Programming in Java

Identifiers, Keywords, and Types

Hua Huang, Ph.D.

Spring 2019

Objectives

- Use **comments** in a source program
- Distinguish between valid and invalid **identifiers**
- Recognize Java technology **keywords**
- List the **eight primitive types**
- Define **literal values** for numeric and textual types
- Define the terms **primitive variable** and **reference variable**
- **Numeric output format** control
- Declare variables of class type
- Construct an object using **new**
- Describe **default initialization**
- Describe the significance of a reference variable
- State the consequences of assigning variables of class type

Relevance

- Do you know the **primitive Java types**?

- Can you describe the difference between variables holding **primitive values** as compared with **object references**?

Comments

- The **three permissible styles of comment** in a Java technology program are:

```
// comment on one line

/* comment on one
* or more lines
*/

/** documentation comment
*  can also span one or more lines
*/
```

- Question: Is comment a necessity? How do you like it?

Semicolons, Blocks, and White Space

- A **statement** is one or more lines of code terminated by a **semicolon** (;):

```
totals = a + b + c
+ d + e + f;
```

- A **block** is a collection of statements bound by opening and closing braces:

```
{
   x = y + 1;
   y = x + 1;
}
```

Semicolons, Blocks, and White Space(Cont.)

- A class definition uses a special block:

```
public class MyDate {
    private int day;
    private int month;
    private int year;
}
```

- You can **nest block statements**.

```
while ( i < large ) {
    a = a + i;
    // nested block
    if ( a == max ) {
        b = b + a;
        a = 0;
    }
    i = i + 1;
}
```

Semicolons, Blocks, and White Space(Cont.)

- Any amount of **white space** is permitted in a Java program.

- Whites spaces: **Spaces**, **tabs**, and **new-line** characters
- For example:

```
{int x;  x=23*54;}
```

is equivalent to:

```
{
   int x;
   x = 23 * 54;
}
```

Identifiers

- **Identifiers** have the following characteristics:
  - Are **names** given to a **variable**, **class**, or **method**
  - Can start with a **Unicode letter**, **underscore** (_), or **dollar sign** ($), **Subsequent** characters can be **digits**.
  - Are **case-sensitive** and have no maximum length

Question: **Which identifiers are valid?**

- `identifier`
- `userName`
- `_a`
- `user_name`
- `_sys_var1`
- `byte`
- `$change`
- `2mail`
- `_$`
- `room#`
- `.f`
- `-d`
- `_____2_w`
- `变量1`
- `:b`
- `this_is_a_very_detailed_name_for_an_identifier`

Java Programming Language Keywords

abstract	continue	for	new	switch
assert[1.4]	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum[1.5]	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

serveliteral words: null, true, and false

Data Types

- **Primitive types** – Are simple values as opposed to objects.

- **Class types** (**Reference types**)– Are used for more complex types, including all of the types that you declare yourself. **Class types are used to create objects.**

Primitive Data Types

- The Java programming language defines **eight** primitive types:
  - Logical – **boolean**
  - Textual – **char**
  - Integral – **byte**, **short**, **int**, and **long**
  - Floating – **double** and **float**

Primitive Type	Category	Integer or Float Length	Range
boolean	Logical	Not applicable	true, false
char	Textual	16 bits	0 to 2^{16}-1
byte	Integral	8 bits	-2^{7} to 2^{7} -1
short	Integral	16 bits	-2^{15} to 2^{15} -1
int	Integral	32 bits	-2^{31} to 2^{31} -1
long	Integral	64 bits	-2^{63} to 2^{63} -1
float	Floating point	32 bits	
double	Floating point	64 bits	

Logical – `boolean`

- The `boolean` primitive has the following characteristics:
  - The `boolean` data type has two literals, `true` and `false`.
  - For example, the statement:

```
boolean truth = true;
```

declares the variable `truth` as `boolean` type and assigns it a value of `true`.

Textual – `char`

- The textual **`char`** primitive has the following characteristics:
  - Represents a **16-bit Unicode** character
  - Must have its literal enclosed in **single quotes** (' ')
  - Uses the following notations:

`'a'`	The letter `a`
`'\t'`	The tab character
`'\u????'`	A specific Unicode character, `????`, is replaced with exactly four hexadecimal digits .For example, `'\u03A6'` is the Greek letter phi [Φ].

Question: How about the char type in C/C++??

