Programming in Java

Expressions, Flow Control, and Array

Hua Huang, Ph.D.

Spring 2019

Objectives

- Distinguish between **instance** and **local variables**
- Describe how to **initialize instance variables**
- Identify and correct a **Possible reference before assignment** compiler error
- Recognize, describe, and use Java software **operators**
- Distinguish between **legal and illegal assignments** of primitive types
- Identify **boolean expressions** and their requirements in control constructs
- Recognize **assignment compatibility** and required **casts** in fundamental types
- Use if, switch, for, while, and do constructions and the **labelled forms** of break and continue as flow control structures in a program

Objectives(Cont.)

- **Declare** and **create** **arrays** of primitive, class, or array types
- Explain why elements of an array are initialized
- Explain how to initialize the elements of an array
- Determine the **number of elements** in an array
- Create a **multidimensional** **array**
- Write code to **copy array** values from one array to another

Relevance

- What types of variables are useful to programmers?

- Can multiple classes have **variables with the same name** and, if so, what is their scope?

- What types of **control structures** are used in other languages? What methods do these languages use to control flow?

- What is the **purpose of an array**?

Variables and Scope

- **Local variables** are:
  - Variables that are **defined inside a method** and are called **local**, **automatic**, **temporary**, or **stack** variables
  - Variables that are **created** when the method is executed are **destroyed** when the method is exited

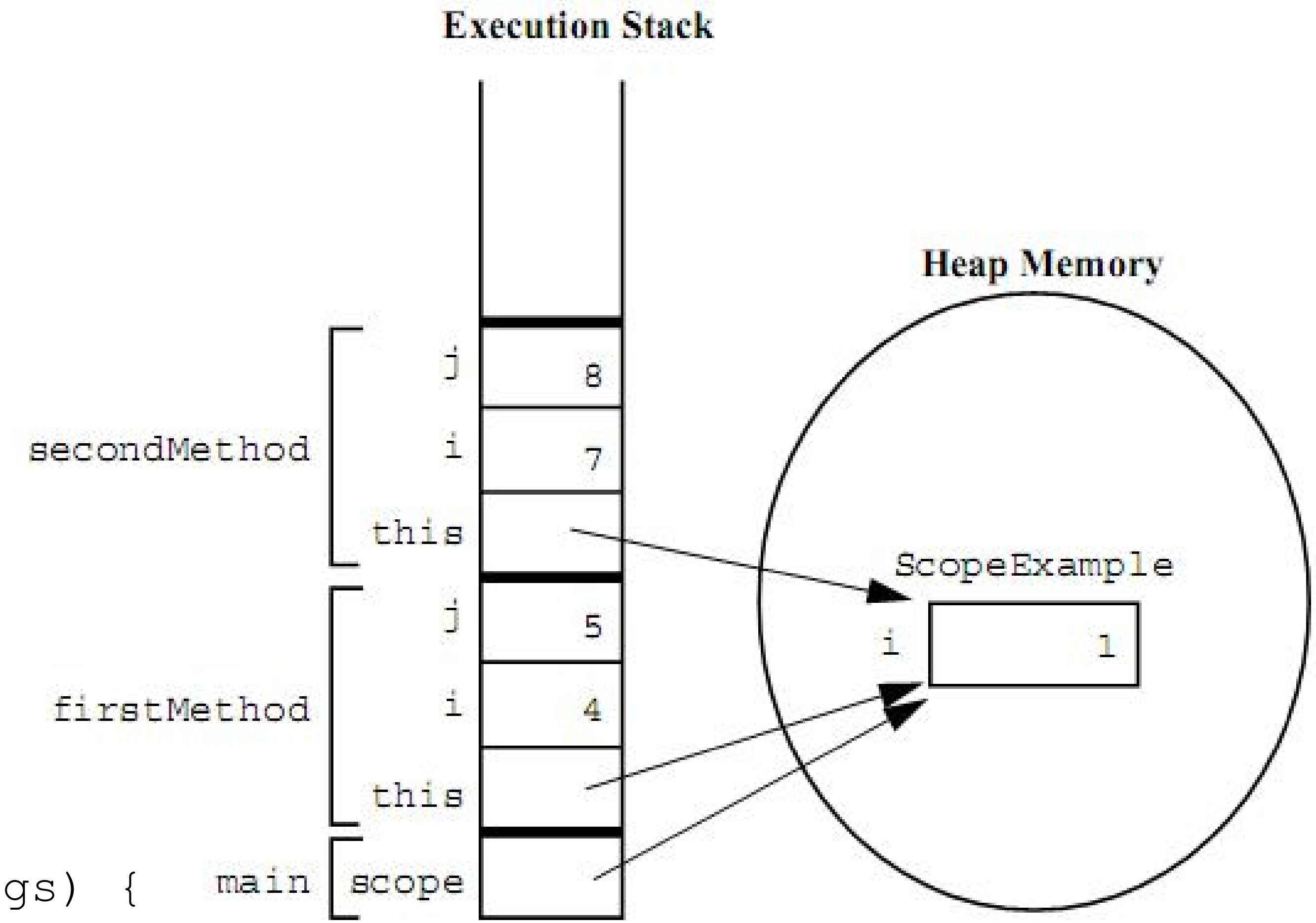
- Variable initialization comprises the following:
  - **Local variables** require **explicit initialization**.
  - **Instance variables** are **initialized automatically**.

