# Chapter 2 the Basics of Java Language

## 2.1 Identifiers 标识符

- Java **identifiers** are tokens that represent names of variables, methods, classes, etc.
- Examples of identifiers are
  - Car,
  - donald,
  - main,
  - System,
  - out.

## 2.1 Identifiers

- The rules for naming identifiers:
  - An identifier is a sequence of characters that consists of letters, digits, underscores (_), and dollar signs ($).
  - Identifiers must begin with either a letter, an underscore (_), or a dollar sign "$". Letters may be lower or upper case. Subsequent characters may use digits 0 to 9.

3

## 2.1 Identifiers

- Identifiers cannot use Java keywords like class, public, void, etc.
- Identifiers cannot be true, false, or null.
- No special symbols in identifiers such as exclamation mark (!), at symbol (@), number sign (#), percent sign (%), ampersand (&), circumflex accent (^), asterisk (*) and white spaces.

4

## 2.1 Reserved words保留字（关键字）

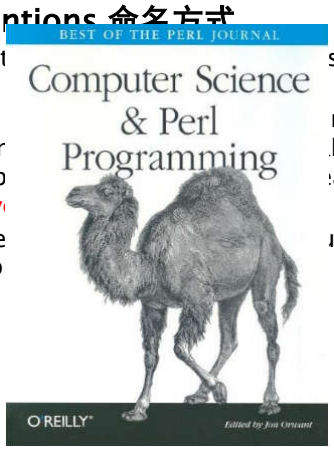
- **Keywords**, or **reserved words**, are those predefined identifiers reserved by Java for a specific purpose

| | | | | | |
|---|---|---|---|---|---|
| abstract | assert | boolean | break | byte | case |
| catch | char | class | continue | default | do |
| double | else | enum | extends | false | final |
| finally | float | for | if | implements | import |
| instanceof | int | interface | long | native | new |
| null | package | private | protected | public | return |
| short | static | strictfp | super | switch | synchronized |
| this | throw | throws | transient | true | try |
| void | volatile | while | | | |

5

## Java Source Code Style

- **Naming Conventions 命名方式**
  - Capitalize the first ... ss name. For example, CarRace.
  - Use lowercase for ... name consists of several words, cor ...king the first word lowercase and cap ...ach subsequent word. For example, mov...
  - Capitalize every le ...nderscores between words. For examp...




6

## Java Naming Conventions

| Name | Convention |
|---|---|
| class name | should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc. |
| interface name | should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc. |
| method name | should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc. |
| method with boolean result | should start with 'is', e.g. isPersistent() etc. |
| method for getting/setting values | should start with get or set, e.g. getFirstName(), setTime() etc. |
| variable name | should start with lowercase letter e.g. firstName, orderNumber etc. |
| package name | should be in lowercase letter e.g. java, lang, sql, util etc. |
| constants name | should be in uppercase letter. e.g. RED, YELLOW, MAX_PRIORITY etc. |

7

## 2.2 Primitive Data Types基本数据类型

- A data type is a set of values and a set of operations on those values.
- There are two kinds of type in Java:
  - primitive types 基本数据类型
  - and reference types 引用数据类型
- There are three kinds of reference type:
  - class, 类
  - interface, 接口
  - array, 数组

8

## 2.2 Primitive Data Types

- The Java programming language defines eight primitive data types:
  - boolean (for logical),
  - char (for text),
  - byte, short, int and long (for integral),
  - double and float (for floating point)

9

## 2.2 Primitive Data Types

- A boolean data type represents two states: true and false.
- A character data type (char) represents a single Unicode character, which is 16 bits.
- A string is a sequence of characters. The String data type is not a primitive type in Java.

10

## 2.2 Primitive Data Types

- Java has two basic kinds of numeric values:
  - integers and
  - floating points.
- All numeric types are signed
- Standard arithmetic operators for the integral types include addition, subtraction, multiplication, division, and remainder.

11

## 2.2 Primitive Data Types

- Integral types and size

| Type | Size（bit） | Minimum | Maximum |
|---|---|---|---|
| byte | 8 | -128 | 127 |
| short | 16 | -32768 | 32767 |
| int | 32 | -2,147,483,648 | 2,147,483,647 |
| long | 64 | $-2^{63}$ | $2^{63}-1$ |

12

## 2.2 Primitive Data Types

- The typical foating-point number that can be represented

$$<\text{significant\_digits}> \times <\text{base}>^{<\text{exponent}>}$$

13

## 2.2 Primitive Data Types

- The arithmetic operators +, -, *, and / are defined for floating-point numbers.
- Beyond the build-in operators, the Java Math class defined the square root, logarithm function, exponential function, trigonometric functions, and other common functions for floating-point numbers.

14

## 2.2 Primitive Data Types

A "Float" number occupies 32 bits (4 bytes) and its significant digits part has a precision of 24 bits (about 7 decimal digits) while a "Double" number occupies 64 bits (8 bytes) and its significant digits part has a precision of 53 bits (about 16 decimal digits).

| Type | Size（bit） | Minimum(approximately) | Maximum(approximately) |
|---|---|---|---|
| float | 32 | $-3.4\times10^{38}$ | $-1.4\times10^{-45}$ |
| | | $1.4\times10^{-45}$ | $3.4\times10^{38}$ |
| double | 64 | $-1.8\times10^{308}$ | $-4.9\times10^{-324}$ |
| | | $4.9\times10^{-324}$ | $1.8\times10^{308}$ |

## 2.3 Literals 字面值

- A **literal** is a source-code representation of a data-type value, which is a constant value that appears directly in a program.
  - boolean literals,
  - character literals,
  - integer literals,
  - floating-point literals, and
  - String literals.

```
int i = 100;
double d = 200.0;
```

16

## 2.3 Literals

- Boolean literals have only two values, true or false.
- Character literals represent single Unicode characters.
  - 'a'
  - '\n'