Textual – `String`

- The textual **String** type has the following characteristics:
  - Is **not a primitive data type**; it is a **class**
  - Has its literal enclosed in **double quotes** (" ")

    "The quick brown fox jumps over the lazy dog."
  - Can be used as follows:

```
// declares two String variables and initializes them
String greeting = "Good Morning !! \n";
String errorMessage = "Record Not Found !";

// declares two String variables
String str1, str2;

// using the new keyword to create a String object
String s1 = new String("Hello");
```

Question: Could you tell me the difference of symbol ”A” and 'A'??

Integral – `byte`, `short`, `int`, and `long`

- The integral primitives have the following characteristics:
  - Integral primates use three forms: **Decimal**, **octal**, or **hexadecimal**

2	The decimal form for the integer 2.
077	The leading 0 indicates an octal value.
0xBAAC	The leading 0x indicates a hexadecimal alue.
1234_5678	0b11110000 0b1111_0000 (valid for JDK 7.0 and above)

- Literals have a default type of **`int`**.
- Literals with the suffix **L or l** are of type `long`.

```
byte b1 = 0x03;                    // ??
byte b2 = 128;                     // ??
byte b3 = 0x03L;                   // ??
```

Integral – `byte`, `short`, `int`, and `long`(Cont.)

- Integral data types have the following **ranges**:

Integer Length	Name or Type	Range
8 bits	**`byte`**	-2^{7} to $2^{7}-1$
16 bits	**`short`**	-2^{15} to 2^{31} -1
32 bits	**`int`**	-2^{31} to 2^{31} -1
64 bits	**`long`**	-2^{63} to 2^{63} -1

Floating Point – `float` and `double`

- The floating point primitives have the following characteristics:
  - Floating-point literal includes either a **decimal point** or **one of the following**:
    - `E` or `e` (add exponential value)
    - `F` or `f` (float)
    - `D` or `d` (double)

3.14	A simple floating-point value (**a double**)
6.02E23	A large floating-point value
2.718**F**	A simple float size value
123.4E+306**D**	A large double value with redundant D
3.141_592_653	Valid for JDK 7.0 and above

Floating Point – `float` and `double`(Cont.)

- **Literals have a default type** of `double`.
- Floating-point data types have the following sizes:

Float Length	Name or Type
32 bits	`float`
64 bits	`double`

Question: Are the following codes correct?

```
float pi = 3.1415;          ??
double d = 3.1415f;         ??
```

Variables, Declarations, and Assignments

```
1     public class Assign {
2     public static void main (String args []){
3                 int x, y;                    // declare integer variables
4                 float z = 3.414f;            // declare and assign floating point
5                 double w = 3.1415;           // declare and assign double
6                 boolean truth = true;        // declare and assign boolean
7                 char c;                      // declare character variable
8                 String str;                  // declare String variable
9                 String str1 = "bye";         // declare and assign String variable
10                c = 'A';                     // assign value to char variable
11                str = "Hi out there!";       // assign value to String variable
12                x = 6;                       // assign values to int variables
13                y = 1000;
14          }
15    }
```

Formatting Numeric Print Output

- The printf and format Methods(Since JDK 1.5)
- Can be used to replace print and println, are equivalent;
- Usage: like printf in C, format specifiers begin with (%) and end with a converter, see documented in java.util.Formatter.

```
int i = 461012;
System.out.format("The value of i is: %d%n", i);
//%n==\n, You are recommended to use %n rather than \n
```

- The `DecimalFormat` Class
  - Can be used to control the display of leading and trailing zeros, prefixes and suffixes, grouping (thousands) separators, and the decimal separator.
  - Offers flexibility but are more complex.

Formatting Numeric Print Output(Cont.)

Converter	Flag	Explanation
d		A decimal integer.
f		A float.
n		A new line character appropriate to the platform running the application. You should always use **`%n`**, rather than **`\n`**.
td, te		A date & time conversion—2-digit day of month. td has leading zeroes as needed, te does not.
tD		A date & time conversion—date as %tm%td%ty
	08	Eight characters in width, with **leading zeroes** as necessary.
	+	Includes **sign**, whether positive or negative.
	,	Includes locale-specific **grouping characters**.
	-	**Left-justified**.
	.3	Three places after decimal point.
	10.3	Ten characters in width, right justified, with three places after decimal point.