Variable Scope Example

```
public class ScopeExample {
  private int i=1;

  public void firstMethod() {
    int i=4, j=5;
    this.i = i + j;
    secondMethod(7);
  }
  public void secondMethod(int i){
    int j=8;
    this.i = i + j;
  }
}

public class TestScoping {
  public static void main(String[] args) {
    ScopeExample scope = new ScopeExample();
    scope.firstMethod();
  }
}
```



Variable Initialization

Variable	Value
boolean	false
char	'\u0000'
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
All reference types	null

Initialization Before Use Principle

- The compiler will verify that local variables have been initialized before used.

```
3      public void doComputation() {
4        int x = (int)(Math.random() * 100);
5        int y;
6        int z;
7        if (x > 50) {
8          y = 9;
9        }
10       z = y + x; // Possible use before initialization
11     }
```

```
javac TestInitBeforeUse.java
TestInitBeforeUse.java:10: variable y might not have been
initialized z = y + x;  // Possible use before initialization
^
1 error
```

Expressions

- An **expression** has **at minimum** **one operator**.

```
x + 5;        // Simple expressions have a single operator
x + 5 * y;    // Compound expressions have multiple operators
```

- The number of **operands** an **operator** has is determined by the operator.

```
x < 2;   // Binary operator example
++x;     // Unary operator example
```

- **An expression evaluates to a type**. The data type of an expression depends on the:
  - Operator
  - Data types of the operand(s)

Expression Evaluation Data Types

Expression	Operator Type	Operand Data Type	Result Type
x + y;	Numeric addition	Numeric	Numeric
x < 2;	Comparison	Numeric	`boolean`
“sun” + 22	String concatenation	At least one operand is a `string`	`String`
x & 22;	Bitwise AND	`int` or `long`	`int` or `long`

Binary Arithmetic Operators

- Suppose x = 7 and y = 3 then

Purpose	**Operator**	**Example**	**Result**
Addition	+	`result = x + y;`	10
Subtraction	-	`result = x - y;`	4
Multiplication	*	`result = x * y;`	21
Division	/	`result = x / y;`	2
Remainder	%	`result = x % y;`	1

Unary Arithmetic Operators

Purpose	Operator	Example	`num1` **Pre-Expression Evaluation**	`result`	`num1` **Post-Expression Evaluation**
Unary plus	`+`	`result = +num1;`	7	7	7
Unary minus	`-`	`result = -num1;`	7	-7	7
Pre-Increment	`++`	`result =++num1;`	7	8	8
Post-Increment	`++`	`result =num1++;`	7	7	8
Pre-Decrement	`--`	`Result=--num1;`	7	6	6
Post-Decrement	`--`	`result = num1--;`	7	7	6

Bitwise and Bitwise Shift Operators

Purpose	Operator	Usage Example
Bitwise complement	~	~x
Bitwise OR Bitwise AND Bitwise XOR	| & ^	x | y x & y x ^ y
Bitwise signed left shift Bitwise signed right shift Bitwise unsigned right shift	<< >> >>>	x << y x >> y x >>> y

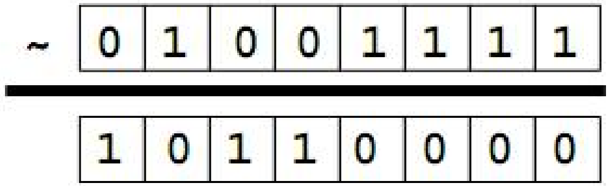
Bitwise Logical Operators

- The integer bitwise operators are:

