

CSE1007

Java Basics – Arrays, Class and Methods

Keywords

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

*boolean literals

** null literal

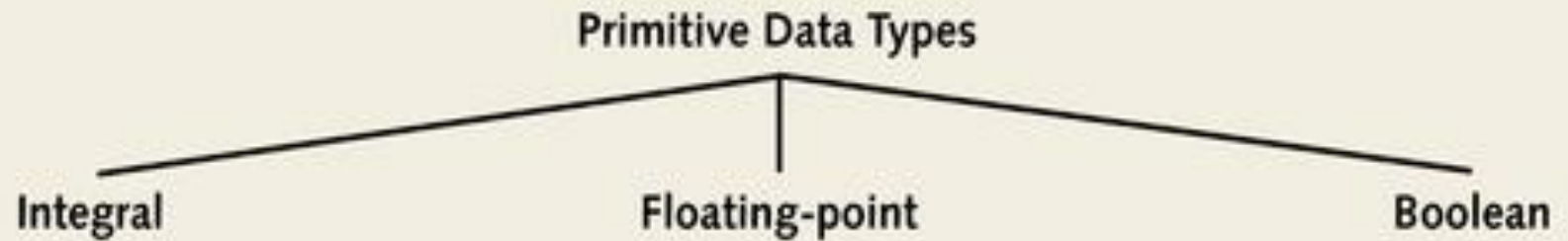
Identifiers

- Names of things
- Cannot be the same as any keyword
- Are case-sensitive
- Consist of and must begin with:
 - Letters
 - Digits
 - The underscore character (`_`)
 - The dollar sign (`$`)

Data Types

- **Data type:** set of values together with a set of operations

Primitive Data Types



Primitive Data Types

➤ Floating-Point Data Types

➤ **float**: precision = 6 or 7

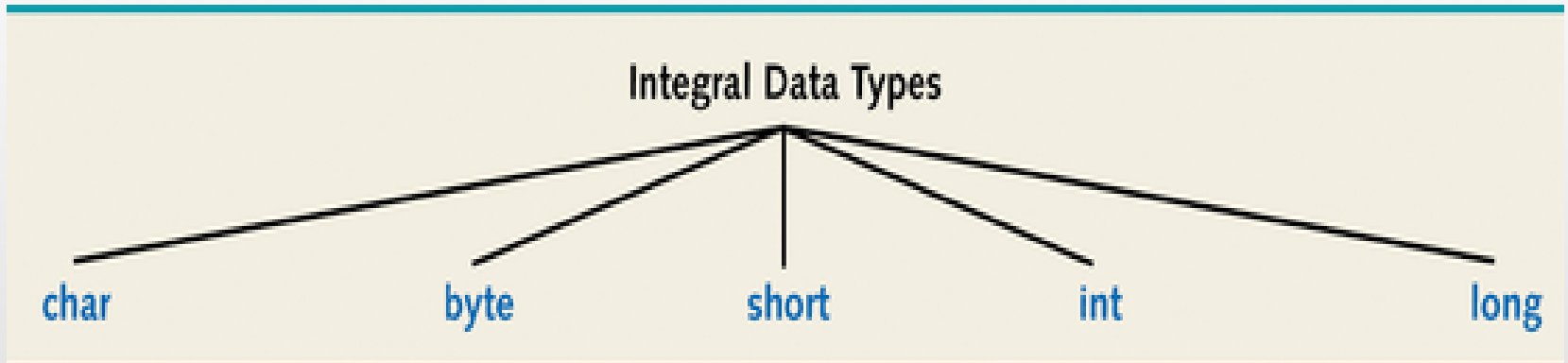
➤ **double**: precision = 15

➤ Boolean

➤ **boolean**

- true
- false

Integral Data Types



Values and Memory Allocation for Integral Data Types

Data Type	Values	Storage (in bytes)
char	0 to 65535	2 (16 bits)
byte	-128 to 127	1 (8 bits)
short	-32768 to 32767	2 (16 bits)
int	-2147483648 to 2147483647	4 (32 bits)
long	-9223372036854775808 to 9223372036854775807	8 (64 bits)

Literals

- Null literal

- null

- boolean

- true false

- int

- Default for integrals

- 2 0372 0xDadaCafe 1996 0x00FF00FF

- long

- Must end with “L” or “l”

- 0l 0777L 0x100000000L 2147483648L
0xC0B0L

Floating-Point Literals

➤ float

- Must end with “f” or “F”

- 1e1f 2.f .3f 0f 3.14F 6.022137e+23f

➤ double

- Default for floating-point types

- 1e1 2. .3 0.0 3.14 1e-9d 1e137

Character Literals

➤ char

➤ Must be in single quotes

➤ Escape character is \

- '\t' for tab, '\n' for newline, '\\' for '\', '\'' for "'", etc.
- If followed by numbers, represents the unicode ascii value of that character

➤ Examples:

- 'a' '%' '\t' '\\' '\"' '\u03a9' '\uFFFF' '\177'

➤ Cannot have more than one character

- 'abc' is not a valid character

Commonly Used Escape Sequences

	Escape Sequence	Description
<code>\n</code>	Newline	Insertion point moves to the beginning of the next line
<code>\t</code>	Tab	Insertion point moves to the next tab stop
<code>\b</code>	Backspace	Insertion point moves one space to the left
<code>\r</code>	Return	Insertion point moves to the beginning of the current line (<i>not</i> the next line)
<code>\\</code>	Backslash	Backslash is printed
<code>\'</code>	Single quotation	Single quotation mark is printed
<code>\"</code>	Double quotation	Double quotation mark is printed

String Literals

- Anything in double quotes
- Examples:
 - `""` // the empty string
 - `"\"` // a string containing " alone
 - `"This is a string"` // a string containing 16 characters
 - `"This is a " +` // a string-valued constant expression,
`"two-line string"` // formed from two string literals
- NOT Strings:
 - `'abc'`

Arithmetic Operators and Operator Precedence

- Five Arithmetic Operators
 - + addition
 - - subtraction
 - * multiplication
 - / division
 - % mod (modulus) operator (integer operands only)
- Unary operator: operator that has one operand
- Binary operator: operator that has two operands

Precedence of Operators

Operators	Precedence
!, +, - (unary operators)	first
*, /, %	second
+, -	third
<, <=, >=, >	fourth
==, !=	fifth
&&	sixth
	seventh
= (assignment operator)	last

Expressions

- Integral expressions
- Floating-point or decimal expressions
- Mixed expressions

Integral Expressions

➤ All operands are integers

➤ Examples:

$$2 + 3 * 5$$

$$3 + x - y/7$$

$$x + 2 * (y - z) + 18$$

Floating-point Expressions

- All operands are floating-point numbers
- Examples:

$12.8 * 17.5 - 34.50$

$x * 10.5 + y - 16.2$

Mixed Expressions

- Operands of different types
- Examples:
 - $2 + 3.5$
 - $6/4 + 3.9$
- Integer operands yield an integer result; floating-point numbers yield floating-point results
- If both types of operands are present, the result is a floating-point number
- Precedence rules are followed

Type Conversion (Casting)

- Used to avoid implicit type coercion
- Syntax
(dataTypeName) expression
- Expression evaluated first, then type converted to dataTypeName
- Examples:
 $(\text{int})(7.9 + 6.7) = 14$
 $(\text{int})(7.9) + (\text{int})(6.7) = 13$

The class String

- Used to manipulate strings
- String
 - Sequence of zero or more characters
 - Enclosed in double quotation marks
 - Null or empty strings have no characters
 - Numeric strings consist of integers or decimal numbers
 - `length()` returns the number of characters in string

The Wrapper Classes

- Class representations of primitive types
- One for each primitive type
 - int → Integer
 - double → Double
 - boolean → Boolean
 - float → Float
 - char → Character

Parsing Numeric Strings

- **String to int**

`Integer.parseInt(strExpression)`

- **String to float**

`Float.parseFloat(strExpression)`

- **String to double**

`Double.parseDouble(strExpression)`

***strExpression: expression containing a numeric string**

Constants

- Cannot be changed during program execution
- Declared by using the reserved word **final**
- Initialized when it is declared
- Should be in all uppercase with words separated by _ according to Sun's naming conventions

```
final int MIN_VALUE = 0;
```

```
final String MY_NAME = "Beth";
```


Variables

- Content may change during program execution
- Must be declared before it can be used
- May not be automatically initialized
 - Class level – initialized to default value
- If new value is assigned, old one is destroyed
- Value can only be changed by an assignment statement or an input (read) statement
- Sun's naming conventions:
 - Mixed case with a lowercase first letter. Internal words start with capital letters.
 - Variable names should not start with underscore _ or dollar sign
 - Should be short, yet meaningful
 - Examples: accountBalance, firstName, menuChoice.

Class Naming Conventions

- Nouns
- Mixed case with the first letter of each internal word capitalized.
- Try to keep your class names simple and descriptive.
- Use whole words-avoid acronyms and abbreviations.
- Examples:
 - Calculator, User, BankAccount, ClassRoster

Method Naming Conventions

- Verbs
- Mixed case with the first letter lowercase, with the first letter of each internal word capitalized.
- Examples:
 - `readLine()`, `getFirstName()`, `setDateOfBirth()`

Assignment Statements

- Use “=”
- Operands must be compatible
 - String s = “s”; // ok
 - char c = “s”; // Not ok
- Left hand operand must be large enough of a type to contain the result of the right hand operand.
 - int x = 5 + 3; // ok
 - double d = 5 + 3; // ok
 - int n = 8.0 + 5; // Not ok
 - float f = 8.0 + 5; // Not ok... why not?