Formatting Numeric Print Output(Cont.)

printf & format Example:

```
import java.util.Calendar;
import java.util.Locale;
public class TestFormat {
  public static void main(String[] args) {
    long n = 461012;
    System.out.format("%d%n", n);          //  -->  "461012"
    System.out.format("%08d%n", n);        //  -->  "00461012"
    System.out.format("%+8d%n", n);       //  -->  " +461012"
    System.out.format("%,8d%n", n);       //  -->  " 461,012"
    System.out.format("%+,8d%n%n", n);    //  -->  "+461,012"
    double pi = Math.PI;
    System.out.format("%f%n", pi);         //  -->  "3.141593"
    System.out.format("%.3f%n", pi);       //  -->  "3.142"
    System.out.format("%10.3f%n", pi); //  -->  "     3.142"
    System.out.format("%-10.3fasdf%n", pi); //  -->  "3.142     "
    System.out.format(Locale.FRANCE,"%-10.4f%n%n", pi); //--> "    3,1416"
    Calendar c = Calendar.getInstance();
    System.out.format("%tB %te, %tY%n", c, c, c);  // -->  "May 29, 2014"
    System.out.format("%tl:%tM %tp%n", c, c, c);   //  -->  "2:34 am"
    System.out.format("%tD%n", c);                 //  -->  "05/29/14"
```

Formatting Numeric Print Output(Cont.)

DecimalFormat Example:

```
import java.text.*;
public class DecimalFormatDemo {
   static public void customFormat(String pattern, double value ){
      DecimalFormat myFormatter = new DecimalFormat(pattern);
     String output = myFormatter.format(value);
      System.out.println(value + "  " + pattern + "  " + output);
   }

   static public void main(String[] args) {
      customFormat("###,###.###", 123456.789);
      customFormat("###.##", 123456.789);
      customFormat("000000.000", 123.78);
      customFormat("$###,###.###", 12345.67);
   }
}
```

The output is:

```
123456.789  ###,###.###
123,456.789123456.789  ###.##
123456.79123.78  000000.000
000123.78012345.67  $###,###.###
$12,345.67
```

Java Reference Types

- In Java technology, beyond **primitive types** all others are **reference types**(**Class Types**).
- A **reference variable** contains a **handle** to an object.

- Reference Type are from:
  - Java SE **class libraries**
  - Commercial or open source class libraries(3rd parties)
  - User created or in-house classes

Java SE Class Libraries

Library Name	Sample Classes in Library
`java.lang`	`Enum, Float, String, Object`
`java.util`	`ArrayList, Calendar, Date`
`java.io`	`File, Reader, Writer`
`java.math`	`BigDecimal, BigInteger`
`java.text`	`DateFormat, Collator`
`javax.crypto`	`Cipher, KeyGenerator`
`java.net`	`Socket, URL, InetAddress`
`java.sql`	`ResultSet, Date, Timestamp`
`javax.swing`	`JFrame, JPanel`
`javax.xml.parsers`	`DocumentBuilder, SAXParser`

Wrapper Classes

Primitive Data Type	Wrapper Class
`boolean`	**`Boolean`**
`byte`	**`Byte`**
`char`	**`Character`**
`short`	**`Short`**
`int`	**`Integer`**
`long`	**`Long`**
`float`	**`Float`**
`double`	**`Double`**

Autoboxing of Primitive Types

- Autoboxing has the following description:
  - **Conversion of primitive types to the object equivalent**
  - Wrapper classes **not always needed**
  - Example:

```
int pInt = 420;
Integer wInt = new Integer(pInt); // this is called boxing
int p2 = wInt.intValue(); // this is called unboxing

Integer wInt = 5;                    ??
int p2 = wInt;                       ??
```

Autoboxing of Primitive Types(Cont.)

- Language feature **used most often when dealing with collections**
- Wrapped primitives **also usable in arithmetic expressions**
- **Performance loss** when using **autoboxing**
- Some useful wrapper class methods

```
int x = Integer.valueOf(str).intValue();
int x = Integer.parseInt(str);
```

- Other methods:

```
byte byteValue()    short shortValue()
int intValue()      long longValue()
float floatValue()  double doubleValue()
```

Reference Types Example

```
1     public class MyDate {
2       private int day = 1;
3       private int month = 1;
4       private int year = 2000;
5       public MyDate(int day, int month, int year) { ... }
        // How to implement it?
6       public String toString() { ... }
7     }
```

```
1     public class TestMyDate {
2       public static void main(String[] args) {
3         MyDate today = new MyDate(22, 7, 1964);
4       }
5     }
```

Constructing and Initializing Objects

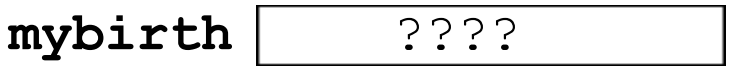
- Calling `new Xyz()` performs the following actions:
  a. **Memory is allocated** for the object.
  b. Initialize using **default values**(**implicit**)
  c. **Explicit** attribute initialization is performed.
  d. A **constructor** is executed.
  e. The object **reference** is returned by the **new** operator.