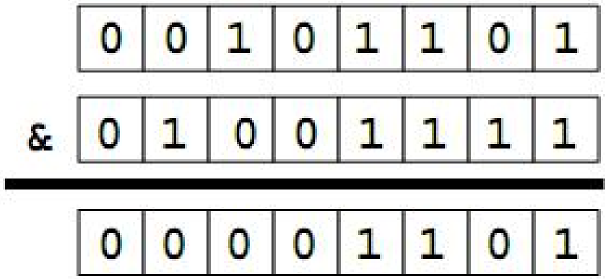
  ~ – Complement & – AND

  ^ – XOR | – OR

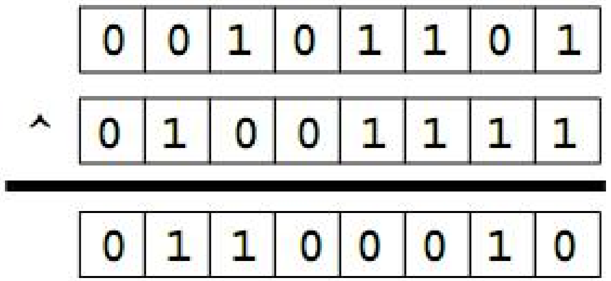
- • Byte-sized examples include:



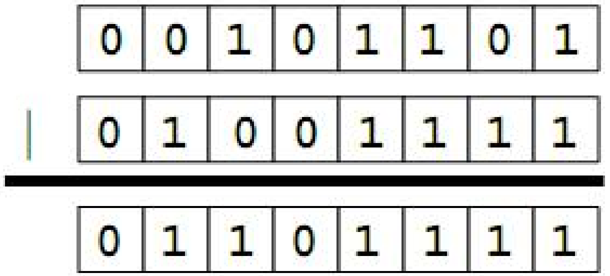
```
~ 0 1 0 0 1 1 1 1
  ---------------
  1 0 1 1 0 0 0 0
```



```
  0 0 1 0 1 1 0 1
& 0 1 0 0 1 1 1 1
  ---------------
  0 0 0 0 1 1 0 1
```



```
  0 0 1 0 1 1 0 1
^ 0 1 0 0 1 1 1 1
  ---------------
  0 1 1 0 0 0 1 0
```



```
  0 0 1 0 1 1 0 1
| 0 1 0 0 1 1 1 1
  ---------------
  0 1 1 0 1 1 1 1
```

Right-Shift Operators >> and >>>

- **Arithmetic** or **signed right shift** (>>) operator, Examples are:

  `128 >> 1` returns $128/2^1 = 64$

  `256 >> 4` returns $256/2^4 = 16$

  `-256 >> 4` returns $-256/2^4 = -16$

  - **The sign bit is copied during the shift.**

- **Logical** or **unsigned right-shift** (**>>>**) operator:
  - This operator is used for bit patterns.
  - The **sign bit is not copied** during the shift.