Increment and Decrement Operators

➤ ++

- Increments the value of its operand by 1
- $x++$ is equivalent to $x = x + 1$

➤ --

- Decrements the value of its operand by 1
- $x--$ is equivalent to $x = x - 1$

➤ Syntax

- Pre-increment: $++\text{variable}$
- Post-increment: $\text{variable}++$
- Pre-decrement: $--\text{variable}$
- Post-decrement: $\text{variable}--$

More on Assignment Statements

➤ `variable = variable * (expression);`

is equivalent to

➤ `variable *= expression;`

Similarly,

➤ `variable = variable + (expression);`

is equivalent to:

➤ `variable += expression;`

Input

- Standard input stream object: `System.in`
- Input numeric data to program
 - Separate by blanks, lines, or tabs
- To read a line of characters:
 1. Create an input stream object of the **class** `BufferedReader`
 2. Use the method `readLine()`

Output

- Standard output object: `System.out`

- Methods

 - `print`

 - `println`

 - `flush`

- Syntax

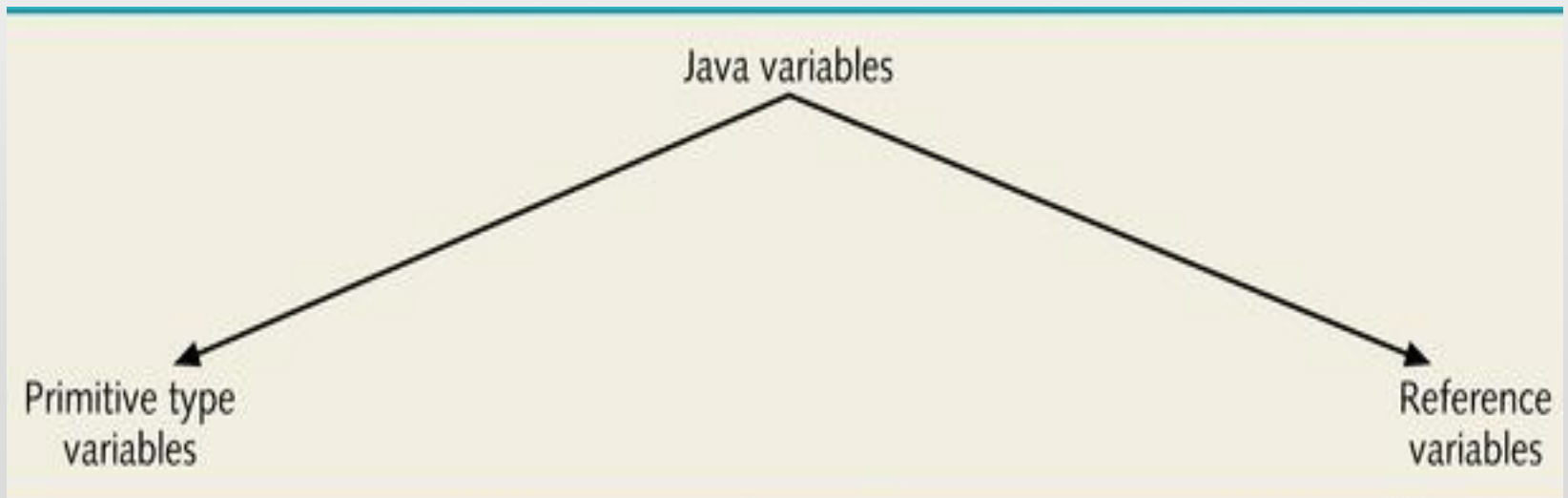
`System.out.print(stringExp);`

`System.out.println(stringExp);`

`System.out.flush();`

Object and Reference Variables

- Primitive variables: directly store data into their memory space
- Reference variables: store the address of the object containing the data



Reference Types

- Reference types include class and interface types.
- Runtime type - every object belongs to a class
 - Class mentioned in the creation statement that produced the object
 - The class whose Class object was used using reflection to create an instance of that class
 - Or the String class for objects created using the + operator

Reference Types (cont'd)

➤ Compile-time type

- The class or interface used when the variable was declared (Person p;)
- Limits the possible run-time types.
 - p can now only be assigned to an instance of a class compatible with Person
 - A class that inherits from Person
 - A class that implements the interface Person
 - An instance of the Person class
 - The null literal

Reference Types (cont'd)

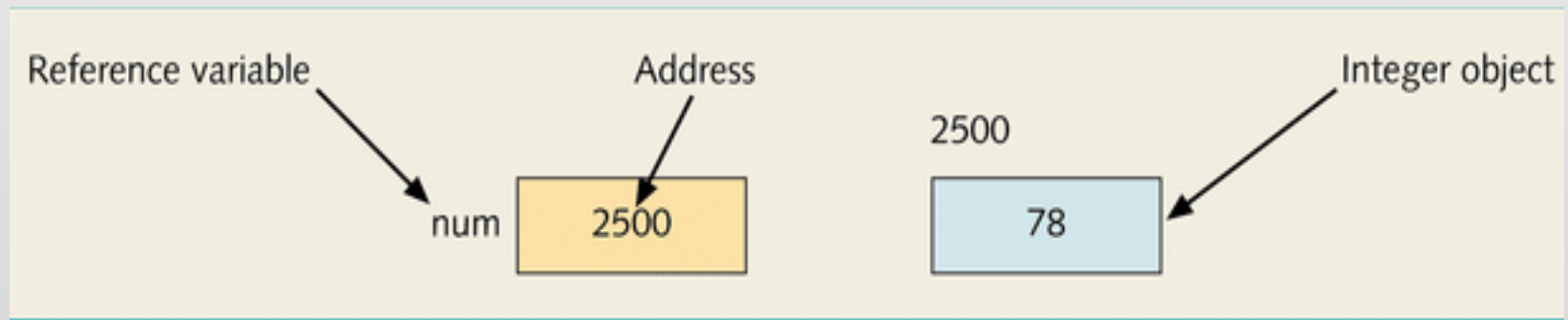
- The declaration creates the reference, but does not create an instance of the class.
 - String s does not reserve the memory to hold the String to which s will refer.
- Can have multiple references to the same object.
 - `Object o1 = new Object(); Object o2 = o1;`
 - Garbage collection occurs only when there are no more references to the created object.

Object and Reference Variables

- Declare a reference variable of a class type
- Use the operator new to:
 - Allocate memory space for data
 - Instantiate an object of that class type
- Store the address of the object in a reference variable

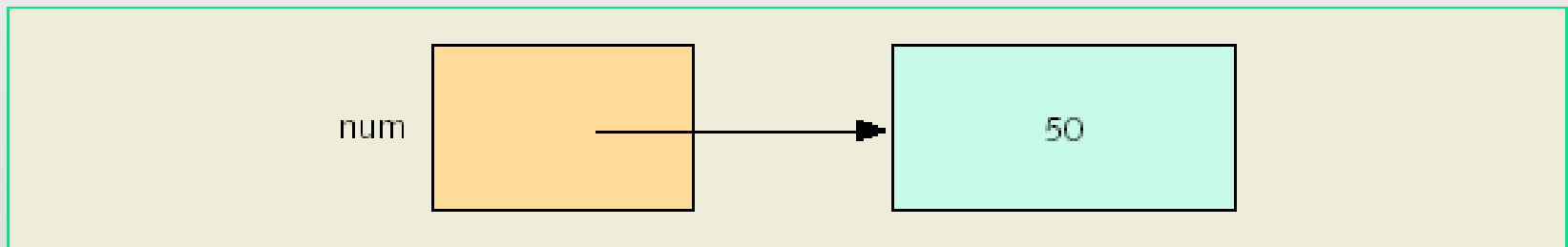
The Operator new

- Statement:
Integer num;
num = new Integer(78);
- Result:



Garbage Collection

- Change value of num:
`num = new Integer(50);`
- Old memory space reclaimed



Packages, Classes and Methods

Terminology

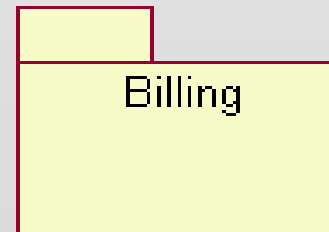
- **Package:** collection of related classes
- **Class:** consists of methods. All Java code *must* reside in a class.
- **Method:** designed to accomplish a specific task

Packages

- Files with a *.class* extension are aggregated into packages, or collections of related classes.
- Packages enforce modularity.
- Any class can belong to only 1 package.
- Packages may contain subpackages to arbitrary levels.
- The primary package is **java** and its main subpackage is **lang**.

Packages (cont'd)

- All classes in the `java.lang` package are automatically imported into every program.
- Programmers typically **import** classes from packages to avoid using fully qualified names.
 - If a program imports the class `java.util.Date`, the programmer then can use `Date` instead of the fully qualified name.
- In UML, they look like:



Packages (cont'd)

- **java** and the **javax** packages are standard packages.
 - The **javax** packages are extensions to the earlier **java** packages.
 - Support string and text processing, numeric computation, networking, graphics, security, and so on.

Packages (cont'd)

➤ Packages can be used to resolve name conflicts.

➤ The `java.awt` package has a `List` class and the `java.util` package has a `List` interface. The fully qualified names

`java.awt.List`

`java.util.List`

disambiguate.

Packages (cont'd)

- Every class belongs to a package.
- The package to which a class belongs can be explicitly declared with the **package** statement in a source file.
- A class belongs to a default unnamed package if a containing package is not explicitly declared.
- Sun's naming convention dictates that package names are in all lowercase letters.

package statement

➤ The package statement, if present, occurs as the first *uncommented* line in a source file.

➤ The source file *Hi.java* could begin

```
package hiPkg;    // Note: 1st line
import java.util.Date;
class Hi {
    ...
}
```

Summary of packages

- Convenient way to group related classes into software libraries.
 - Example: `java.math`
- For small programs and projects, default packages are typically sufficient.
 - Always use packages if you ever expect to reuse code again.
- Explicitly named packages are especially useful for large projects with many programmers.

What is an object?

- A representation of almost anything you need to model in a program.
- The basic unit of object orientation.
- Has attributes, behavior and identity.
- Members of a class.
- Attributes and behaviors are defined by the class definition.

What is a class? (part 1)

- A description or definition of a set of objects, that share common features.
- A class has attributes, behaviors, methods, and state.

ClassName
<code>-privateAttr : type</code> <code>#protectedAttr : type</code> <code>+publicAttr : type</code>
<code>+methodA(args) : returnType</code>

What is a class? (part 2)

- **Attributes** – Hold state information of an object. These should not be directly accessible to the outside world.
- **Behavior** – Activity of an object that is visible to the outside world.
- **Method** – Member function defined as part of the declaration of a class.
- **State** – Reflects the current values of all attributes of a given object and is the result of the behavior of an object over time.

Class and abstract data type

- The class construct in object-oriented languages directly supports *abstract data types*—data types defined by high-level operations rather than by low-level implementation details.
- Use of abstract data rather than primitive data types eases programming tasks.
- As long as there is an API, programmers do not need to know the inner workings of a class.

Stack as an abstract data type

- A *Stack* is an example of an abstract data type. A *Stack* is a list with insertions and deletions done at the same end, known as the *top*. Its high-level operations include
 - *push*, which inserts an item onto the *Stack*.
 - *pop*, which removes the *Stack*'s top item.
 - *peek*, which shows the *Stack*'s top item without removing it.

Stack class

- A **Stack** class type would encapsulate public methods such as **push**, **pop**, and **peek** to represent high-level *Stack* operations.
- Other methods such as **isEmpty** and **isFull** could be included to test whether a *Stack* is empty or full.
- Constructors would be provided to construct **Stack** instances.

Abstract Data Types

- **Expose** to clients a high-level *interface* that specifies the type's behavior.
 - In the *Stack* example, the interface consists of methods such as *push* and *pop* that specify high-level operations on a *Stack*.
- **Hide** from clients the type's low-level, implementation details.
 - In the *Stack* example, we don't know if an array is used to store the data elements, and we shouldn't need to know.

import Statement

- Used to import the components of a package into a program
- Reserved word
- **import** java.io.*;
imports the (components of the) **package** java.io into the program
- Primitive data types and the **class** String
 - Part of the Java language
 - Don't need to be imported

Do I have to **import** ?

- Java programs often include **import** statements such as

```
import java.util.Date;
```

for convenience.

- If a program requires the standard **Date** class, for instance, then the fully qualified name **java.util.Date** must be used if the **import** is omitted.
- Subpackages must be explicitly imported.