17

## 2.3 Literals

- special escape sequences

| Notation | Character represented |
|---|---|
| \n | New Line (\u000a) |
| \r | Carriage Return (\u000d) |
| \f | Form Feed (\u000c) |
| \b | Backspace (\u0008) |
| \s | Space (\u0020) |
| \t | Tab(\u0009) |
| \" | Double Quote (\u0022) |
| \' | Single Quote (\u0027) |
| \\ | Backslash (\u005c) |
| \ddd | octal character (ddd) |
| \uxxxx | Hexadecimal UNICODE character (xxxx) |

18

## 2.3 Literals

- A 16-bit Unicode takes two bytes, preceded by \u, expressed in four hexadecimal digits that run from '\u0000' to '\uFFFF'.
- For example,
- "欢迎" : "\u6B22\U8FCE".
- the Greek letter "α" : "\u03B1".

19

## 2.3 Literals

- Integer literals come in different formats:
  - decimal (base 10),
  - hexadecimal (base 16) or
  - octal (base 8).
  - For decimal literals, there is no special notation, for example, 12.
  - The hexadecimal literals are preceded by 0x or 0X, such as 0xC.
  - The octal literals are preceded by 0, such as 014.
- Floating-point literals can be expressed in standard or scientific notation, such as 583.45 (standard) or 5.8345e2 (scientific).

20

## 2.3 Literals

- String literals represent multiple characters and are enclosed by double quotes.
- "Hello World"

21

## 2.3 Literals

```
char ch = '张';          // 16-bit Unicode character
byte b = 0x7f;           // maximum value of byte literals，127，in hexadecimal
int i = 0x2f;   // 0x indicates hexadecimal
int j = 0X2F;   // 0X indicates hexadecimal
int k = 0177; // The prefix 0 indicates octal
long m = 200L;           // 64-bit decimal literal
long n = 200l;           // 64-bit decimal literal
float f1 = 128.6F;       // 32-bit float literal
float f2 = 128.6f;       // 32-bit float literal
float f3 = 1e-45f;       // 10^-45, in scientific notation
float f4 = 1e+9f;        // 10^9, in scientific notation
double d1 = 1256.8d;           // 64-bit double literal
double d2 = 1256.8D;           // 64-bit double literal

double d3 = 1.2568e3d;         // 1.2568×10^3 , in scientific notation
```

22

## 2.3 Literals

- // out of scope of an integer, L is necessary.
- long la = 9876543234L;
- //not necessary.
- long lb = 98765432L;
- long lc = 98765432;

```
long l = 999999999999;
```

The literal 999999999999 of type int is out of range

Press 'F2' for focus

23

## 2.4 Variables

- A **variable** is a cell that stores a value in a computer's memory.
- The value of a variable can change throughout a program.
- Java variables must be declared before being used.
- A variable declaration statement associates a variable name with a type at compile time.

24

## 2.4 Variables

- declaration
- <data_type> <name> [=<initial_value>];

```
int x;              // Declare x to be an integer variable;
double radius;          // Declare radius to be a double variable;
char a;             // Declare a to be a character variable;
double grade = 0.0;     //declare a data type with variable name
grade, double data type and initialized to 0.0
```

25

## 2.4 Variables

- System.out.println()
- System.out.print()
- System is a build-in class in Java and out is the standard output object declared in System, which is already open and ready to accept output data before the first statement in your program. Typically, this object corresponds to display output.

## 2.4 Variables

- The methods, println(), or print(), can print out any values in any types.



## 2.4 Variables

- Address, variables and values

| Memory Address | Variable name | Value hold |
|---|---|---|
| 0X08100000 | i | 10 |
| … | | … |
| 0X081000F0 | name | 0X0810A000 |
| … | | … |
| 0X0810A000 | | "Donald" |

## 2.4 Variables

- widening:
- byte > short > char > int > long > float > double
- 2 * 0.3

- Two types are compatible if values of one type can appear wherever values of the other type are expected, and vice versa.

29

## 2.4 Variables

- Cast can perform the narrowing conversions

```
int i;
long g = 100;
i = (int) g;
```

30

## 2.4 Variables

```
public class CastingDemo {
    public static void main(String[] argv) {
        int i;
        double j = 2.75;
        i = j;              //COMPILE ERROR HERE
        i = (int) j;                // Cast occurs; i gets 2
        System.out.println("i ="+ i);
        long b;
        b = i;              // NO COMPILE ERROR HERE
        System.out.println("b ="+ b);
    }
}
```

- i = 2
- b = 2

31

## 2.5 Operators

- Assignment 赋值运算符
- arithmetic operators,算术运算符
- relational operators, 关系运算符
- logical operators, and 逻辑运算符
- conditional operators条件运算符

32

## 2.5 Operators

- Assignment

```
int a = 1;          //assign 1 to variable a
int b = 2 + 3;      //assign the result of 2 + 3 to b
String name = "Hello";          //assign the reference to the String object "Hello" to name
int d = a = b;      //assign b to a, then assign a to d; results in d, a, and b being equal
```

- A variable can store only one value of its declared type.
- The expression on the right-hand side of an assignment statement must evaluate to a value compatible with the type of the variable on the left-hand side

33

## 2.5 Operators

- arithmetic operators

| Expression | Results in |
|---|---|
| op1 + op2 | op1 added to op2 |
| op1 - op2 | op2 subtracted from op1 |
| op1 * op2 | op1 multiplied with op2 |
| op1 / op2 | op1 divided by op2 |
| op1 % op2 | Calculates the remainder of dividing op1 by op2 |

34

## 2.5 Operators

- The increment operator (++) adds 1 to the operand while the decrement operator (--) subtracts 1 from the operand.
- If the increment and decrement operators are applied after a variable, it is called the postfix form of the operator.
- On the contrary, if the increment and decrement operators are applied before a variable, it is called the prefix form.
- For example, if the variable i contains 1 currently, the following statement assigns 2 to i and 1 to j:
- j = i ++;
- However, j = ++ i assigns 2 to both i and j.