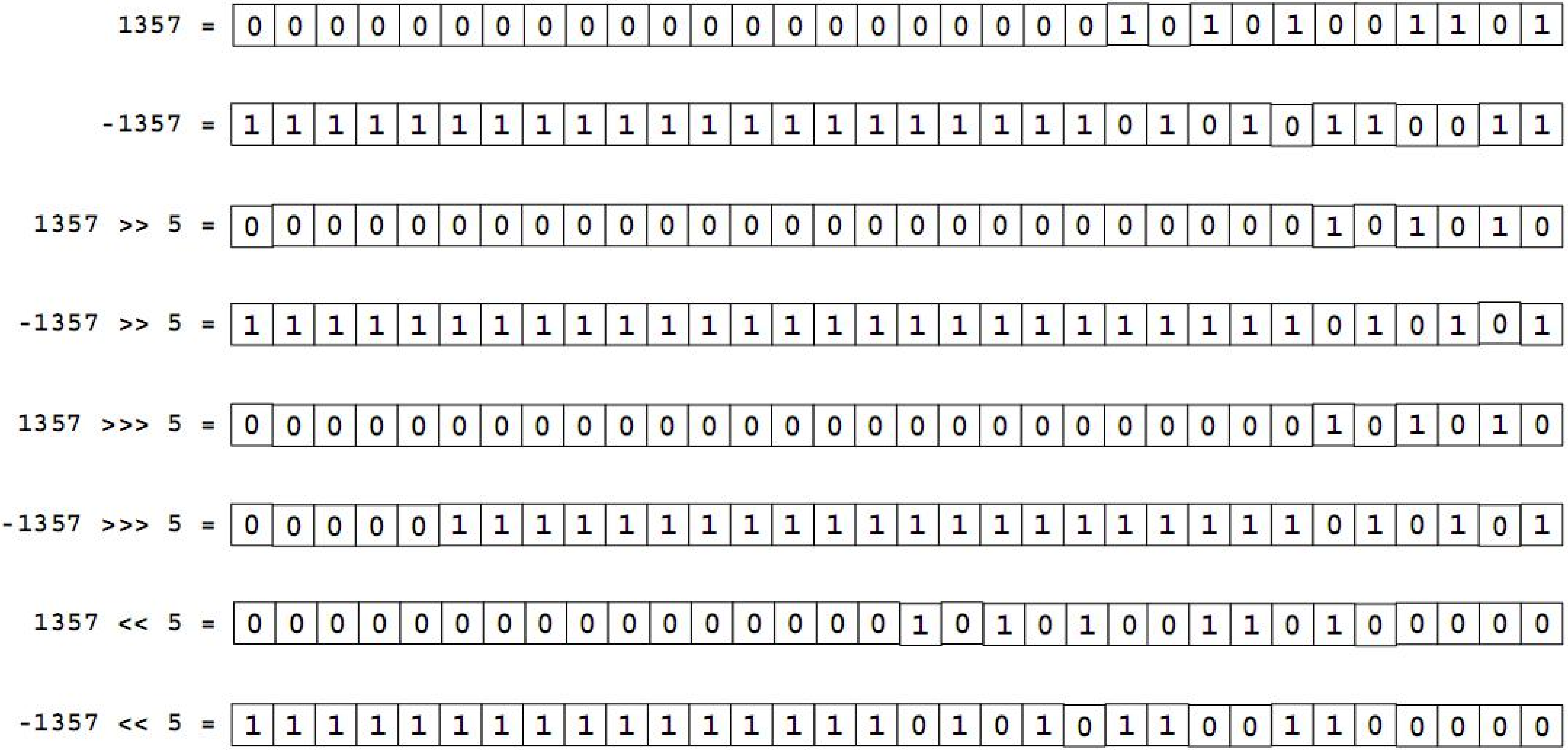
Left-Shift Operator <<

- Left-shift (<<) operator works as follows:

  128 << 1 returns 128 * 2^1 = 256

  16 << 2 returns 16 * 2^2 = 64

Shift Operator Examples



1357 =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	1	1	0	1
-1357 =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1	1	0	0	1	1
1357 >> 5 =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0
-1357 >> 5 =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1
1357 >>> 5 =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0
-1357 >>> 5 =	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1
1357 << 5 =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	1	1	0	1	0	0	0	0	0
-1357 << 5 =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1	1	0	0	1	1	0	0	0	0	0

Assignment Operators

Purpose	Operator	Example	Equivalent
Simple assignment	=	x = y;	
Compound arithmetic operation and assignment	+=	x += y;	x = x + y;
	-=	x -= y	x = x - y;
	*=	x *= y	x = x * y;
	/=	x /= y	x = x / y;
	%=	x %= y	x = x % y;
Compound **bitwise** logical operation and assignment	|=	x |= y;	x = x | y;
	&=	x &= y;	x = x & y;
	^=	x ^= y;	x = x ^ y;
Compound **bitwise shift** operation and assignment	<<=	x <<= y;	x = x << y;
	>>=	x >>= y;	x = x >> y;
	>>>=	x >>>= y;	x = x >>> y;

Logical Operators

- The **`boolean`** operators are:

  `!` – NOT **`&`** – AND

  `|` – OR **`^`** – XOR

- The short-circuit boolean operators are:

  `&&` – AND **`||`** – OR

- You can use these operators as follows:

```
MyDate d = reservation.getDepartureDate();
if ( (d != null) && (d.day > 31)) {
// do something with d
}
```

Relational Operators

Note: x=7, y=3 as above mentioned
a = false, b= true;

Purpose	**Operator**	**Example**	**Result**
Greater than	`>`	`result = x > 7;`	`false`
Greater than or equal to	`>=`	`result = x >= 7;`	`true`
Equal to	`==`	`result = x == y;`	`false`
Not equal to	`!=`	`result = x != y;`	`true`
Less than	`<`	`result = x < y;`	`false`
Less than or equal to	`<=`	`result = x <= y;`	`false`
Logical OR	`||`	`result = a || b;`	`true`
Logical AND	`&&`	`result = a && b;`	`false`
Logical OR	`||`	`result = (x<3) || (y>2);`	`true`
Logical AND	`&&`	`result = (x<3) && (y>2);`	`false`

String Concatenation With +

- The + operator works as follows:
  - Performs `String` **concatenation**
  - Produces a **new** String:

```
String salutation = "Dr.";
String name = "Pete" + " " + "Seymour";
String title = salutation + " " + name;
```

- One argument must be a `String` **object**.