First Program Revisited

```
package hiPkg;  // rest of line is comment

/* comment that spans multiple
   lines */
import java.util.Date;
public class Hi {

    public static void main(String [] args) {
        System.out.print("Hi!");
    }
}
```

Running the program

- As stated before, the entry point of the program is the main method
 - If the signature of the main method is not “public static void main(String [] args)”, then it may compile, but it will not run
- To compile
 - javac Hi.java
- To run
 - java Hi

Using Predefined Classes and Methods in a Program

- To use a method you must know:
 - Name of class containing method (Math)
 - Name of package containing class (java.lang)
 - Name of method (pow), its parameters (int a, int b), and function (a^b)

class Math (Package: java.lang)

class Math (Package: java.lang)

Expression	Description
Math.abs(x)	Returns the absolute value of x. If x is of the type int , it returns a value of the type int ; if x is of the type long , it returns a value of the type long ; if x is of the type float , it returns a value of the type float ; if x is of the type double , it returns a value of the type double . Example: Math.abs(-67) returns the value 67. Math.abs(35) returns the value 35. Math.abs(-75.38) returns the value 75.38.
Math.ceil(x)	x is of the type double . Returns a value of the type double , which is the smallest integer value that is not less than x. Example: Math.ceil(56.34) returns the value 57.0.
Math.exp(x)	x is of the type double . Returns e^x , value of the type double , where e is approximately 2.7182818284590455. Example: Math.exp(3) returns the value 20.085536923187668.
Math.floor(x)	x is of the type double . Returns a value of the type double , which is the largest integer value less than x. Example: Math.floor(65.78) returns the value 65.0.
Math.log(x)	x is of the type double . Returns a value of the type double , which is the natural logarithm of x. Example: Math.log(2) returns the value 0.6931471805599453.
Math.max(x, y)	Returns the larger of x and y. If x and y are of the type int , it returns a value of the type int ; if x and y are of the type long , it returns a value of the type long ; if x and y are of the type float , it returns a value of the type float ; if x and y are of the type double , it returns a value of the type double . Example: Math.max(15, 25) returns the value 25. Math.max(23.67, 14.28) returns the value 23.67. Math.max(45, 23.78) returns the value 45.00.

class Math (Package: java.lang)

Expression	Description
<code>Math.min(x, y)</code>	Returns the smaller of x and y. If x and y are of the type <code>int</code> , it returns a value of the type <code>int</code> ; if x and y are of the type <code>long</code> , it returns a value of the type <code>long</code> ; if x and y are of the type <code>float</code> , it returns a value of the type <code>float</code> ; if x and y are of the type <code>double</code> , it returns a value of the type <code>double</code> . Example: <code>Math.min(15, 25)</code> returns the value 15. <code>Math.min(23.67, 14.28)</code> returns the value 14.28. <code>Math.min(12, 34.78)</code> returns the value 12.00.
<code>Math.pow(x, y)</code>	x and y are of the type <code>double</code> . Returns a value of the type <code>double</code> , which is x^y . Example: <code>Math.pow(2.0, 3.0)</code> returns the value 8.0. <code>Math.pow(4, 0.5)</code> returns the value 2.0.
<code>Math.round(x)</code>	Returns a value which is the integer closest to x. If x is of the type <code>float</code> , it returns a value of the type <code>int</code> ; if x is of the type <code>long</code> , it returns a value of the type <code>double</code> . Example: <code>Math.round(24.56)</code> returns the value 25. <code>Math.round(18.35)</code> returns the value 18.
<code>Math.sqrt(x)</code>	x is of the type <code>double</code> . Returns a value of the type <code>double</code> , which is the square root of x. Example: <code>Math.sqrt(4.0)</code> returns the value 2.0. <code>Math.sqrt(2.25)</code> returns the value 1.5.
<code>Math.cos(x)</code>	x is of the type <code>double</code> . Returns the cosine of x measured in radians. Example: <code>Math.cos(0)</code> returns the value 1.0.
<code>Math.sin(x)</code>	x is of the type <code>double</code> . Returns the sine of x measured in radians. Example: <code>Math.sin(0)</code> returns the value 0.0.
<code>Math.tan(x)</code>	x is of the type <code>double</code> . Returns the tangent of x measured in radians. Example: <code>Math.tan(0)</code> returns the value 0.0.

Using Predefined Classes and Methods in a Program

- Example method call:

`Math.pow(2,3);` //calls pow method in class Math

- (Dot) . Operator: used to access the method in the class

The Class String Revisited

- String variables are reference variables

- Given String name;

 - Equivalent Statements:

 - `name = new String("Lisa Johnson");`

 - `name = "Lisa Johnson";`

The Class String Revisited

- The String object is an instance of class `String`
- The value “Lisa Johnson” is instantiated
- The address of the value is stored in `name`
- The `new` operator is unnecessary when instantiating Java strings
- String methods are called using the dot operator

Commonly Used String Methods

- `String(String str)`
- `char charAt(int index)`
- `int indexOf(char ch)`
- `int indexOf(String str, int pos)`
- `int compareTo(String str)`

Commonly Used String Methods

- `String concat(String str)`
- `boolean equals(String str)`
- `int length()`
- `String replace(char ToBeReplaced, char ReplacedWith)`
- `String toLowerCase()`
- `String toUpperCase()`

Using Dialog Boxes for Input/Output

- Use a graphical user interface (GUI)
- class JOptionPane
 - Contained in package javax.swing
 - Contains methods: showInputDialog and showMessageDialog
- Syntax:

```
str = JOptionPane.showInputDialog(strExpression)
```

- Program must end with `System.exit(0);`

Parameters for the Method showMessageDialog

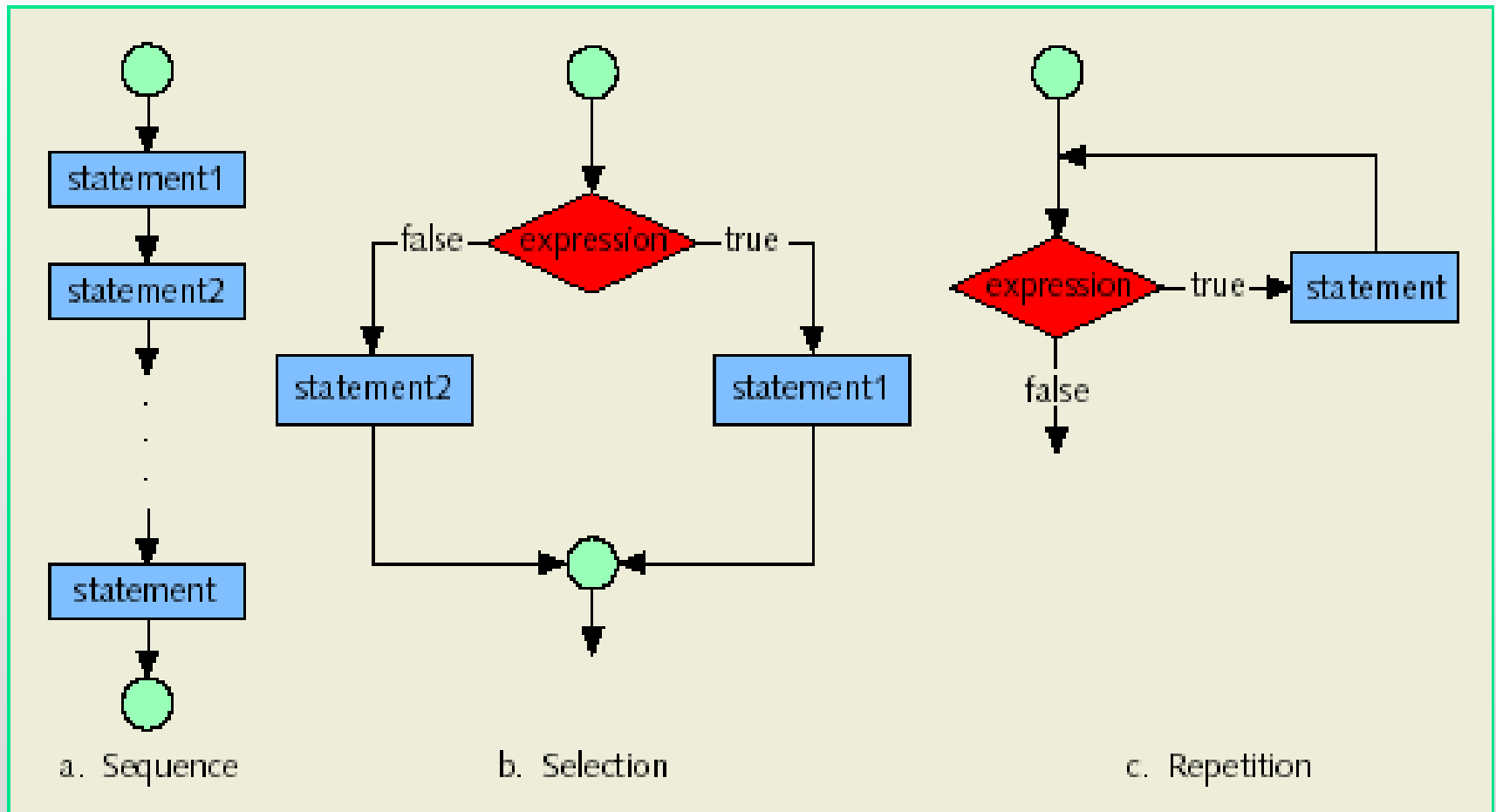
**JOptionPane.showMessageDialog(parentComponent,
messageStringExpression, boxTitleString, messageType);**

Parameter	Description
parentComponent	This is an object that represents the parent of the dialog box. For now, we will specify the parentComponent to be null , in which case the program uses a default component that causes the dialog box to appear in the middle of the screen. Note that null is a reserved word in Java.
messageStringExpression	The messageStringExpression is evaluated and its value appears in the dialog box.
boxTitleString	The boxTitleString represents the title of the dialog box.
messageType	An int value representing the type of icon that will appear in the dialog box. Alternatively, you can use certain JOptionPane options described below.

Control Structures

- Three methods of processing a program
 - In sequence
 - Branching or Selection
 - Looping
- Branch: Altering the flow of program execution by making a selection or choice
- Loop: Altering the flow of program execution by repetition of statement(s)

Flow of Execution



Relational Operators

- Relational Operator
 - Allows you to make comparisons in a program
 - Binary operator
- Condition is represented by a logical expression in Java
- Logical expression: expression that has a value of either true or false

Relational Operators in Java

Operator	Description
<code>==</code>	equal to
<code>!=</code>	not equal to
<code><</code>	less than
<code><=</code>	less than or equal to
<code>></code>	greater than
<code>>=</code>	greater than or equal to

Relational Operators and Primitive Data Types

- Can be used with integral and floating-point data types
- Can be used with the char data type
 - Unicode Collating Sequence

Relational Operators and the Unicode Collating Sequence

Expression	Value of the Expression	Explanation
' ' < 'a'	true	The Unicode value of ' ' is 32, and the Unicode value of 'a' is 97. Because 32 < 97 is true, it follows that ' ' < 'a' is true.
'R' > 'T'	false	The Unicode value of 'R' is 82, and the Unicode value of 'T' is 84. Because 82 > 84 is false, it follows that 'R' > 'T' is false.
'+' < '*'	false	The Unicode value of '+' is 43, and the Unicode value of '*' is 42. Because 43 < 42 is false, it follows that '+' < '*' is false.
'6' <= '>'	true	The Unicode value of '6' is 54, and the Unicode value of '>' is 62. Because 54 <= 62 is true, it follows that '6' <= '>' is true.