35

## 2.5 Operators

- Relational operators

| Expression | Results in |
|---|---|
| op1 > op2 | Is op1 greater than op2? |
| op1 >= op2 | Is op1 greater than or equal to op2? |
| op1 < op2 | Is op1 less than to op2? |
| op1 <= op2 | Is op1 less than or equal to op2? |
| op1 == op2 | Are op1 and op2 equal? |
| op1 != op2 | Are op1 and op2 not equal? |

36

# 2.5 Operators

- Logical operators

| Expression | Results in |
|---|---|
| op1 && op2 | If both op1 and op2 are true, result is true. |
| | If either op1 or op2 are false, the result is false |
| | If op1 is false, the result is false and op2 is not evaluated. |
| op1 \|\| op2 | If either op1 or op2 are true, the result is true. |
| | If op1 is true, the result is true and op2 is not evaluated. |
| ! op1 | If op1 is true, the result is false. |
| | If op1 is false, the result is true. |
| op1 ^ op2 | If op1 is true and op2 is false, the result is true. |
| | If op1 is false and op2 is true, the result is true. |
| | Otherwise, the result is false. |
| | Both op1 and op2 are evaluated before the test. |

37

# example

Test.java

```
public class Test {
    public static void main(String[] args)
    {
        if(getBoolean1()&&getBoolean2())
        {
            System.out.println("true");
        }
    }
    public static boolean getBoolean1()
    {
        System.out.println("1");
        return false;
    }
    public static boolean getBoolean2()
    {
        System.out.println("2");
        return false;
    }
}
```

Problems Javadoc Declaration Console

<terminated> Test [Java Application] C:\Program Files\Java\jre1.8.0_40\bin\jav

```
1
```

Shortcircuiting And

Test.java

```
public class Test {
    public static void main(String[] args)
    {
        if(getBoolean1()&getBoolean2())
        {
            System.out.println("true");
        }
    }
    public static boolean getBoolean1()
    {
        System.out.println("1");
        return false;
    }
    public static boolean getBoolean2()
    {
        System.out.println("2");
        return false;
    }
}
```

Problems Javadoc Declaration Console

<terminated> Test [Java Application] C:\Program Files\Java\jre1.8.0_40\bin\javaw.ex

```
1
2
```

Logical And

38

## 2.5 Operators

- Bitwise operators

| Expression | Operator Name | Results in |
|---|---|---|
| ~op1 | Compliment | Flip each bit, ones to zeros, zeros to ones |
| op1 & op2 | Bitwise AND | AND each bit in op1 with corresponding bit in op2 |
| op1 \| op2 | Bitwise OR | OR each bit in op1 with corresponding bit in op2 |
| op1 ^ op2 | Bitwise XOR | XOR each bit in op1 with corresponding bit in op2 |
| op1 << op2 | Left Shift | Shift op1 to the left by op2 bits. High order bits lost. Zero bits fill in right bits. |
| op1 >> op2 | Right-Signed Shift ( propagates the sign bit ) | Shift op1 to the right by op2 bits. Low order bits lost. Same bit value as sign (0 for positive numbers, 1 for negative) fills in the left bits. |
| op1 >>> op2 | Right-Unsigned Shift (does not propagate the sign bit) | Shift op1 to the right by op2 bits. Low order bits lost. Zeros fill in left bits regardless of sign. |

39

## 2.5 Operators

- Examples

00011100 << 2 results in 01110000;
00011100 >> 2 results in 00000111
10011011 >> 2 results in 00100110
00110111 & 01000110 results in 00000110
00110111 | 0001100 results in 00111111
00110101 ^ 00111010 results in 00001111
~00101010 results in 11010101

40

## 2.5 Operators

- Conditional operator
- <boolean_expression> ? <expression1> :< expression2>
- Example

```
if (a > b) {
    max = a;
}
else {
    max = b;
}
can be rewritten in a single line like this:
max = (a > b) ? a : b;
```

41

## 2.5 Operators

- Operator Precedence

| Level | Operator | | | | | | | | | | | | Associates |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | = | *= | /= | %= | += | -= | <<= | >>= | >>>= | &= | ^= | \|= | R to L |
| 2 | ? : | | | | | | | | | | | | R to L |
| 3 | \|\| | | | | | | | | | | | | L to R |
| 4 | && | | | | | | | | | | | | L to R |
| 5 | \| | | | | | | | | | | | | L to R |
| 6 | ^ | | | | | | | | | | | | L to R |
| 7 | & | | | | | | | | | | | | L to R |
| 8 | == | != | | | | | | | | | | | L to R |
| 9 | < | <= | > | >= | instanceof | | | | | | | | L to R |
| 10 | << | >> | >>> | | | | | | | | | | L to R |
| 11 | + | - | | | | | | | | | | | L to R |
| 12 | * | / | % | | | | | | | | | | L to R |
| 13 | new | (type) | | | | | | | | | | | R to L |
| 14 | ++x | --x | +x | -x | ~ | ! | | | | | | | R to L |
| 15 | . | [] | (args) | x++ | x-- | | | | | | | | L to R |

42

## 2.5 Operators

- Operations in parentheses are always performed first.
- When operators of the same precedence are mixed together in the absence of parentheses, unary operators and assignment operators are evaluated right-to-left, while the remaining operators are evaluated left-to-right.
- For example,
- A * B / C means (A * B) / C,
- A = B = C means A = (B = C).
- In a complicated expression, it is good practice to use parentheses even when it is not necessary, to make the expression more clear 适当多用括号，让表达式更清晰可读

43

## 2.6 Expressions and Statements

- An **expression** is a literal, a variable, or a sequence of operations on literals and/or variables.

```
int x = 2;
int y ;
y = x + 3;
```

```
int x = 2;
int y, z;
z = (y = x + 3) * 4;
```

44

## 2.6 Expressions and Statements

- A **statement** in Java forms a complete and fundamental unit to be executed and can include one or more expressions.
- A semicolon terminates all statements except blocks.
- A block is a series of zero or more statements between a pair of open and close curly braces.