- **Non-strings are converted to String objects automatically.**

Operator Precedence

- Operator precedence Operator associativity

Operators	Associative
++ -- **+ unary** **- unary** ~ ! (**<data_type>**)	R to L
* / %	L to R
+ -	L to R
<< >> >>>	L to R
< > <= >= instanceof	L to R
== !=	L to R
&	L to R
^	L to R
|	L to R
&&	L to R
||	L to R
<boolean_expr> ? <expr1> : <expr2>	R to L
= *= /= %= += -= <<= >>= >>>= &= ^= |=	R to L

Numeric Promotions in Simple Expressions

- **Numeric promotion** is the **conversion of data types of operands** to the result type, before the application of the operator. Applies to following operators:
  - Numeric operators

    `+      -      *      /      %`

  - Bitwise operators

    `&      |      ^`

  - Relational operators

    `<      <=      ==      !=      >      >=`

Numeric Promotions in Simple Expressions(Cont.)

Operand 1 Data Type	Operand 2 Data Type	Result
Byte, Character, Short,**byte, char, short**	Byte, Character, Short,**byte, char, or short**	**int**
Byte, Character, Short, Integerbyte, char, short, int	Integer or int	int
Byte, Character, Short, Integerbyte, char, short, int	Long or long	long
Byte, Character, Short, Integerbyte, char, short, int	Float or float	float
Byte, Character, Short, Integer, Floatbyte, char, short, int, float	Double or double	double

Casting

- If **information might be lost** in an assignment, the programmer must **confirm** the assignment **with a cast**.
- The assignment between `long` and `int` requires an **explicit cast**.

```
long bigValue = 99L;
int squashed = bigValue; // Wrong, needs a cast
int squashed = (int) bigValue; // OK

int squashed = 99L; // Wrong, needs a cast
int squashed = (int) 99L;     // OK, but...
int squashed = 99;  // default integer literal
```

Promotion and Casting of Expressions

- Variables are promoted **automatically** to a **longer form** (such as `int` to `long`).
- Expression is **assignment-compatible** if the variable type is at least as large (the same number of bits) as the expression type.

```
long bigval = 6;          // 6 is an int type, OK
int smallval = 99L;       // 99L is a long, illegal

double z = 12.414F;       // 12.414F is float, OK
float z1 = 12.414;        // 12.414 is double, illegal
```

- Casting is required when assigning a long form of a type to an equivalent short form.

```
short a, b, c;
a = 1;
b = 2;
c = a + b;  // ??     // illegal, should be c = (short)(a + b);
```

Simple `if`, `else` Statements

- The if statement syntax:

```
if ( <boolean_expression> )
     <statement_or_block>
```

- Example:

```
if ( x < 10 )
     System.out.println("Are you finished yet?");
```

or (*recommended*):

```
if ( x < 10 ) {
     System.out.println("Are you finished yet?");
}
```

Complex if, else Statements

- The if-else statement syntax:

```
if ( <boolean_expression> )
    <statement_or_block>
else
    <statement_or_block>
```

- Example:

```
if ( x < 10 ) {
    System.out.println("Are you finished yet?");
} else {
    System.out.println("Keep working...");
}
```

Complex `if`, `else` Statements(Cont.)

- The if-else-if statement syntax:

```
if ( <boolean_expression> )
     <statement_or_block>
else if ( <boolean_expression> )
     <statement_or_block>
```

- Example:

```
int count = getCount(); // a method defined in the class
if (count < 0) {
    System.out.println("Error: count value is negative.");
} else if (count > getMaxCount()) {
    System.out.println("Error: count value is too big.");
} else {
    System.out.println("There will be " + count + " people for lunch today.");
}
```

switch Statements

- The switch statement syntax:

```
switch ( <expression> ) {
case <constant1>:
    <statement_or_block>* [break;]
case <constant2>:
    <statement_or_block>* [break;]
default:
    <statement_or_block>* [break;]
}
```

Question: Where can we put the **default** statement?

Note: switch statement support **String** since JDK 7.0!

switch Statements(Cont.)

- A switch statement example:

```
switch ( carModel ) {
    case DELUXE:
        addAirConditioning();
        addRadio();
        addWheels();
        addEngine();
        break;
    case STANDARD:
        addRadio();
        addWheels();
        addEngine();
        break;
    default:
        addWheels();
        addEngine();
}
```

switch Statements(Cont.)