Comparing Strings

- class String

- Method compareTo

- Method equals

- Given string str1 and str2

integer < 0 if str1 < str2

Str1.compareTo(str2) = { 0 if str1 = str2

integer > 0 if str1 > str2

Logical (Boolean) Operators

Operator	Description
<code>!</code>	not
<code>&&</code>	and
<code> </code>	or

The ! (not) Operator

Expression	!(Expression)
<code>true</code>	<code>false</code>
<code>false</code>	<code>true</code>

AND and OR Operators

Expression1	Expression2	Expression1 && Expression2
true	true	true
true	false	false
false	true	false
false	false	false

Expression1	Expression2	Expression1 Expression2
true	true	true
true	false	true
false	true	true
false	false	false

Short-Circuit Evaluation

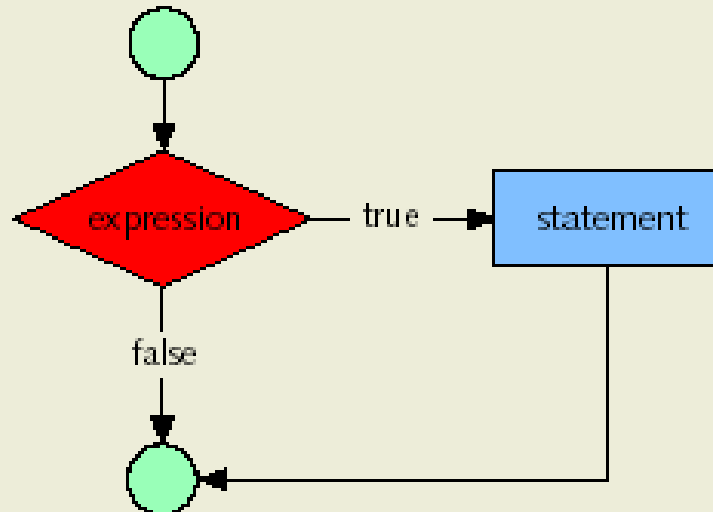
- Definition: a process in which the computer evaluates a logical expression from left to right and stops as soon as the value of the expression is known

Selection

- One-Way Selection
- Two-Way Selection
- Compound (Block of) Statements
- Multiple Selections (Nested if)
- Conditional Operator
- switch Structures

One-Way Selection

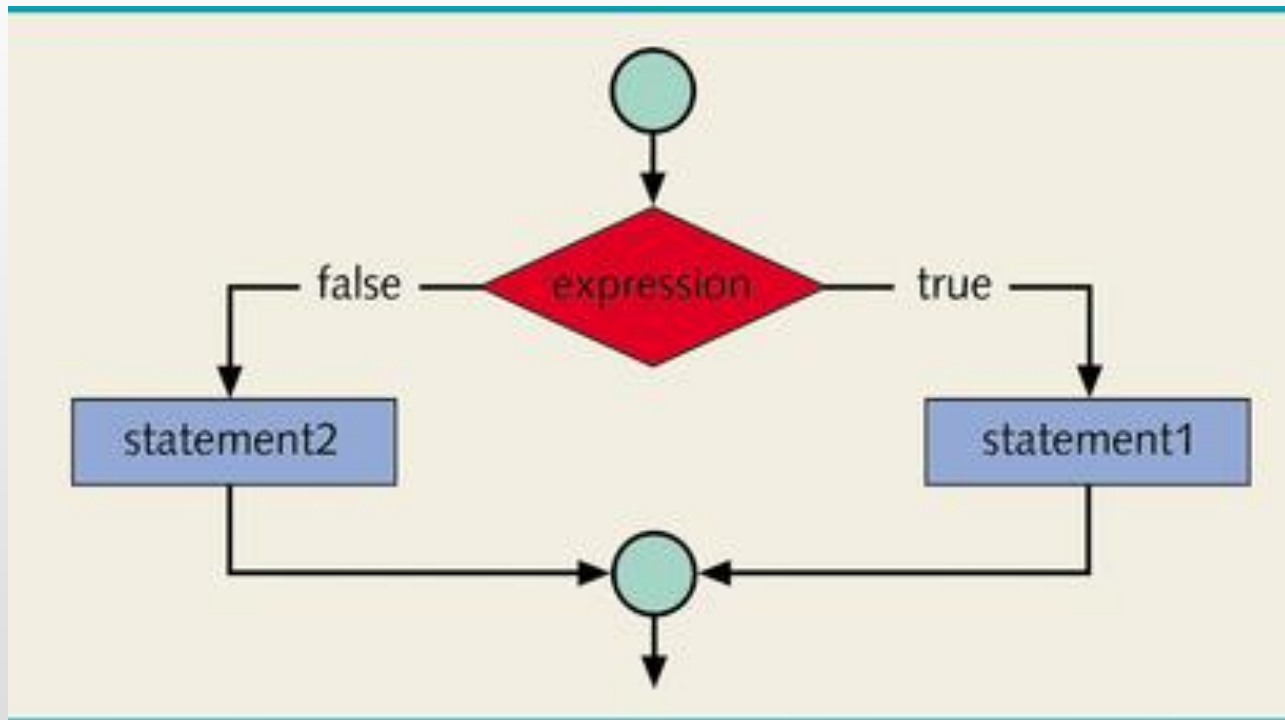
- Syntax: if(expression)
 statement
- Expression referred to as decision maker
- Statement referred to as action statement



Two-Way Selection

- Syntax: if(expression)
 statement1
 else
 statement2
- else statement must be paired with an if

Two-Way Selection



Compound (Block of) Statements

➤ Syntax

```
{  
    statement1  
    statement2  
    .  
    .  
    .  
    statement $n$   
}
```

Multiple Selection: Nested if

➤ Syntax

```
if(expression1)
    statement1
else
    if(expression2)
        statement2
    else
        statement3
```

- Else associated with most recent incomplete if
- Multiple if statements can be used in place of if...else statements
- May take longer to evaluate

Conditional (? :) Operator

- Ternary operator

- Syntax

expression1 ? expression2 : expression3

- If expression1 = true, then the result of the condition is expression 2

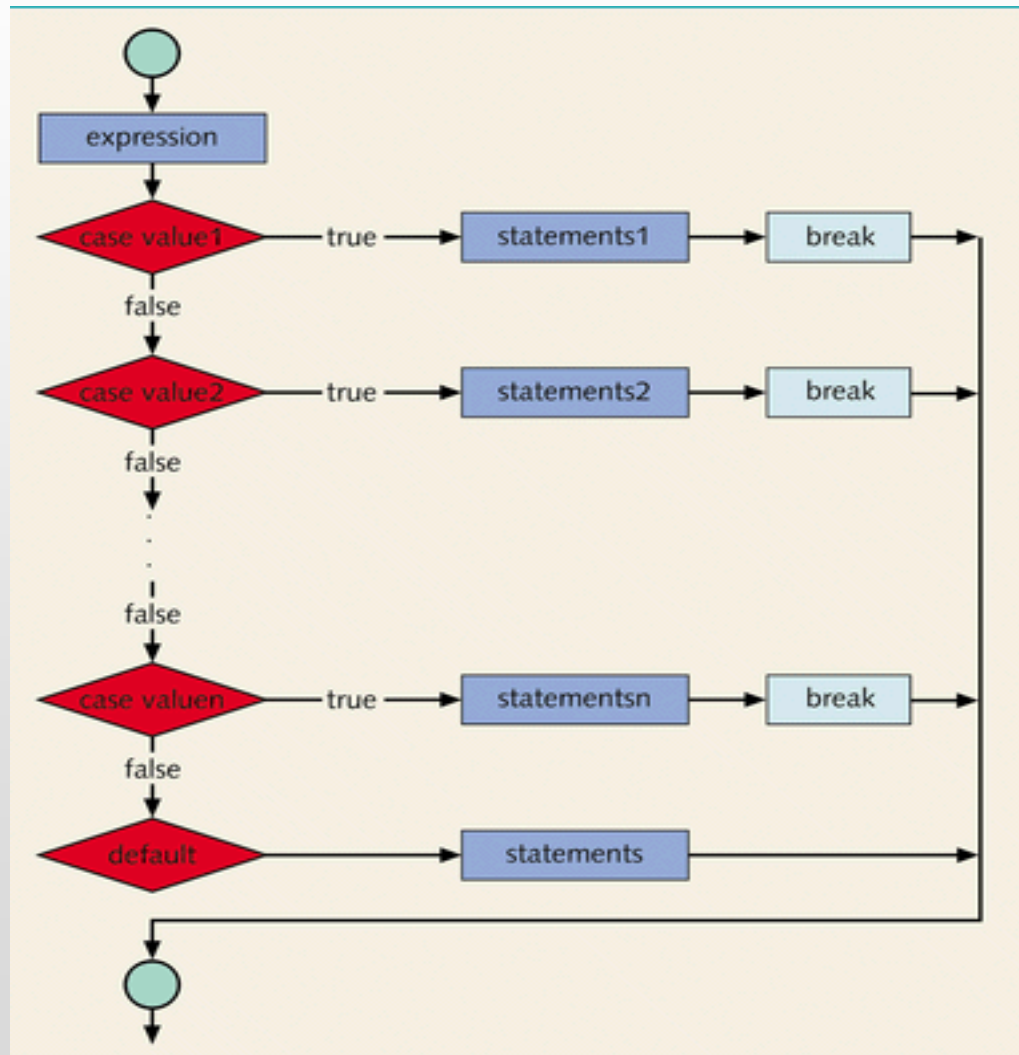
otherwise, the result of the condition is expression3

switch Structures

```
switch(expression)
{
case value1: statements1
    break;
case value2: statements2
    break;
    ...
case valuen: statementsn
    break;
default: statements
}
```

- Expression also known as selector
- Expression can be identifier
- Value can only be integral

switch Statement

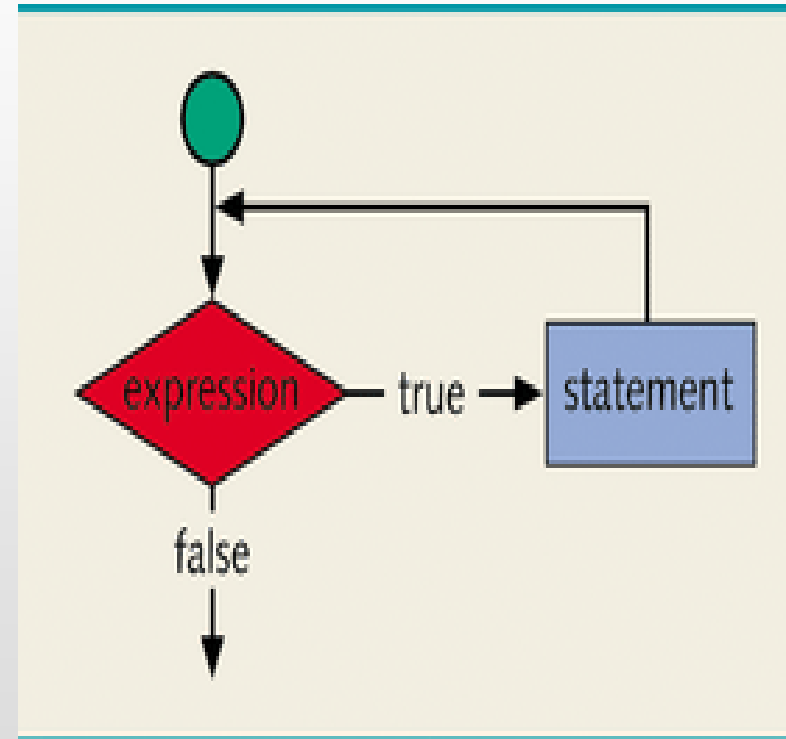


Why Is Repetition Needed?