## 2.6 Expressions and Statements

- The three groups of statements encompass the different kinds of statements in Java are
  - expression statements,
  - declaration statements, and
  - control flow statements.

```
int i;              //declaration statement
i = 1;              //expression statement
if (i < 10 ) {   //control flow statement
   System.out.println(i);       //expression statement
}
```

## 2.7 Getting Data from the Keyboard

```
import java.util.Scanner;

public class InputTest {
    public static void main(String[] args) {
        // Create a Scanner object
        Scanner sc = new Scanner(System.in);

        // Prompt to enter an integer
        System.out.println("Enter an integer:");
        int x = sc.nextInt();
        System.out.println("Your input is: " + x);

        // Prompt to enter a double value
        System.out.println("Enter a double value:");
        double y = sc.nextDouble();
        System.out.println("Your input is: " + y);

        // Prompt to enter a string
        System.out.println("Enter a string:");
        String s = sc.next();
        System.out.println("Your input is: " + s);

    }
}
```

- Enter an integer:12
- Your input is: 12
- Enter a double value: 3.14
- Your input is:3.14
- Enter a string: Car
- Your input is: Car

47

## 2.7 Getting Data from the Keyboard

- get a single letter

```
Scanner sc = new Scanner(System.in);
char ch = sc.nextLine().charAt(0);

System.out.println(ch);
```

48

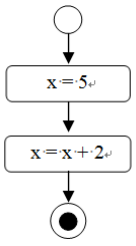
## 2.8 Control Structures

- sequence structures,（顺序）
- decision structures, （选择）
- repetition structures（循环）

49

## 2.8 Control Structures

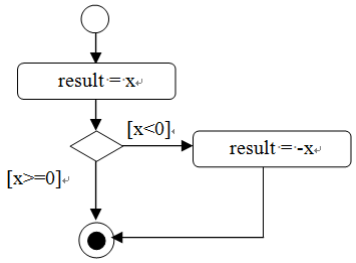
- Sequence Structure




50

## 2.8 Control Structures

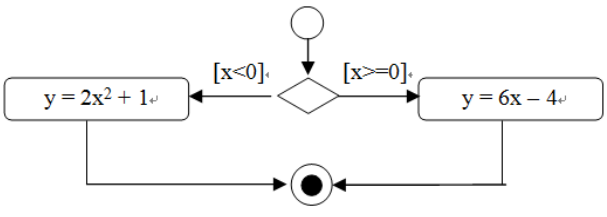
- Decision structures




51

## 2.8 Control Structures

$$y = \begin{cases} 2x^2 + 1 & (x < 0) \\ 6x - 4 & (x \geq 0) \end{cases}$$




52

## 2.8 Control Structures

```
if( <boolean_expression> ){
   <statement1>;
   <statement2>;
   . . .
}else{
   <statement3>;
   <statement4>;
   . . .
 }
```

53

```
switch( <switch_expression> ){
   case <case_selector1>:
      <statement1>;
      <statement2>;
      break;
   case <case_selector2>:
      <statement1>;
      <statement2>;
      break;
   default:
      <statement1>;
      <statement2>;
}
```

- When a switch is encountered, the <switch_expression> is evaluated first, and control passes to the case whose selector matches the value of the expression.
- The statements in sequence from that point on are executed until a break statement is encountered, skipping then to the first following statement after the end of the switch statement.
- If none of the cases is satisfied, the default block is executed.

54

## 2.8 Control Structures

```
int testScore = 100;
      char grade;
      switch (testScore / 10) {
         case 0:
         case 1:
         case 2:
         case 3:
         case 4:
         case 5:
            grade = 'F';
            break;
         case 6:
            grade = 'D';
            break;
         case 7:
            grade = 'C';
            break;
         case 8:
            grade = 'B';
            break;
         case 9:
         case 10:
            grade = 'A';
            break;
         default:
            grade = 'I';
      }
```

55

## 2.8 Control Structures

- Repetition structure
  - while statement
  - do-while statement
  - for statement

56

## Repetition structure

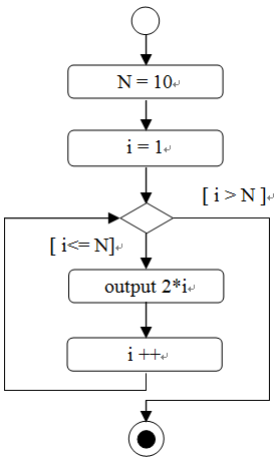
While loop

```
while (<loop_condition> ) {
        <statement1>;
        <statement2>;
        …
}
```

57

## while loop

```
int N=10;
int i = 1;
while (i <= N) {
  System.out.println("The even number "
+ i + " is: " + 2*i );
  i++;
}
```




initialization,
loop condition,
loop condition update,
loop body

58

## do-while loop

- do{
  - <statement1>;
  - <statement2>;
  - . . .
- }while( <loop_condition> );

- The do-while loop is almost the same as a while loop except that it will execute its block before it evaluates its condition.