- The reference to the object is assigned to a variable.
- An example is:

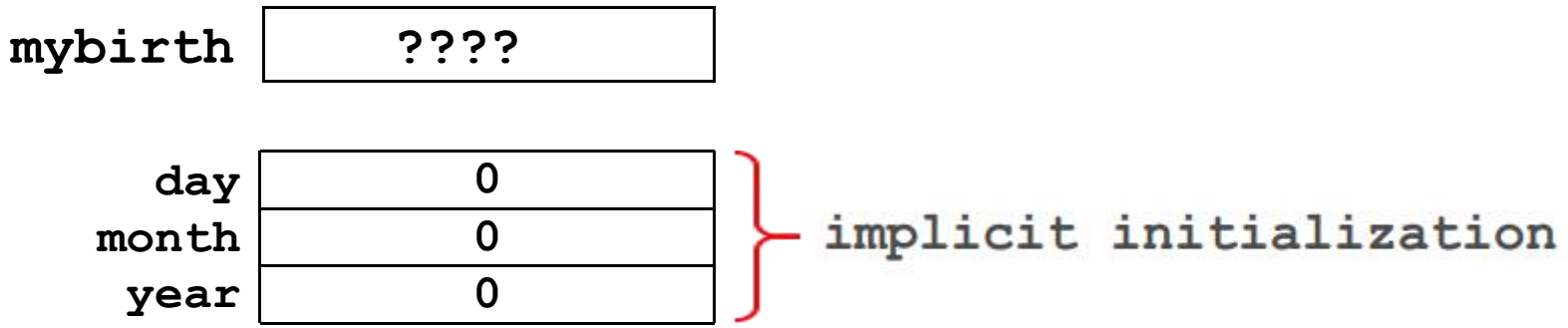
```
MyDate mybirth = new MyDate(22, 7, 1964);
```

Memory Allocation and Layout

- A declaration allocates storage only for a reference:
- `MyDate mybirth = new MyDate(22, 7, 1964);`



- Use the **new** operator to allocate space for `MyDate`:
- `MyDate mybirth = new MyDate(22, 7, 1964);`



Default Values

Data Type	Default Value
`boolean`	`false`
`byte`	`0`
`char`	`'\u0000'`
`short`	`0`
`int`	`0`
`long`	`0L`
`float`	`0.0f`
`double`	`0.0d`
Any class type	`null`

Explicit Attribute Initialization

- Initialize the attributes as follows:
- `MyDate mybirth = new MyDate(22, 7, 1964);`

mybirth	????

day	1
month	1
year	2000

- The default values are taken from the **attribute declaration** in the class.

Executing the Constructor

- Execute the matching constructor as follows:

```
MyDate mybirth = new MyDate(22, 7, 1964);
```

mybirth	????

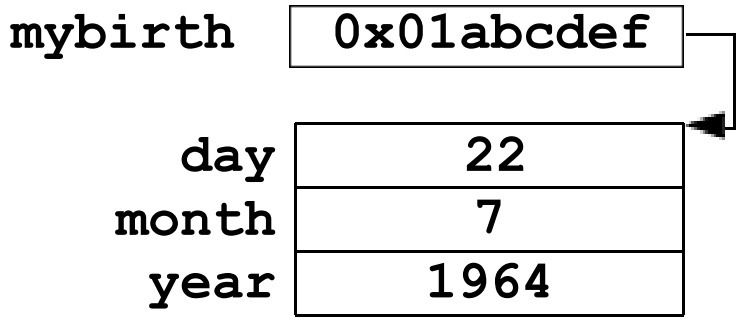
day	22
month	7
year	1964

- In the case of an overloaded constructor, the first constructor can call another.

Assigning a Variable

- Assign the newly created object to the reference variable as follows:

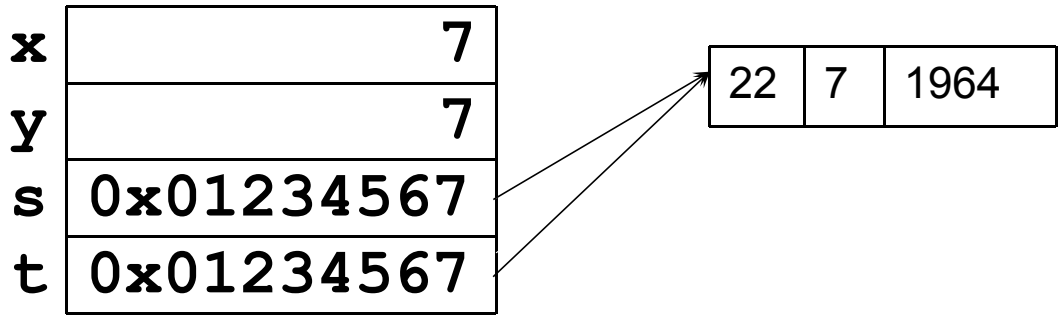
```
MyDate mybirth = new MyDate(22, 7, 1964);
```