- There are many situations in which the same statements need to be executed several times
- Example
 - Formulas used to find average grades for students in a class

The while Looping (Repetition) Structure

- Syntax
while(expression)
 statement
- Expression is always true in an infinite loop
- Statements must change value of expression to false



Counter-Controlled while Loop

➤ Used when exact number of data or entry pieces is known

➤ Syntax:

```
int N = //value input by user or specified in  
program;
```

```
int counter = 0;
```

```
while(counter < N){
```

```
    statement(s);
```

```
    counter++;
```

```
}
```

Sentinel-Controlled while Loop

➤ Used when exact number of entry pieces is unknown but last entry (special/sentinel value) is known)

➤ Syntax:

```
input first data item into variable;  
while(variable != sentinel){  
    statement(s);  
    input a data item into variable;  
}
```

Flag-Controlled while Loop

➤ Boolean value used to control loop

➤ Syntax:

```
boolean found = false;  
while(!found){  
    statement(s);  
    if(expression)  
        found = true;  
}
```

EOF(End of File)-Controlled while Loop

- Used when input is from files
- Sentinel value is not always appropriate
- Syntax:

```
input the first data item;  
while(not_end_of_input_file){  
    statement(s);  
    input a data item;  
}
```

- read method with value `-1` can be used as expression in while

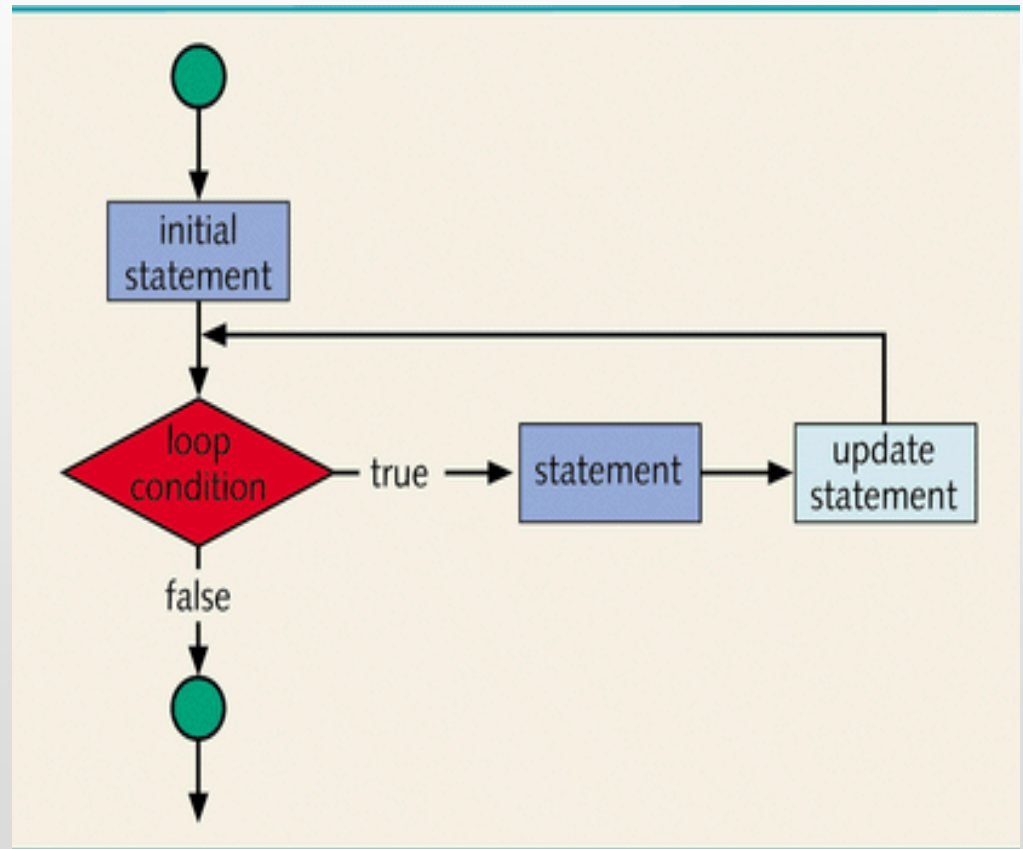
The for Looping (Repetition) Structure

- Specialized form of while loop
- Simplifies the writing of count-controlled loops
- Syntax

```
for(initial statement; loop condition; update  
    statement)  
{  
    statement(s);  
}
```

The for Looping (Repetition) Structure

- Execution:
 - initial statement executes
 - loop condition evaluated
 - If loop condition evaluates to true, execute for loop statement and execute update statement
 - Repeat until loop condition is false



The for Looping (Repetition) Structure

- Does not execute if initial condition is false
- Update expression changes value of loop control variable, eventually making it false
- If loop condition is always true, result is an infinite loop
- Infinite loop can be specified by omitting all three control statements
- If loop condition is omitted, it is assumed to be true
- for statement ending in semicolon is empty; does not effect program

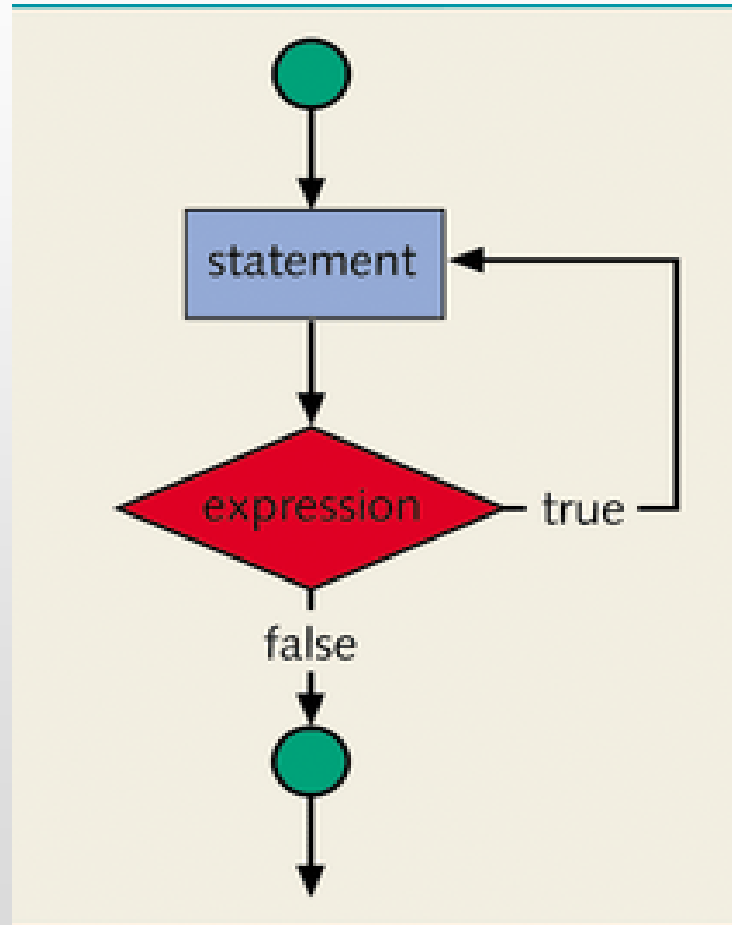
The do...while Loop (Repetition) Structure

➤ Syntax

```
do{  
    statement(s);  
}  
while(expression);
```

- Statements executed first, then expression evaluated
- Statement(s) executed at least once then continued if expression is true

do...while Loop (Post-test Loop)



break Statements

- Used to exit early from a loop
- Used to skip remainder of switch structure
- Can be placed within if statement of a loop
 - If condition is met, loop exited immediately

continue Statements

- Used in while, for, and do...while structures
- When executed in a loop, the remaining statements in the loop are skipped; proceeds with the next iteration of the loop
- When executed in a while/do...while structure, expression evaluated immediately after continue statement
- In a for structure, the update statement is executed after the continue statement; then the loop condition executes

Nested Control Structures

- Provides new power, subtlety, and complexity
- if, if...else, and switch structures can be placed within while loops
- for loops can be found within other for loops

Nested Control Structures (Example)

```
➤ for(int i = 1; i <= 5; i++){  
    for(int j = 1; j <= i; j++){  
        System.out.print("*");  
        System.out.println();  
    }  
}
```

➤ Output:

```
*  
**  
***  
****  
*****
```

Programming example: do-while and nested loops

- Write a program that will prompt the user for the number of times to say Hello.
 - Read in the user's input and output "Hello" n times (where n is the number of time specified by the user).
- At the end of the program, ask the user if he/she would like to run the program again...
 - If yes, prompt the user again for the number of times
 - If no, exit.