59

## for loop

- for (<initialization>; <loop_condition>; <step_update>) {
  - <statement1>;
  - <statement2>;
  - . . .
- }


```
int i, j = 1;
for(i = 0; i < 10; i ++)
    j = j + 1;
```

60

```
public class Test{
    public static void main(String[] args){
        for(int i=0;i<11;i++){
            System.out.println(i);
        }
    }
}
```

```
public class Test{
    public static void main(String[] args){
        int i = 0;
        for(;;){
            if(i>10){
                break;
            }
        System.out.println(i);
        i++;
        }
    }
}
```

61

## Foreach loop

- Syntax：
  - for(type x : obj)
  - {
  - statements with x;
  - }
- example：
  - public class Test {
  - public static void main(String[] args) {
  - int[] a = {1,2,3};
  - for(int i : a)
  - System.out.print(i + " ");
  - }
  - }

62

## For loop and foreach loop

```
ArrayList<Integer> al = new ArrayList<Integer>();
for (int i = 0; i < 10; i++) {
        al.add(new Integer(i));
}
for(Integer i : al){
        System.out.println(i);
}
for (int i = 0; i < al.size(); i++) {
        if (al.get(i).equals(new Integer(5))) {
        al.remove(i);
        }
}
//for (Integer i : al) {
//        if (i.equals(new Integer(5))) {
//        al.remove(i);
//        }
//}
```

63

## branching statements 跳转语句

- break
- continue
- return

64

## Example: break

```
public class Test {
   public static void main(String[] args) {
      int j;
      for (int n = 1; n <= 100; n++) {
         for (j = 2; j <= n - 1; j++) {
            if (n % j == 0)
               break;
            }
            if (j >= n - 1)
               System.out.println(n);
         }
      }
}
```

65

## Example: continue

```
public class Test {
   public static void main(String[] args) {
      for (int n = 9; n <= 100; n++) {
         if (n % 9 == 0) {
            System.out.println(n);
         } else {
            continue;
         }
      }
   }
}
```

66

## break n and continue n

```
String[] arr = new String[] { "a", "b", "c" };
labelA: for (String s : arr) {
    for (String ss : arr) {
      for (String sss : arr) {
            System.out.print(sss);
              continue  labelA;
    }
  }
}
```

67

## Example: return

- return statement is used to exit method and return value.

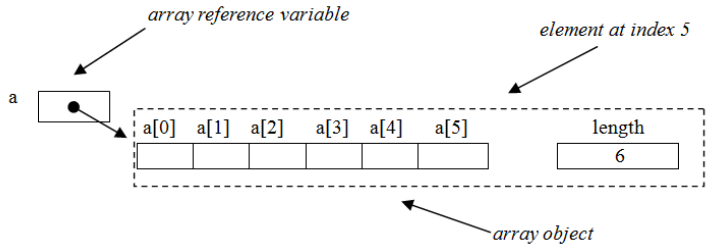
```
public int set(int a,int b){
    return sum=a+b;
}
```

```
public void set(int a,int b){
    sum=a+b;
    return;
}
```

68

## 2.9 arrays

- int[] a = new int[6];




69

## initialization

- int[] a = { 1, 3 , 5, 6};

- Scanner sc = new Scanner(System.in);
- for (i = 0; i < 4; i++){
- a[i] = sc.nextInt();
- }

70

## access elements

- We use a number called an index or a subscript to access an array element

- //assigns 10 to the first element in the array
- a[0] = 10;
- //prints element in the array
- System.out.print(a[3]);

71

## Example

- int[] a = new int[6];
- int i;
- // Input six numbers into an array.
- Scanner s = new Scanner(System.in);
- for (i=0; i< a.length; i++){
- a[i] = s.nextInt();
- }
- 
- // Output the reversed numbers in the array a.
- for(i = a.length - 1; i>=0; i--){
- System.out.print(a[i]+" ");
- }

- The size of an array can be found via the attribute length of an array
- a.length

72

## Command Line Arguments命令行参数

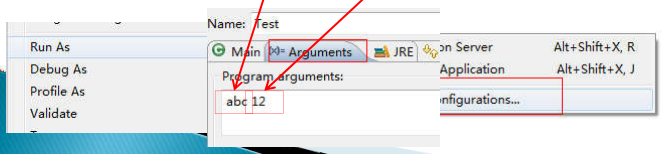
- public static void main(String[] args)

```
public class Test {

public static void main(String[] args) {
      String arg1 = args[0];
      int arg2 = Integer.parseInt(args[1]);
      System.out.println(arg1);
      System.out.println(arg2);
   }
}
```

```
C:\>javac Test.java

C:\>java Test abc 12
abc
12
```




73

## Java.Util.Arrays class Arrays工具类

Fast filling

```
int[] a = new int[1024];
java.util.Arrays.fill(a, 128); // Every element as 128

int[][] a = new int[10][20];
   for(int[] r : a) {
      Arrays.fill(r, 128);
}
```

74

## Compare method

- Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal

- int[] a = {2, 4, 5, 6};
- int[] b = {2, 4, 5, 6};
- int[] c = {3, 4, 5, 6};
- System.out.println(java.util.Arrays.equals(a, b)); //true
- System.out.println(java.util.Arrays.equals(b, c)); //false

75

## Compare method

- String[] dinnerA = {"Soup", "Mushroom", "Seasonal Fruit" };
- String[] dinnerB = {"Soup", "Mushroom", "Seasonal Fruit" };

- if(java.util.Arrays.equals(dinnerA, dinnerB)) {
- System.out.println("Dinner A is equal to dinner B.");
- } else {
- System.out.println("Dinner A and B are not equal.");
- }

76

## Sort method

```
String[] dinnerA = {"Soup", "Mushroom", "Seasonal Fruit"};
Arrays.sort(dinnerA);
for (String s : dinnerA) {
    System.out.println(s);
}
```

- Mushroom
- Seasonal Fruit
- Soup

77

## Search method

```
int [] numbers = {1, 2, 2, 3, 4, 5, 6};
System.out.println(Arrays.binarySearch(numbers, 5));
```

78

## Java.util.Arrays

- Fast outputting

```
public class Practice {
        public static void main(String[] args){
                String[] A = {"H","e","l","l","o"};
                System.out.println(Arrays.asList(A));
        }
}
```

Result:
[H, e, l, l, o];