Assigning References

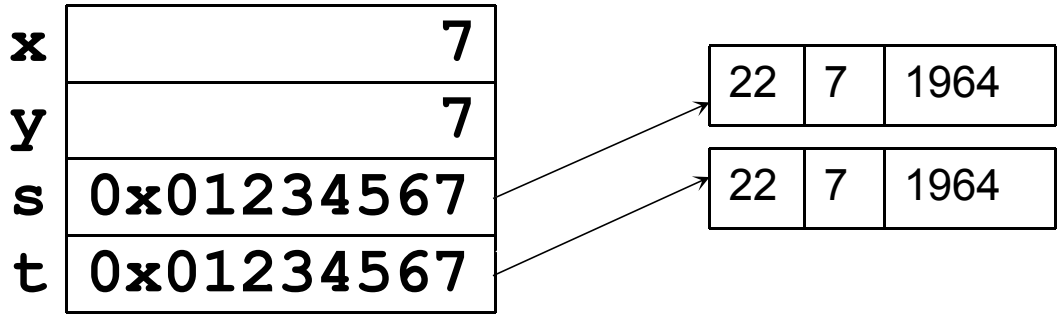
Two variables refer to a single object:

```
1       int x = 7;
2       int y = x;
3       MyDate s = new MyDate(22, 7, 1964);
4       MyDate t = s;
```



Reassignment makes two variables point to two objects:

```
5       t = new MyDate(22, 12, 1964);
```



Pass-by-Value

- In a single virtual machine, the Java programming language **only passes arguments by value**.

- When an **object instance** is passed as an argument to a method, the v**alue of the argument** is a **reference to the object.**

- The **contents of the object can be changed** in the called method, but **the original object reference is never changed**.

Pass-by-Value(Cont.)

```
1     public class PassTest {
2
3         // Methods to change the current values
4         public static void changeInt(int value) {
5             value = 55;
6         }
7         public static void changeObjectRef(MyDate ref) {
8             ref = new MyDate(1, 1, 2000);
9         }
10        public static void changeObjectAttr(MyDate ref){
11            ref.setDay(4);
12        }
```

Pass-by-Value(Cont.)

```
13
14          public static void main(String args[]) {
15              MyDate date;
16              int val;
17
18              // Assign the int
19              val = 11;
20              // Try to change it
21              changeInt(val);
22              // What is the current value?
23              System.out.println("Int value is: " + val);
```

The result of this output is:

```
Int value is: ??
```

Pass-by-Value(Cont.)

```
24
25              // Assign the date
26              date = new MyDate(22, 7, 1964);
27              // Try to change it
28              changeObjectRef(date);
29              // What is the current value?
30              System.out.println("MyDate: " + date);
```

The result of this output is:

```
MyDate: ??
```

Pass-by-Value(Cont.)

```
31
32              // Now change the day attribute
33              // through the object reference
34              changeObjectAttr(date);
35              // What is the current value?
36              System.out.println("MyDate: " + date);
37          }
38      }
```

The result of this output is:

```
MyDate: ??
```

The `this` Reference

- Here are a few uses of the **this** keyword:
  - To **resolve ambiguity between instance variables** and **parameters**
  - To pass the **current object** as a parameter to another method or constructor

The `this` Reference(Cont.)

```
1     public class MyDate {
2         private int day = 1;
3         private int month = 1;
4         private int year = 2000;
5
6         public MyDate(int day, int month, int year) {
7             this.day      =     day;
8             this.month    =     month;
9             this.year     =     year;
10        }
11        public MyDate(MyDate date){
12          this.day  =     date.day;
13          this.month=     date.month;
14          this.year =     date.year;
15        }
```

The `this` Reference(Cont.)

```
16
17      public MyDate addDays(int moreDays) {
18          MyDate newDate = new MyDate(this);
19          newDate.day = newDate.day + moreDays;
20          // Not Yet Implemented: wrap around code...
21          return newDate;
22      }
23      public String toString() {
24          return "" + day + "-" + month + "-" + year;
25      }
26  }
```

The `this` Reference(Cont.)

```
1     public class TestMyDate {
2         public static void main(String[] args) {
3             MyDate mybirth = new MyDate(22, 7, 1964);
4             MyDate the_next_week = mybirth.addDays(7);
5
6             System.out.println(the_next_week);
7         }
8     }
```

Questions or Comments?