Array

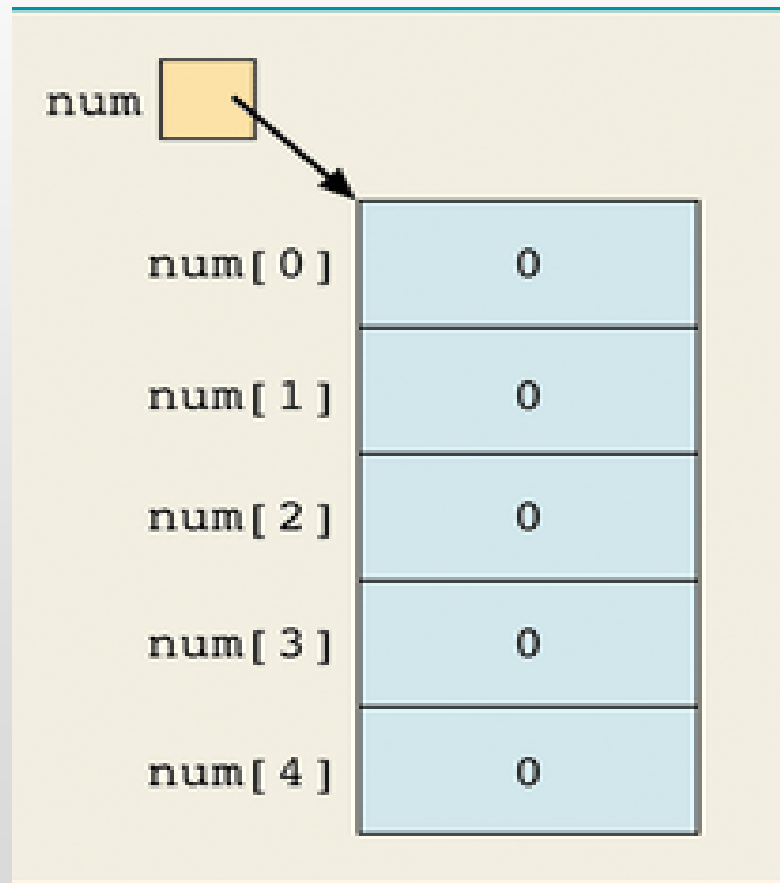
- **Definition:** structured data type with a fixed number of components
- Every component is of the same type
- Components are accessed using their relative positions in the array

One-Dimensional Arrays

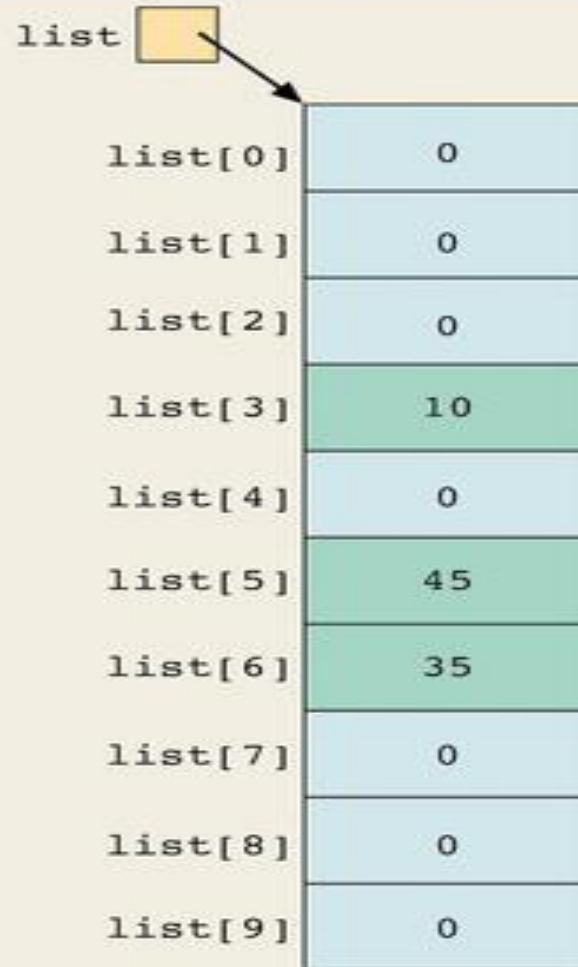
- Syntax to instantiate an array:
 - `dataType[] arrayName;`
`arrayName = new dataType[intExp]`
 - `dataType[] arrayName = new dataType[intExp]`
 - `dataType[] arrayName1, arrayName2;`
- Syntax to access an array component:
 - `arrayName[indexExp]`
 - `intExp` = number of components in array ≥ 0
 - $0 \leq \text{indexExp} \leq \text{intExp}$

Array num:

```
int[ ] num = new int [5];
```



Array list



<code>list</code>	
<code>list[0]</code>	0
<code>list[1]</code>	0
<code>list[2]</code>	0
<code>list[3]</code>	10
<code>list[4]</code>	0
<code>list[5]</code>	45
<code>list[6]</code>	35
<code>list[7]</code>	0
<code>list[8]</code>	0
<code>list[9]</code>	0

Figure 9-4 Array `list` after execution of the statements `list[3] = 10;`, `list[6] = 35;`, and `list[5] = list[3] + list[6];`

Arrays

- Not necessary to know array size at compile time
- `arrayName.length` returns the number of components in array
- Loops used to step through elements in array and perform operations
- Elements initialized to default value
 - 0 for integrals
 - 0.0 for floating point
 - null for object references

Arrays

- Some operations on arrays:
 - Initialize
 - Input data
 - Output stored data
 - Find largest/smallest/sum/average of elements

How To Specify Array Size During Program Execution

```
int arraySize; //Line 1
System.out.print("Enter the size of the array: "); //Line 2
arraySize = Integer.parseInt(keyboard.readLine()); //Line 3
System.out.println(); //Line 4

int[] list = new int[arraySize]; //Line 5
```

Instance Variable length

- Contains size of array
- public member
- Is final
- Can be directly accessed in program using array name and dot operator
- Example
 - If: `int[] list = {10, 20, 30, 40, 50, 60};`
 - Then: `list.length` is 6

Code to Initialize Array to Specific Value (10.00)

```
for(index = 0; index < sale.length; index++)  
    sale[index] = 10.00;
```

Code to Read Data into Array

```
for(index = 0; index < sale.length; index++)  
    sale[index] = Integer.parseInt(keyboard.readLine());
```

Code to Print Array

```
for(index = 0; index < sale.length; index++)  
    System.out.print(sale[index] + " ");
```

Code to Find Sum and Average of Array

```
sum = 0;
for(index = 0; index < sale.length; index++)
    sum = sum + sale[index];

if(sale.length != 0)
    average = sum / sale.length;
else
    average = 0.0;
```

Determining Largest Element in Array

```
maxIndex = 0;  
for(index = 1; index < sale.length; index++)  
    if(sale[maxIndex] < sale[index])  
        maxIndex = index;  
  
largestSale = sale[maxIndex];
```

Array Index Out of Bounds

- Array in bounds if:
 $0 \leq \text{index} \leq \text{arraySize} - 1$
- If $\text{index} < 0$ or $\text{index} > \text{arraySize}$:
`ArrayIndexOutOfBoundsException`
exception is thrown
- Base address: memory location of first component in array

Array variables as reference variables

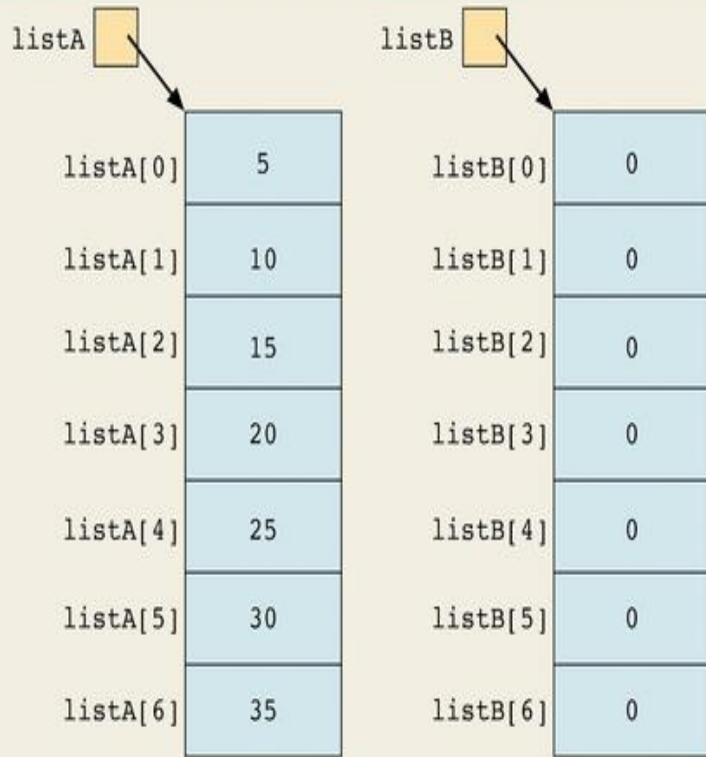


Figure 9-6 Arrays `listA` and `listB`

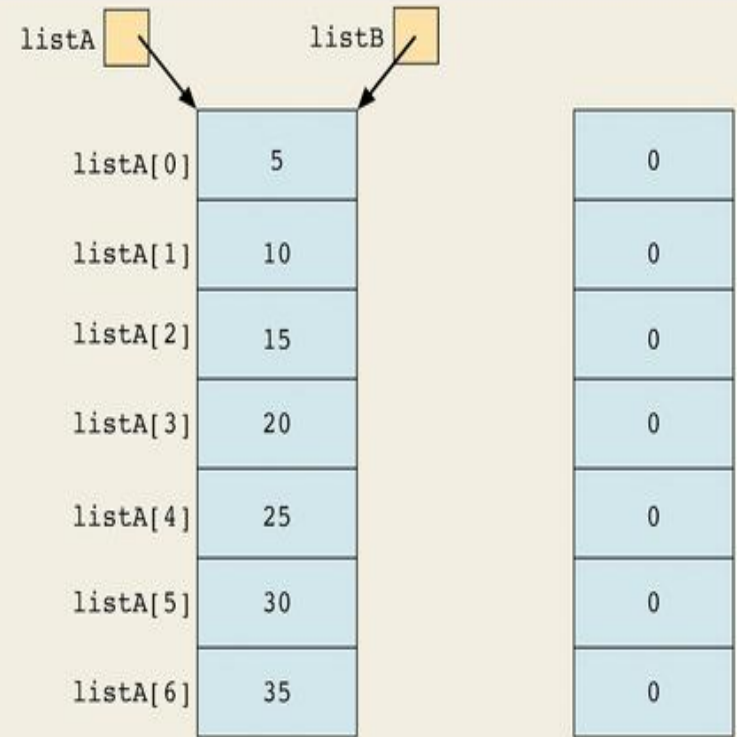


Figure 9-7 Arrays after the statement `listB = listA;` executes

Changes to elements in `listA` or `listB` will change both, as both references point to the same array

Copying an Array

```
for(int index = 0; index < listA.length; index++)  
    listB[index] = listA[index];
```