79

## Array copyOf method

```
String[] a = {“a”,“d”,“e”,“w”,“f”};
        String[] b = new String[4];
        String[] c = new String[5];
        String[] d = new String[6];
        b = Arrays.copyOf(a, b.length);
        c = Arrays.copyOf(a, c.length);
        d = Arrays.copyOf(a, d.length);
      System.out.println(“Elements of Array b: " + Arrays.asList(b));
      System.out.println("Elements of Array c: " + Arrays.asList(c));
      System.out.println("Elements of Array d: " + Arrays.asList(d));
```

Elements of Array b: [a, d, e, w]
Elements of Array c: [a, d, e, w, f]
Elements of Array d: [a, d, e, w, f, null]

80

# API Specifications


81

# 2.10 work with built-in classes 预定义类

- String
- StringBuffer
- Random
- Date and Time

82

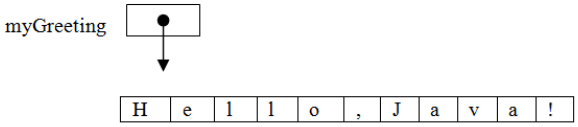
## String

A string is a sequence of characters, which can be treated as an array of characters.

Java provides a build-in class for string.

Java.lang.String

- String literal: "Hello, Java!"




83

## Declaration and creation

- String myGreeting;
- myGreeting = new String("Hello, Java!");
- String myGreeting = "Hello, Java!";
- String aStr = null;
- String bStr = "";

from an array of characters

```
char[] message = {'N', 'i', 'c', 'e', ' ', 'd', 'a', 'y'};
String myGreeting = new String(message);
```

84

## String equals() method

- returns true if and only if the argument is not null and is a String object that represents the same sequence of characters as this string

- String s1=new String("Hello");
  String s2=new String("Hello");
  System.out.println(s1 == s2) ;
  false
- System.out.println(s1.equals(s2));
  true
  String s1="Hello";
  String s2="Hello";
  System.out.println(s1 == s2) ;
  true
- System.out.println(s1.equals(s2));
  true

86

## Concatenation

```
String myAddress = “HBUT, No.1 Lizhi Road,";
myAddress = myAddress + “Wuhan";
```

- The above concatenation statements are equivalent to

```
myAddress = myAddress.concat(“Wuhan");
```

- String s1="Hello";
- String s2=s1+"world";
- String s3=s1+"world";
- System.*out.println(s2 == s3);*
- False

- String str1 = "abc";
  String str2 = "abc";
- String str3 = str1+str2; //not into pool
  String str4 = str1+"cd"; // not into pool
  String str5 = "ab"+str2; // not into pool
  String str6 = "ab"+"cd"; // into pool
- String str7 = "abcd";

- System.out.println(str6==str7);
- True

88

## String substring() Method

- This method has two variants and returns a new string that is a substring of this string.
  - public String substring(int beginIndex)
  - public String substring(int beginIndex, int endIndex)
  - Example:
  - String Str = new String("Hello World!");
  - System.out.println(Str.substring(6));
  - System.out.println(Str.substring(6, 11));
  - Output:

World!
World

89

## String split() Method

- Splits this string around matches of the given regular expression
  - String[] split(String regex, int limit)
  - String[] split(String regex)

```
String str = "boo:and:foo";
String[] a = str.split(":", 2);
String[] b = str.split(":", 5);
String[] c = str.split(":", -2);
String[] d = str.split("o", 5);
String[] e = str.split("o", -2);
String[] f = str.split("o", 0);
String[] g = str.split("m", 0);
```

90

## other methods

- length() returns the length of a string.
- charAt(int index) returns a character from a string at a specified index, where the index is between 0 and s.length() – 1.

## count spaces and letters in the given paragraph

- // a paragraph to be analyzed
- String paragraph = "The JDK includes the JRE plus command-line "
- +"development tools such as compilers and debuggers that are "
- +"necessary or useful for developing applets and applications.";

- int spaces = 0, // Count of spaces
- letters = 0; // Count of letters

## count spaces and letters in the given paragraph

```
// Check all the characters in the string
int paragraphLength = paragraph.length(); // Get the string length
for (int i = 0; i < paragraphLength; i++) {
    char ch = paragraph.charAt(i);
    //Check for letters
    if (Character.isLetter(ch)) {
        letters++;
    }
    // Check for spaces
    if (Character.isSpaceChar(ch)) {
        spaces++;
    }
}
```

```
System.out.println("The paragraph contains " + letters + " letters and
" + spaces + " spaces.\n");
```

## StringBuffer

- strings created with the String class cannot be modified.
- They are called **immutable**.
- The StringBuffer class enables flexible string operations

94

## StringBuffer

- StringBuffer s = new StringBuffer();
- s.append("Welcome to Java");
- s.append("!"); //changes s to "Welcome to Java!"
- s.insert(11,"XML and "); //the new s is "Welcome to XML and Java!"
- s.delete(8,11); //changes the buffer to "Welcome XML and Java!"
- s.replace(8,11,"HTML"); //changes s to "Welcome HTML and Java!"
- s.setCharAt(0,'w'); //sets the buffer to "welcome to Java!"