- This switch statement is equivalent to the previous example:

```
switch ( carModel ) {
    case DELUXE:
        addAirConditioning();
    case STANDARD:
        addRadio();
    default:
        addWheels();
        addEngine();
}
```

Without the break statements, the execution falls through each subsequent case clause.

Question:

1. How about placing the default statement at first??
2. What type of expression can be used in switch??

Looping Statements

- The for loop:

```
for( <init_expr>; <test_expr>; <alter_expr> )
    <statement_or_block>
```

- Example:

```
for ( int i = 0; i < 10; i++ )
    System.out.println(i + " squared is " + (i*i));
```

or (*recommended*):

```
for ( int i = 0; i < 10; i++ ) {
    System.out.println(i + " squared is " + (i*i));
}
```

Question: The loop variable i is valid in which scope??

Looping Statements(Cont.)

- The while loop:

```
while ( <test_expr> )
    <statement_or_block>
```

Example:

```
int i = 0;
while ( i < 10 ) {
    System.out.println(i + " squared is " + (i*i));
    i++;
}
```

What happens if i=20 ??

Looping Statements(Cont.)

- The do/while loop:

```
do
    <statement_or_block>
while ( <test_expr> );
```

Example:

```
int i = 0;
do {
    System.out.println(i + " squared is " + (i*i));
    i++;
} while ( i < 10 );
```

What happens if i=20 ??

Special Loop Flow Control

- The `break [<label>];` command
- The `continue [<label>];` command
- The `<label> : <statement>` command, where `<statement>` should be a loop

The `break` Statement

```
do {
  statement;
  if ( condition ) {
    break;
  }
  statement;
} while ( test_expr );
```

The continue Statement

```
do {
    statement;
    if ( condition ) {
        continue;
    }
    statement;
} while ( test_expr );
```

Question: Get the sum of those numbers from 1 to 100 that are not exactly divisible by 3 or 5??

Using `break` Statements with Labels

```
outer:
  do {
    statement1;
    do {
      statement2;
      if ( condition ) {
        break outer;
      }
      statement3;
    } while ( test_expr );
    statement4;
  } while ( test_expr );
  statement5;
```

Using `continue` Statements with Labels

```
test:
  do {
    statement1;
    do {
      statement2;
      if ( condition ) {
        continue test;
      }
      statement3;
    } while ( test_expr );
    statement4;
  } while ( test_expr );
  statement5;
```

Declaring Arrays

- **Group data objects of the same type.**
- **Declare** arrays of **primitive** or **class** types:

```
char s[];
Point p[];
char[] s;
Point[] p;
```

- **Create space for a reference.**
- **An array is an object**; it is created with **new**.

