-point Types

| Type                | Description   | Size    |
|---------------------|---|---------|
| <code>double</code> | Double-precision floating-point type<br>Range: about $-10^{308}$ to $+10^{308}$<br>Significant digits: about 15 | 8 bytes |
| <code>float</code>  | Single-precision floating-point type<br>Range: about $-10^{38}$ to $+10^{38}$<br>Significant digits: about 7    | 4 bytes |

- The other is for special situations
- Inaccuracies are possible – be careful (and don't use for business!)

## Other Java Primitive Types

| Type                 | Description   | Size    |
|----------------------|---|---------|
| <code>boolean</code> | Type used for logic, with the two values <code>false</code> and <code>true</code> | 1 byte  |
| <code>char</code>    | Type to represent individual characters   | 2 bytes |

- We will talk about and use `boolean` later
- Type `char` containing only one character
  - Example: `'a'` is of type `char`, while `"a"` is of type `String`

## Arithmetic In Java

- Differences
- Addition, subtraction, and multiplication
- Division
- Output

## Differences

- Each answer must have a *type*
- Size matters – if the answer needs too much room, information will be lost
- Some new symbols must be used, since standard arithmetical symbols aren't available
  - Examples:  $2 \times 3$ ,  $2 \cdot 3$ ,  $3 \div 4$ ,  $\frac{3}{4}$ ,  $\sqrt{5}$
  - In most languages, we use  $*$  for multiplication and  $/$  for division

## Addition, Subtraction, And Multiplication

- For these operations, arithmetic is as usual, except for symbols used and size issue
- Examples

$2 * 3 + 5 * 6$  gives the answer 36

$32 - 5 * 7$  gives the answer -3

But

$2147483645 + 10$  gives -2147483641, not 2147483655  
(the largest `int` that can be stored is 2147483647)

Note that you are not permitted to use commas

## Division

- Even in ordinary arithmetic, integer division of integers differs from “long” division
  - Example:  $11 \div 3$  gives *quotient* 3 with *remainder* 2 for integer division, or 3.666666666666666... (“forever”) for decimal division
- In Java, if we want an integer *we must start with integers*  
 $11 / 3$  gives 3
- For the remainder, use the symbol `%` (weird!)  
 $11 \% 3$  gives 2
- If *at least one of the numbers is a decimal* (floating-point), you get a decimal answer  
 $11.0 / 3$  gives 3.6666666666666665 (note the inaccuracy)

## Type Casting

### ■ What are the results of

$1.0 / 4.0 * 8$

$1 / 4.0 * 8$  → Converted from int to float

$1 / 4 * 8$  → Integer division

$2.5 / 0.15$

$(int) 2.5 / 0.15$

$(int) (2.5 / 0.15)$

## Variables

### ■ Basics

### ■ Sample program

### ■ Rules for naming variables

### ■ Assignment

### ■ String concatenation

### ■ Incrementing and decrementing a variable

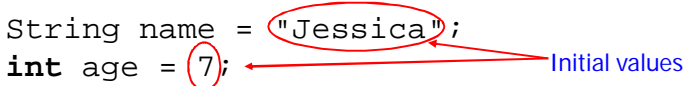
## Basics

- Programs often have to store some information
- You must *name* where an item is stored
- A named storage location is called a *variable*
  - Because you can *change* the item that is stored
  - Example: If the variable is called `age`, you can change the stored value on your birthday
- You must say what *type* of variable you are naming – for example, the `age` variable is of type `int` while a variable that store *text*, such as someone's name, is of type `String`

## Sample Program

- A program that writes a person's current age
- We must *declare* the variables we want to use – tell Java the *names* and *types* of the variables
  - The name of the person (Jessica) is a string
  - The age of the person (7) is an integer
  - In Java:

```
String name = "Jessica";  
int age = 7;
```


- We have also *initialized* these variables

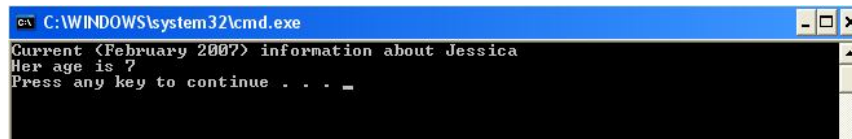
# The Program

```
// A program that displays Jessica's age as of February 2007
public class JessicasAge
{
    public static void main(String[] args)
    {
        String name = "Jessica";
        int age = 7;
        String date = "February 2007";

        System.out.print("Current (");
        System.out.print(date);
        System.out.print(") information about ");
        System.out.println(name);
        System.out.print("Her age is ");
        System.out.println(age);
    }
}
```

Declarations of variables (with Initializations)

Use of variables



```
C:\WINDOWS\system32\cmd.exe
Current (February 2007) information about Jessica
Her age is 7
Press any key to continue . . . _
```

# Rules for Naming Variables

- Give meaningful names for readability
  - Avoid names of single letter.
- The name of a variable must be a single "word"
  - For example, age but not current age
- Case sensitive
  - For example, age and Age are *different*
  - You can use case differences to combine multiple words into one, satisfying Java and aiding readability
  - Example: currentAge
- The *underscore* symbol counts as a "letter"
  - Example: current\_age