95

## StringBuffer

- toString() method returns the string from the object;
- length() method returns the number of characters actually stored in the object;
- capacity() method returns the current capacity of the object. The capacity is the number of characters it is able to store without having to increase its size. The StringBuffer object's capacity is automatically increased if more characters are added and exceed its capacity.
- charAt(index) method returns the character at a specific index in the object;

96

## Random Numbers

- The Random class
  - nextInt() Returns a random number of data type int.
  - nextInt(int n) Returns a random number of data type int ranging from 0(inclusive) to n(exclusive)
  - nextDouble() Returns a random double ranging from 0.0(inclusive) to 1.0(exclusive)
  - nextBoolean() Returns a random boolean (true/false value)

97

## Random Numbers

```
Random aRandom = new Random();
System.out.println(aRandom.nextInt());
//print a random number between 1 and 10
System.out.println(aRandom.nextInt(10) + 1);
System.out.println(aRandom.nextDouble());
System.out.println(aRandom.nextBoolean());
```

- 2139801327
- 3
- 0.4405496397503082 5
- false

- run once again:
- -419027158
- 10
- 0.4255746718974058 3
- false

98

## Random numbers

- Random aRandom = new Random();
- //set the seed of aRandom to 7
- aRandom.setSeed(7);
- //the number generated will not change in every run since the seed is always the same
- System.out.println(aRandom.nextInt());
  - A run displays:
  - -1156638823
  - run once again:
  - -1156638823

99

## BigInteger

- BigInteger a = new BigInteger("66666666666666666");
- BigInteger b = new BigInteger("22222222222222222");
- BigInteger c;
- //get the sum of the two numbers
- c = a.add(b);
- //get the average of the two numbers

- BigInteger average = c.divide(new BigInteger("2"));
- //prints the output on the screen
- System.out.println("Average is = " + average);

100

## BigDecimal

- BigDecimal bd1 = new BigDecimal("840.9");
- BigDecimal bd2 = new BigDecimal("12");
- BigDecimal bd3 = new BigDecimal(new Double(840.9).toString());
- System.out.println(bd1);
- System.out.println(bd2);
- System.out.println(bd3);
- System.out.println(bd1.divide(bd2,2,RoundingMode.HALF_UP));

```
double a= 1.2-0.4;
System.out.println(a);
System.out.println(a == 0.8);

0.7999999999999999
false
System.out.println((a-0.8)<1e-10);
```

```
BigDecimal a= new BigDecimal("1.2").subtract(new BigDecimal("0.4"));
System.out.println(a);
```

## Date and Time

- import java.text.SimpleDateFormat;
- import java.util.Date;
- 
- public class TestDate {
- public static void main(String[] args) {
- Date now = new Date();
- System.out.println(now);
- 
- SimpleDateFormat dateFormatter = new SimpleDateFormat("yyyy.MM.dd hh:mm:ss a z E ");
- System.out.println(dateFormatter.format(now));
- 
- }
- }

- Wed Sep 26 13:46:57 CST 2012
- 2012.09.26 01:46:57 下午 CST 星期三

102

## Calendar class

- The class Calendar maintains a set of calendar fields such as YEAR, MONTH, DAY_OF_MONTH, HOUR, MINUTE, SECOND, MILLISECOND and provides operations on these calendar fields, such as getting the date of the previous week or roll forward by 3 days.

103

## Calendar

```
import java.util.Calendar;
 public class TestCalendar {
        public static void main(String[] args) {
                Calendar cal = Calendar.getInstance();
                int year = cal.get(Calendar.YEAR);
                int month = cal.get(Calendar.MONTH); // 0 to 11
                int day = cal.get(Calendar.DAY_OF_MONTH);
                int hour = cal.get(Calendar.HOUR_OF_DAY);
                int minute = cal.get(Calendar.MINUTE);
                int second = cal.get(Calendar.SECOND);

                System.out.printf("Now is %4d/%02d/%02d
%02d:%02d:%02d\n", year,month + 1, day, hour, minute, second);
 //Output: Now is 2012/09/26 13:55:05
        }
}
```

104

## Calendar

- getTime() Returns a Date object based on this Calendar's value.
- setTime() Sets this Calendar's time with the given Date
- getTimeInMillis() Returns this Calendar's time value in milliseconds.
- setTimeInMillis() Sets the new time in UTC milliseconds from the epoch
- setTimeZone() Sets the time zone with the given time zone value

105

## Gregorian calendar

- The calendar that we use today is called the Gregorian calendar.
- The Gregorian calendar is today's internationally accepted civil calendar and is also known as the "Western calendar" or "Christian calendar". It was named after the man who first introduced it in February 1582: Pope Gregory XIII.

106

## Gregorian calendar

- The calendar is strictly a solar calendar based on a 365-day common year divided into 12 months of irregular lengths. Each month consists of either 30 or 31 days with 1 month consisting of 28 days during the common year. A Leap Year adds an extra day to make the second month of February 29 days long rather than 28 days.
- In the Gregorian calendar, a leap year is a year that is divisible by 4 but not divisible by 100, or it is divisible by 400.

107

## Gregorian calendar

- Calendar.getInstance() returns an instance of implementation class java.util.GregorianCalendar.
- The constructor GregorianCalendar() uses the current time, with the default time zone and locale.

108

## Timing control

- Java provides static methods in System class for timing control:
  - System.currentTimeMillis()
  - System.nanoTime().

109

## currentTimeMillis

- System.currentTimeMillis() returns the current time in milliseconds since January 1, 1970 00:00:00 GMT (known as an "epoch"), in long.
  - // Measuring elapsed time
  - long startTime = System.currentTimeMillis();
  - // The code being measured
  - .....
  - long estimatedTime = System.currentTimeMillis() - startTime;

110

## nanoTime()

- System.nanoTime() returns the current value of the most precise available system timer, in nanoseconds, in long.
- Introduced in JDK 1.5, nanoTime() is meant for measuring relative time intervals instead of providing absolute timing.



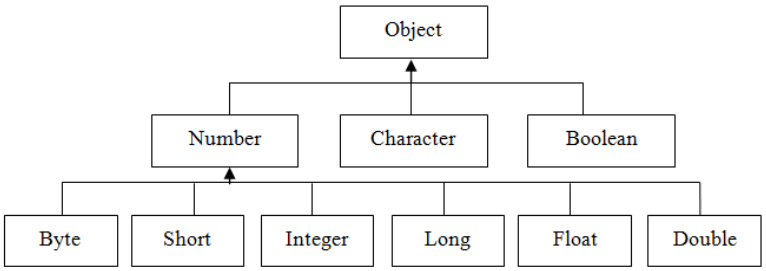
## nanoTime()

- // Measuring elapsed time
- long startTime = System.nanoTime();
- // The code being measured
- .....
- long estimatedTime = System.nanoTime() - startTime;
- Note that milli is $10^{-3}$, nano is $10^{-9}$.