Creating Arrays

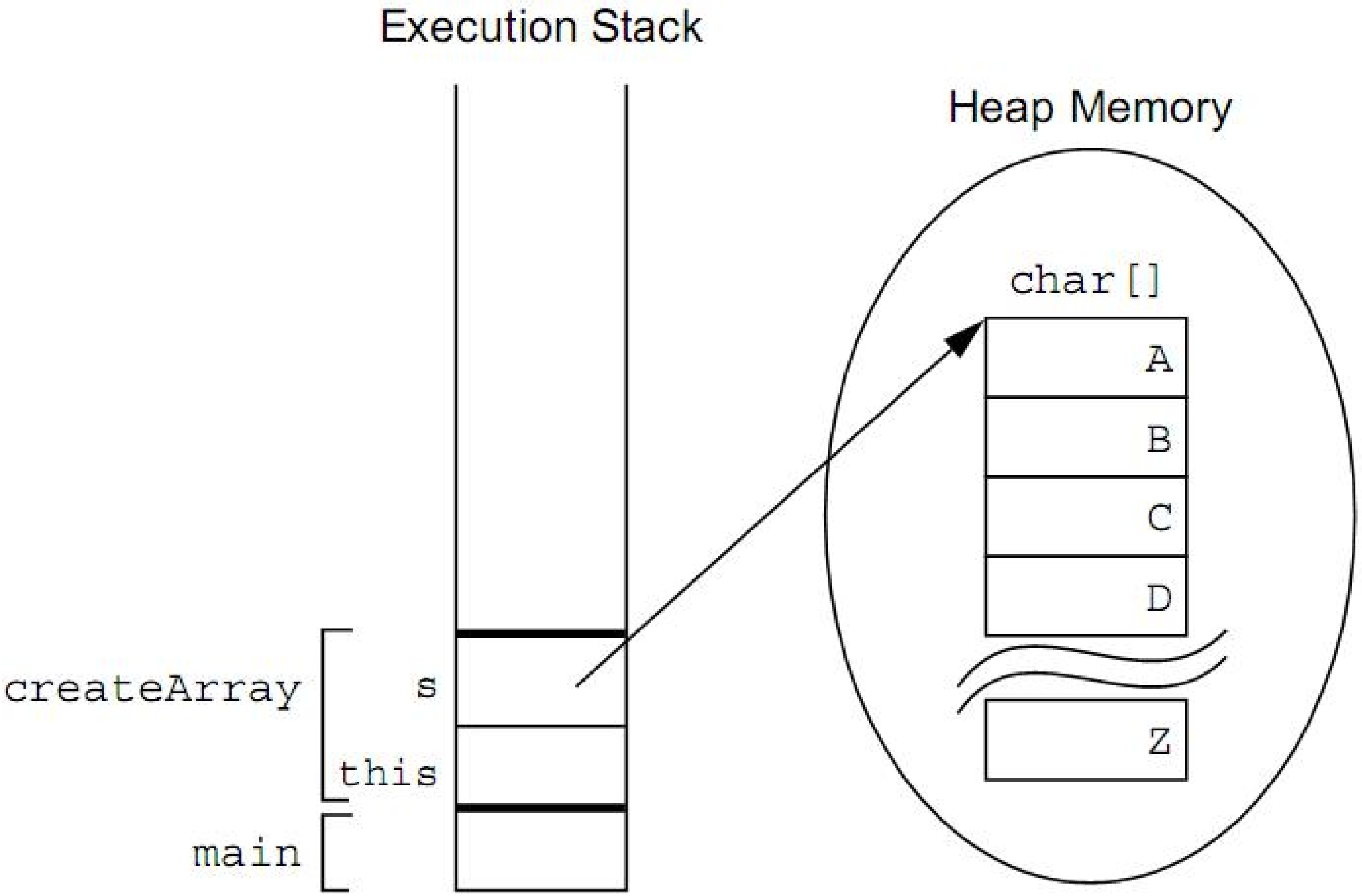
- Use the **new** keyword to create an array object. For example, a primitive (char) array:

```
public char[] createArray(){
  char[] s;

  s = new char[26];
  for ( int i=0; i<26; i++ ) {
    s[i] = (char) (’A’ + i);            ??
  }

  return s;
}
```