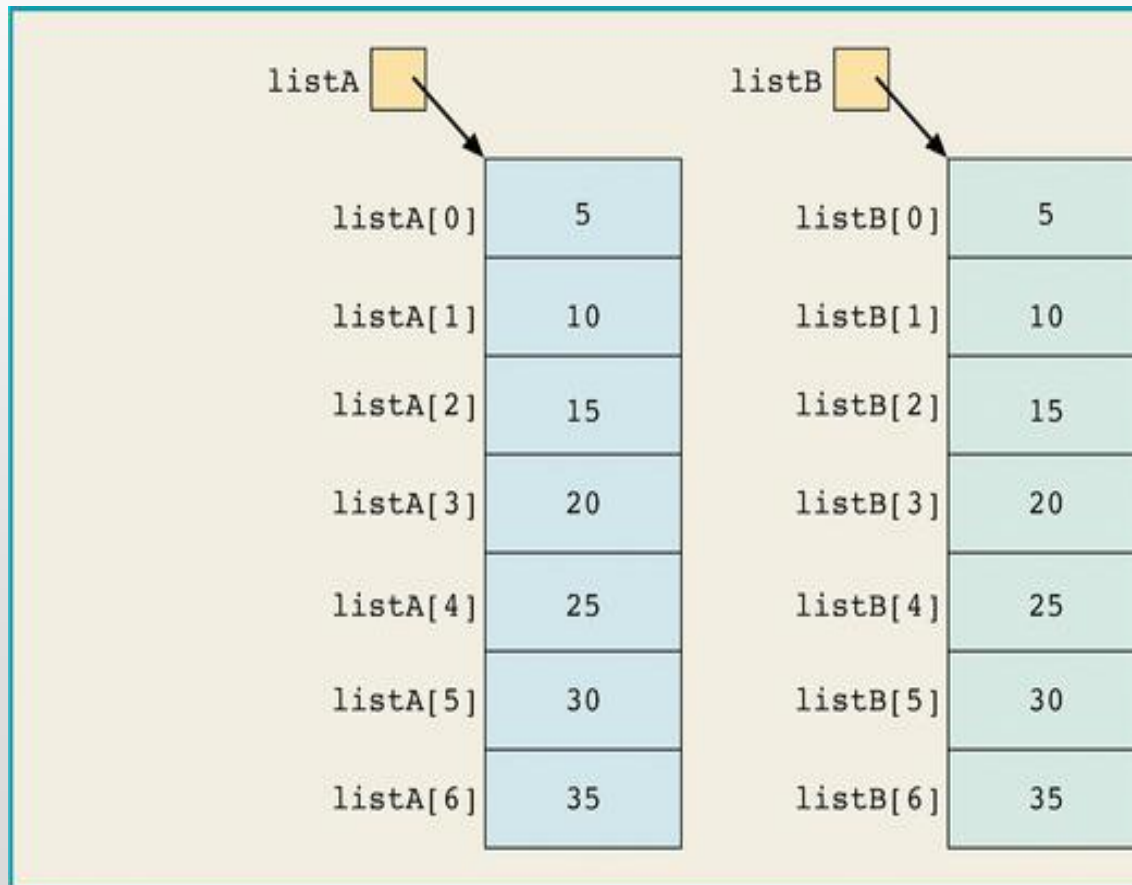


Figure 9-8 Arrays after the `for` loop executes

Arrays of Objects

- Can use arrays to manipulate objects
- Example: create array named array1 with N objects of type T

```
T [ ] array1 = new T[N]
```

- Can instantiate array1 as follows:

```
for(int j=0; j <array1.length; j++)  
    array1[j] = new T();
```

Arrays of Objects:

```
Clock[ ] arrivalTimeEmp = new Clock [100];
```

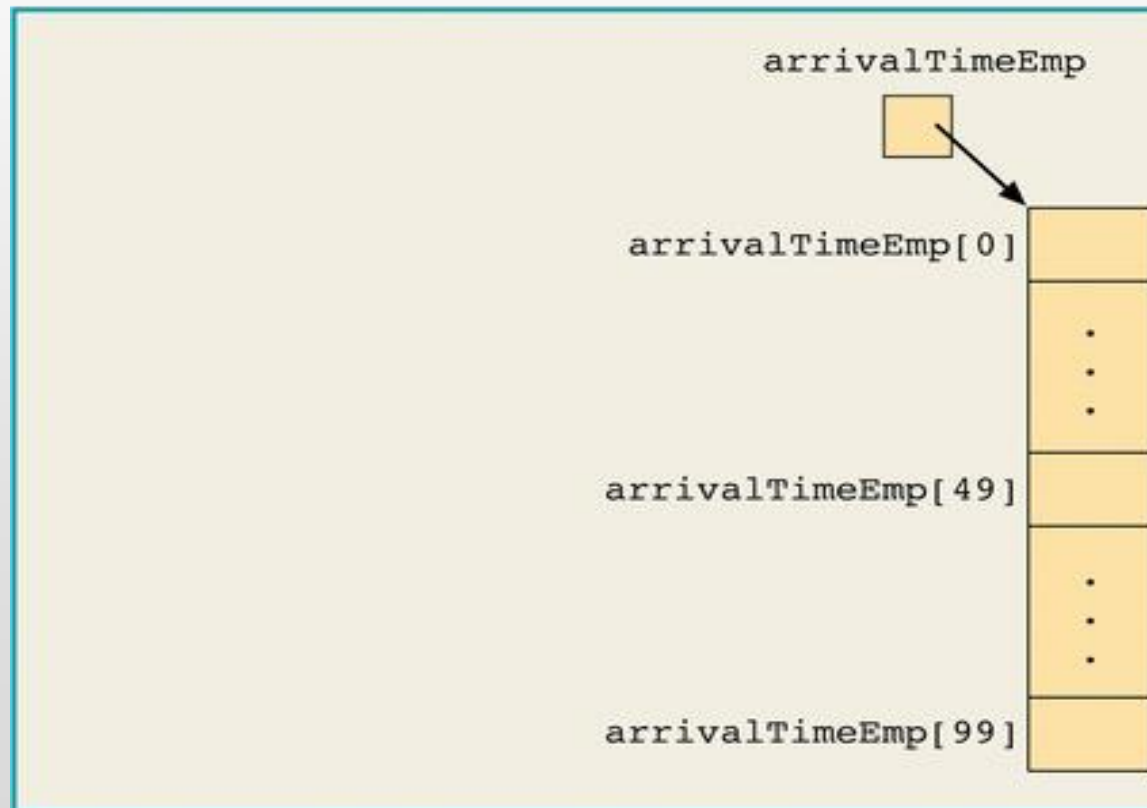


Figure 9-9 Array `arrivalTimeEmp`

Instantiating Array Objects

```
for(int j = 0; j < arrivalTimeEmp.length; j++)  
    arrivalTimeEmp[j] = new Clock();
```

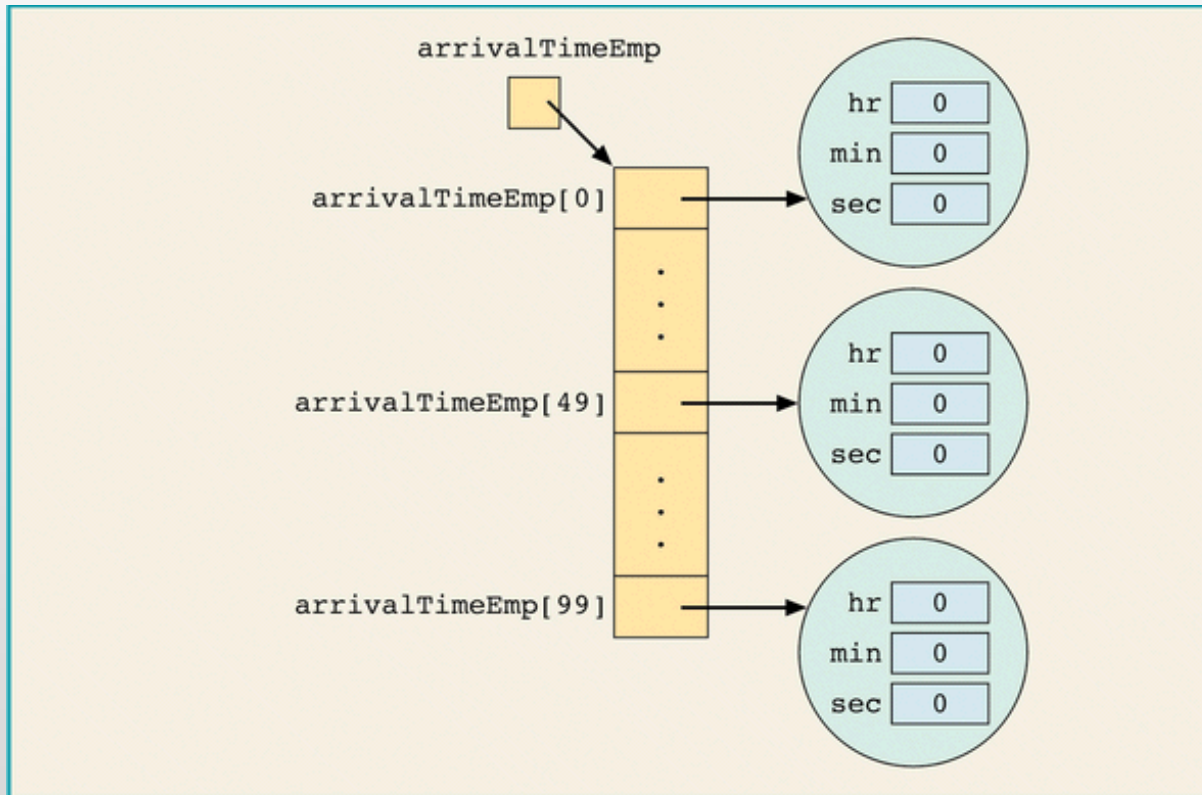


Figure 9-10 Array `arrivalTime` after instantiating objects for each component

Two-Dimensional Arrays

- Data is sometimes in table form (difficult to represent using one-dimensional array)
- To declare/instantiate two-dimensional array:

```
dataType[ ][ ] arrayName = new  
dataType[intExp1][intExp2];
```

- To access a component of a 2-dimensional array:

```
arrayName[indexExp1][indexExp2];
```

- intExp1, intExp2 ≥ 0
- indexExp1 = row position
- indexExp2 = column position

Two-Dimensional Arrays

- Can specify different number of columns for each row (ragged arrays)
- Three ways to process 2-D arrays
 - Entire array
 - Particular row of array (row processing)
 - Particular column of array (column processing)
- Processing algorithms similar to processing algorithms of one-dimensional arrays

Two-Dimensional Arrays

```
double[ ][ ]sales = new double[10][5];
```

	[0]	[1]	[2]	[3]	[4]
[0]	0.0	0.0	0.0	0.0	0.0
[1]	0.0	0.0	0.0	0.0	0.0
[2]	0.0	0.0	0.0	0.0	0.0
[3]	0.0	0.0	0.0	0.0	0.0
[3]	0.0	0.0	0.0	0.0	0.0
[4]	0.0	0.0	0.0	0.0	0.0
[5]	0.0	0.0	0.0	0.0	0.0
[6]	0.0	0.0	0.0	0.0	0.0
[7]	0.0	0.0	0.0	0.0	0.0
[8]	0.0	0.0	0.0	0.0	0.0
[9]	0.0	0.0	0.0	0.0	0.0