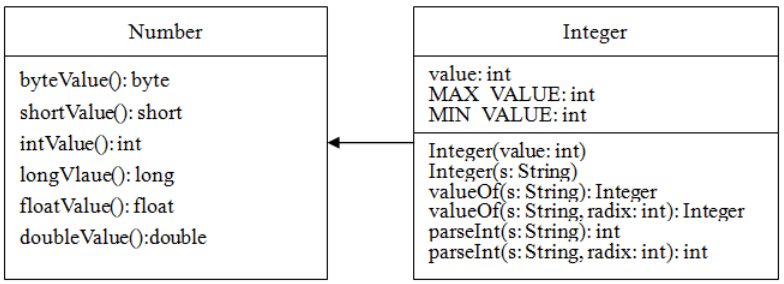
## Wrapper Classes

- Java allows you to manipulate the primitives by wrapper classes that encapsulates a single value for the them.
  - int x = 2;
  - Integer y = new Integer(2);

## Wrapper Classes



## Wrapper Classes




## Wrapper Classes

- Each numeric wrapper class extends the Number class, which contains the methods doubleValue(), floatValue(), intValue(), longValue(), shortValue(), and byteValue().
- These methods "convert" objects into primitive type values.

## Wrapper Classes

- Integer x = new Integer(12);
- Integer y = Integer.valueOf("12");

- //convert a string to a int
- int n = Integer.parseInt("1234");

## Wrapper Classes

- Boxing is the process of wrapping a primitive inside an object
- Auto boxing is the process done without explicit call to constructors.
- Unboxing is the reverse process that allows extracting a value without an explicit call to the xxxValue() methods.

```
Integer x = 23;//auto boxing
int i = x;
        //unboxing
i +=10;
Integer y = i;
        //autoboxing
```

```
Integer x = new Integer(23);
int i = x.intValue();
i += 10;
Integer y = new Integer(i);
```

## Object Pool

- Integer i1 = **new Integer(123);**
- Integer i2 = **new Integer(123);**
- Integer i3 = 123;
- Integer i4 = 123;

- Boolean b1 = **new Boolean(true);**
- Boolean b2 = **new Boolean(true);**
- Boolean b3 = **true;**
- Boolean b4 = **true;**

- Double d1 = **new Double(1.0);**
- Double d2 = **new Double(1.0);**
- Double d3 = 1.0;
- Double d4 = 1.0;

- System.*out.println(i1 == i2);* false
- System.*out.println(i3 == i4);* true
- System.*out.println(b1 == b2);* false
- System.*out.println(b3 == b4);* true
- System.*out.println(d1 == d2);* false
- System.*out.println(d3 == d4);* false

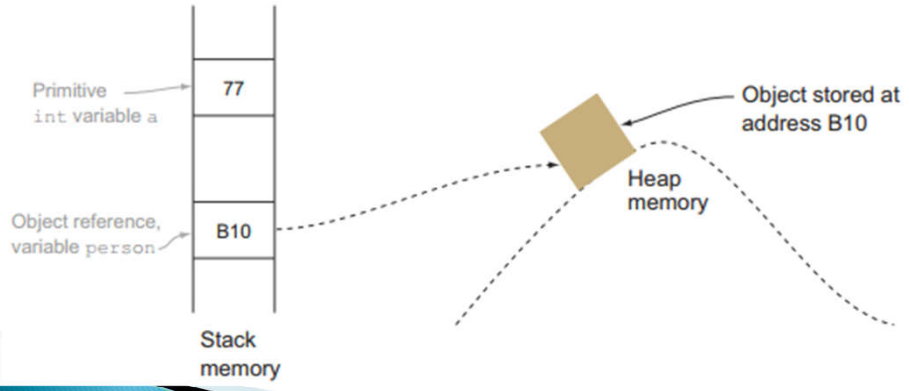


## Difference between primitive type and reference type

- For a primitive type of variable if you visit it's memory location you will find the value of the variable.
- For a reference type of variable if you visit it's memory location unlike the primitive type you will find a memory address pointing to other location and not the values of variables in Object. This memory address points to a location where the details of Object and values of variables inside it reside.

## primitive type and reference type

- int a = 77;
- Person person = new Person();

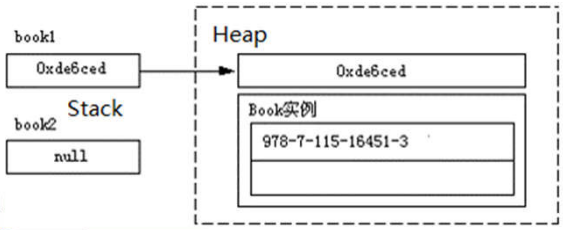



121

## primitive type and reference type

```
public class Book {
  String isbn = "978-7-115-16451-3";
  String name = "Java Textbook";
  String author = "Tom";
  float price = 59.00F;
}

Book book1 = new Book();
Book book2 = null;
```

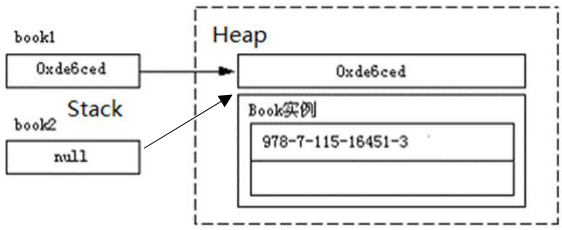



122

## primitive type and reference type

Book book1 = new Book();
Book book2 = null;

book2 = book1;




123