Creating an Array of Character Primitives



Creating Reference Arrays

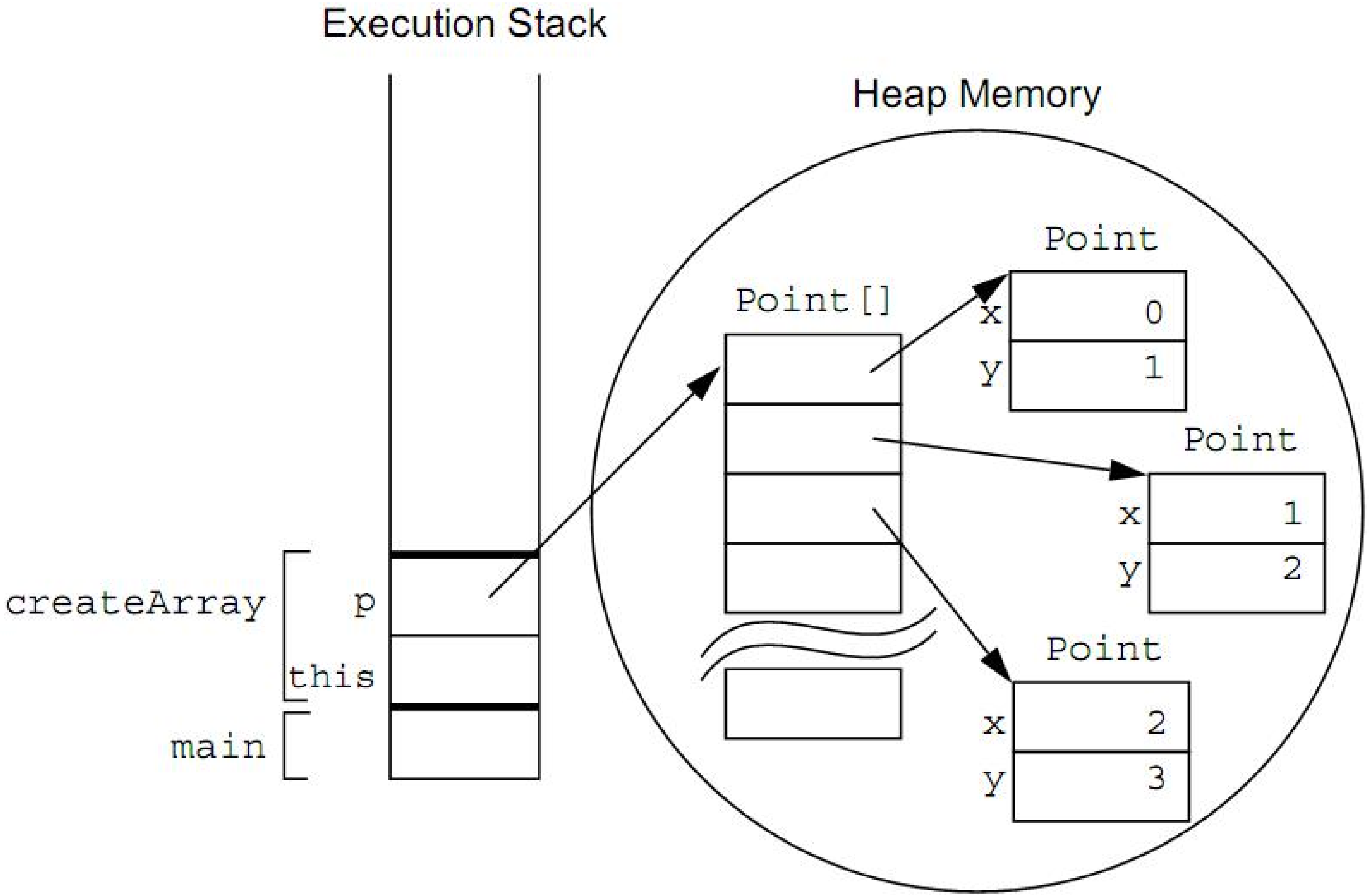
- Another example, an object array:

```
public Point[] createArray() {
  Point[] p;

  p = new Point[10];
  for ( int i=0; i<10; i++ ) {
    p[i] = new Point(i, i+1);
  }

  return p;
}
```

Creating an Array of Reference to Point Objects



Initializing Arrays

- Initialize an array element.
- Create an array with initial values.

```
String[] names;
names = new String[3];
names[0] = "Georgianna";
names[1] = "Jen";
names[2] = "Simon";
String[] names = { "Georgianna", "Jen", "Simon"};

MyDate[] dates;
dates = new MyDate[3];
dates[0] = new MyDate(22, 7, 1964);
dates[1] = new MyDate(1, 1, 2000);
dates[2] = new MyDate(22, 12, 1964);
MyDate[] dates = {new MyDate(22, 7, 1964), new MyDate(1, 1, 2000),
                  new MyDate(22, 12, 1964)}
```

Multidimensional Arrays

- Arrays of arrays:

```
int[][] twoDim = new int[4][];
twoDim[0] = new int[5];
twoDim[1] = new int[5];
int[][] twoDim = new int[][4]; // illegal
```

Non-rectangular arrays of arrays:

```
twoDim[0] = new int[2];
twoDim[1] = new int[4];
twoDim[2] = new int[6];
twoDim[3] = new int[8];
```

Array of four arrays of five integers each:

```
int[][] twoDim = new int[4][5];
```

Array Bounds

- All array subscripts begin at **0**:

```
public void printElements(int[] list){
  for (int i = 0; i < list.length; i++){
    System.out.println(list[i]);
  }
}
```

Using the Enhanced `for` Loop

- Java 2 Platform, Standard Edition (J2SE™) version 5.0 introduced an enhanced **for** loop for iterating over arrays:

```
public void printElements(int[] list) {
  for ( int element : list ) {
    System.out.println(element);
  }
}
```

- The for loop can be read as for each **element** in **list** do.

Array Resizing

- **You cannot resize an array.**
- You can use the same reference variable to **refer to an entirely new array**, such as:

```
int[] myArray = new int[6];
myArray = new int[10];
```

Copying Arrays

- The **System.arraycopy()** method to copy arrays is:

```
//original array
int[] myArray = { 1, 2, 3, 4, 5, 6 };

// new larger array
int[] hold = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };

// copy all of the myArray array to the hold
// array, starting with the 0th index
System.arraycopy(myArray, 0, hold, 0, myArray.length);
```

More to be found in class Arrays. Such as copyOf, copyOfRange, binarySearch, equals, fill, sort …

Questions or Comments?