Figure 9-13 Two-dimensional array sales

Accessing Two-Dimensional Array Components

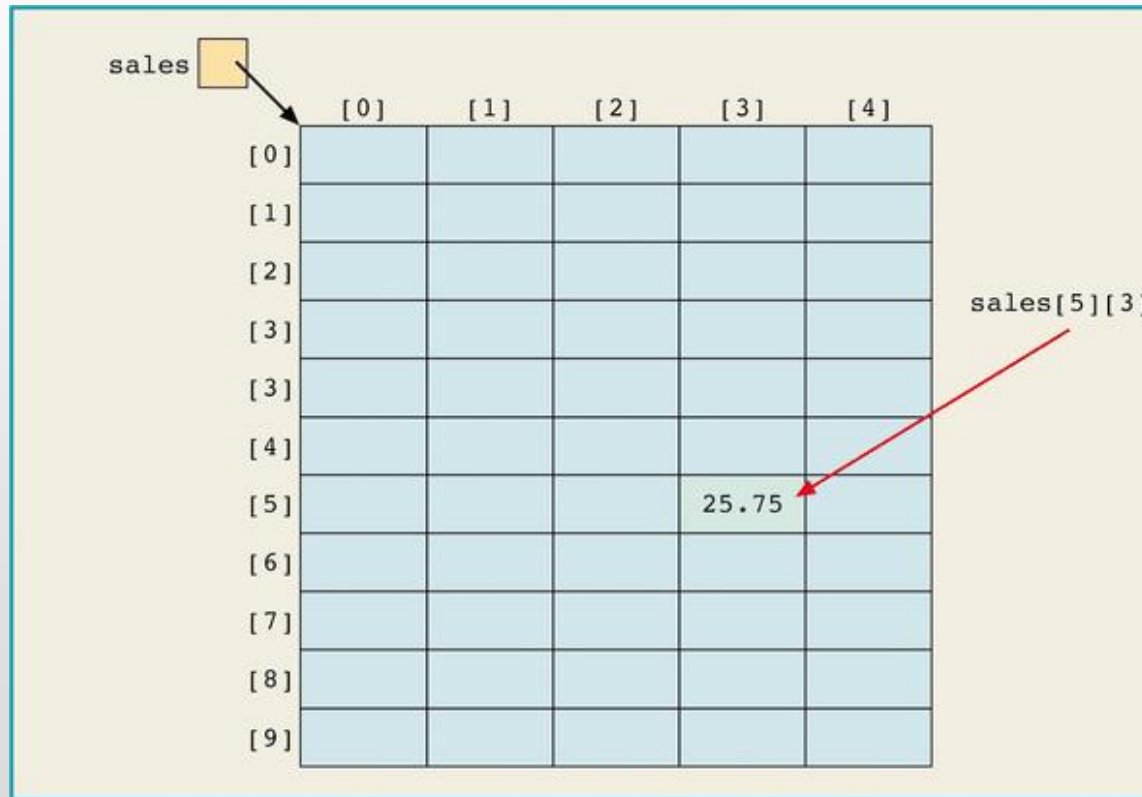


Figure 9-14 `sales[5][3]`

Two-Dimensional Arrays: Special Cases

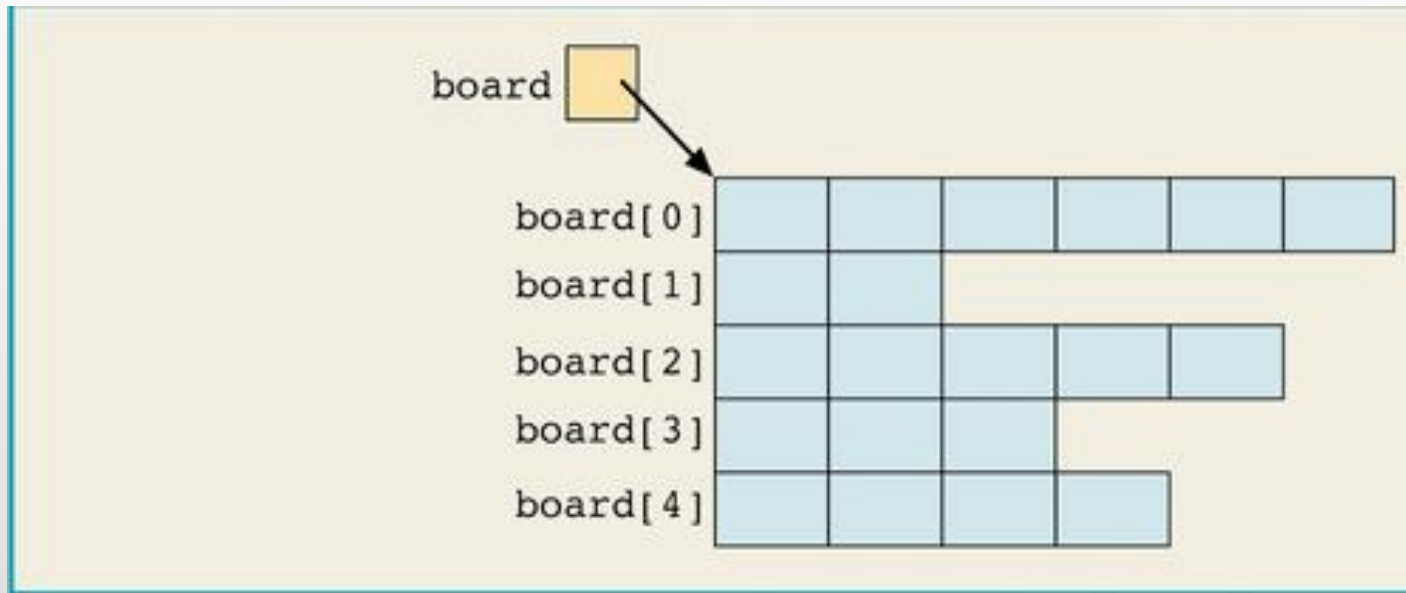


Figure 9-15 Two-dimensional array board

Multidimensional Arrays

- Can define three-dimensional arrays or n-dimensional array (n can be any number)

- Syntax to declare and instantiate array:

```
dataType[ ][ ]...[ ] arrayName = new  
dataType[intExp1][intExp2]...[intExpn];
```

- Syntax to access component:

```
arrayName[indexExp1][indexExp2]...[indexExpn]
```

- intExp1, intExp2, ..., intExpn = positive integers
- indexExp1, indexExp2, ..., indexExpn = non-negative integers

Loops to Process Multidimensional Arrays

```
double[][][] carDealers = new double [10][5][7];
```

```
for(i = 0; i < 10; i++)  
    for(j = 0; j < 5; j++)  
        for(k = 0; k < 7; k++)  
            carDealers[i][j][k] = 10.00;
```

Returning to Methods...

- Let's now switch gears and look at how users can create their own methods.

Syntax: Value-Returning Method

```
modifier(s) returnType methodName(formal parameter list)
{
    statements
}
```

User-Defined Methods

- Value-returning methods
 - Used in expressions
 - Calculate and return a value
 - Can save value for later calculation or print value
- Modifiers: public, private, protected, static, abstract, final
- returnType: type of value that the method calculates and returns (using return statement)
- methodName: Java identifier, name of method

Syntax

Syntax: Formal Parameter List

The syntax of the formal parameter list is:

data Type identifier, data Type identifier,...

Method Call

The syntax to call a value-returning method is:

methodName(actual parameter list)

Syntax

Syntax: Actual Parameter List

The syntax of the actual parameter list is:

expression or variable, expression or variable, ...

Syntax: `return` Statement

The `return` statement has the following syntax:

```
return expr;
```

Equivalent Method Definitions

```
public static double larger(double x, double y)
{
    double max;
    if (x >= y)
        max = x;
    else
        max = y;
    return max;
}
```


Equivalent Method Definitions

```
public static double larger(double x, double y)
{
    if (x >= y)
        return x;
    else
        return y;
}
```

Flow of Execution

- Execution always begins with the first statement in the method main
- User-defined methods execute only when called
- Call to method transfers control from caller to called method
- In method call statement, specify only actual parameters, not data type or method type
- Control goes back to caller when method exits

Programming Example: Largest Number

- Input: Set of 10 numbers
- Output: Largest of 10 numbers
- Solution:
 - Get numbers one at a time
 - Method largest number: returns the larger of 2 numbers
 - For loop: calls method largest number on each number received and compares to current largest number

Void Methods

- Similar in structure to value-returning methods
- No method type
- Call to method is always stand-alone statement
- Can use return statement to exit method early

Void Methods: Syntax

Method Definition

The general form (syntax) of a void method without parameters is as follows:

```
modifier(s) void methodName()  
{  
    statements  
}
```

Method Call (Within the Class)

The method call has the following syntax:

```
methodName();
```

Void Methods with Parameters: Syntax

Method Definition

The definition of a void method with parameters has the following syntax:

```
modifier(s) void methodName(formal parameter list)
{
    statements
}
```

Formal Parameter List

The formal parameter list has the following syntax:

```
dataType variable, dataType variable, ...
```

Void Methods with Parameters: Syntax

➤ **Method Call**

- The method call has the following syntax:

`methodName(actual parameter list);`

➤ **Actual Parameter List**

- The actual parameter list has the following syntax:

`expression or variable, expression or variable, ...`

Primitive Data Type Variables as Parameters

- A formal parameter receives a copy of its corresponding actual parameter
- If a formal parameter is a variable of a primitive data type
 - Value of actual parameter is directly stored
 - Cannot pass information outside the method
 - Provides only a one-way link between actual parameters and formal parameters

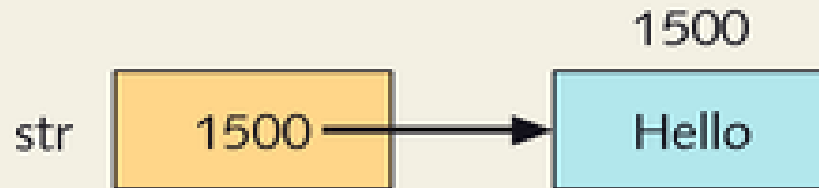
Reference Variables as Parameters

- If a formal parameter is a reference variable
 - Copies value of corresponding actual parameter
 - Value of actual parameter is address of object where actual data is stored
 - Both formal and actual parameter refer to same object

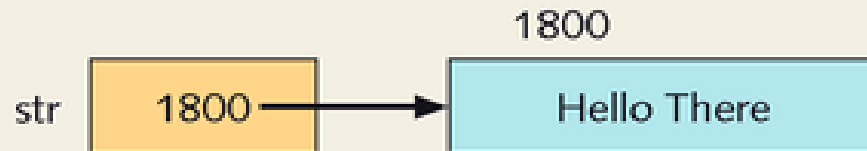
Uses of Reference Variables as Parameters

- Can return more than one value from a method
- Can change the value of the actual object
- When passing address, would save memory space and time, relative to copying large amount of data

Reference Variables as Parameters: type String



Variable `str` and the string object



`str` after the statement `str = "Hello There";` or
`str = str + " There";` executes

Method Overloading

- Method overloading: more than one method can have the same name
- Overloading Rules
 - Every method must have a different number of formal parameters
 - OR
 - If the number of formal parameters is the same, then the data type of the formal parameter (in the order listed), must differ in at least one position
 - Types of parameters determine which method executes

Constructors and methods

- All Java functions are encapsulated as either constructors or methods.
- A constructor has the *same name* as its encapsulating class and no return type or **void** in place of a return type. For instance, a **Date** constructor would have the name **Date**.

Constructors and methods

- All Java functions are encapsulated as either constructors or methods.
- A constructor has the *same name* as its encapsulating class and no return type or **void** in place of a return type. For instance, a **Date** constructor would have the name **Date**.

Constructors

- Constructors are used with the **new** operator to construct instances of a class. The statement

```
Date today = new Date();
```

illustrates.

- Constructors are typically *overloaded*; that is, a class typically has several constructors.

Constructors vs. Methods

- A method does not have the same name as its encapsulating class and has either a return type or `void` in place of a return type.
- Methods define the operations appropriate to a class and its instances, and can be called many times for a single instance
 - Constructors are only called when the object is created.

Constructors vs. Methods

- A constructor cannot be **static**, but a method can be **static**.
- Constructors and methods can be parameterized. The parameter names and data types must be provided.
- Methods, like constructors, can be overloaded but must be distinguished by their names and/or parameter types.