## Assignment

- You can *change* the value of a variable
  - Example: on Jessica's next birthday, I can write  
`age = 8;`
- You do *not* repeat the type name when you make an assignment
  - Declaration: `int age = 7;`
  - Assignment: `age = 8;`
- Notice that you only *declare* a variable (in this case, using `int`) *once*
- You can *assign* a value to a variable many times (That's why it's called a "variable")

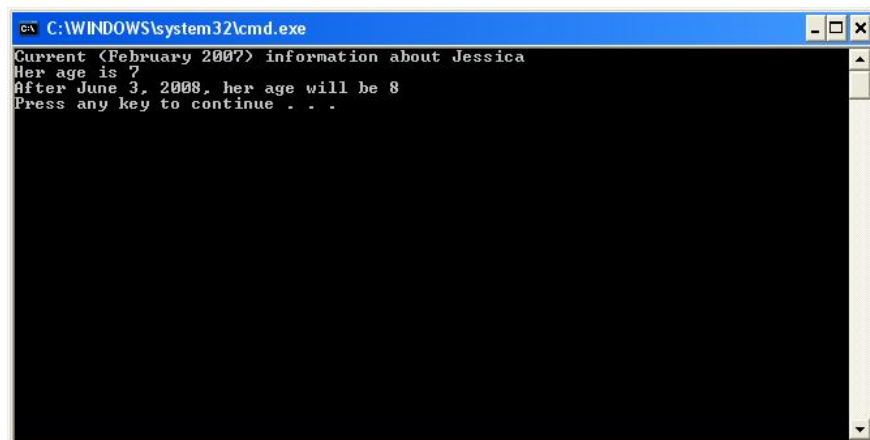
## Assignment

- If you want, you can declare a variable *without an initial value*, then use an assignment statement:  
`int age;`  
and later  
`age = 7;`  
and still later  
`age = 8;`
- You would not do this if you knew the initial value, as here, but sometimes you don't
- But *before you do something with a variable* (such as print its value) you *must* "initialize" it, either when you declare it or later in an assignment

# The Revised Program

```
// A program that displays Jessica's age as of February 2007,  
// then updates it  
public class JessicasAgeUpdated  
{  
    public static void main(String[] args)  
    {  
        String name = "Jessica";  
        int age = 7;  
        String date = "February 2007"; } ← Declarations of  
                                        variables (with  
                                        initializations)  
  
        // Original information  
        System.out.print("Current ");  
        System.out.print(date);  
        System.out.print(" information about ");  
        System.out.println(name);  
        System.out.print("Her age is ");  
        System.out.println(age);  
  
        // Updated information  
        age = 8;  
        date = "June 3, 2008"; } ← Assignments to  
                                variables  
        System.out.print("After ");  
        System.out.print(date);  
        System.out.print(", her age will be ");  
        System.out.println(age);  
    }  
}
```

# Output



```
C:\WINDOWS\system32\cmd.exe  
Current (February 2007) information about Jessica  
Her age is 7  
After June 3, 2008, her age will be 8  
Press any key to continue . . .
```

## String Concatenation

- Frequently – as in the sample programs – we have to print one string after another
- Instead of writing each string in a separate print statement, we can *concatenate* – chain together – the strings into a new one, and print that
- You can even concatenate a string and a number
- Concatenation is done using the + sign

## String Concatenation

- **Example:**  

```
String prefix = "Your password is ";  
String password = "Jessica";  
Value of prefix + password is  
"Your password is Jessica"
```
- **To print both, instead of writing**  

```
System.out.print(prefix);  
System.out.println(password);
```

  
**we can just write**  

```
System.out.println(prefix + password);
```

## String Concatenation with Numbers

- When you “add” a string to a number, Java figures you want concatenation
- Example:

```
String prefix = "Your secret code is ";
int code = 6301;
Value of prefix + code is
"Your secret code is 6301"
```
- But, be careful

```
System.out.println("2 + 3 is " + 2 + 3);
```

results in

```
2 + 3 is 23
```

## Sample Program

```
// A program that displays Jessica's age as of February 2007,
// then updates it
// Revision: Uses string concatenation
public class JessicasAgeUpdated2
{
    public static void main(String[] args)
    {
        String name = "Jessica";
        int age = 7;
        String date = "February 2007";

        // Original information
        System.out.println("Current (" + date + ") information about " + name);
        System.out.println("Her age is " + age);

        // Updated information
        age = 8;
        date = "June 3, 2008";
        System.out.println("After " + date + ", her age will be " + age);
    }
}
```

Output is the same as before

## Incrementing a Variable

- Often, we want to *increment* a variable
- This means, "add something to the variable," often 1
- To add 1 to  $x$ , we can write  
`x = x + 1;`
- Or, we can use the shorthand  
`x++;`
- To add 5 to  $x$  we write  
`x = x + 5;`
- Or use the shorthand  
`x += 5;`
- You have to be able to *read* the shorthand, but may use it or not, as you like

## Decrementing a Variable

- Similarly, we can *decrement* a variable by 1 using  
`x = x - 1;`  
or  
`x--;`
- And we can decrement by 5 using  
`x = x - 5;`  
or  
`x -= 5;`
- `x--` and `x++` are not always the same
- The second form of shorthand is also available for other operations such as `*`  
`x = x * 5` can be written `x *= 5`



## Summary

- Conversion between bases
- Java primitive types
- Arithmetic in Java
- Variables
- String concatenation