

# Java Primer

Based on 'Data Structures for Java' by W. H. Ford, W. R. Topp

# Structure of a Java Program

## Sample Problem:

A person has a height of 74 inches.

The program converts the height into feet and inches and into centimeters.

The conversion factor "1 inch = 2.54 cm" is used for the metric measure.

Output to the screen displays the conversion in the form

" Height of <foot> foot <inch> in metric is <centimeter> cm"  
where the bracketed symbol <value> is the actual value of the corresponding unit.

# Structure of a Java Program

- The class name with the extension ".java" becomes the file name.
- By convention, all class names in Java begin with a capital letter.
  - For the class name DemoProgram the file name is DemoProgram.java.
  - The body of the class is enclosed in a pair of braces and includes a special method called *main*.
    - This method designates where the runtime system will begin execution of the program.

# Structure of a Java Program

```
// main class for source code in file "DemoProgram.java"  
public class DemoProgram  
{  
    public static void main(String[] args)  
    {  
        <code to implement main()>  
    }  
}
```

# Program Listing

```
[1] // DemoProgram: Converts height in inches into
[2] // units of feet and inches as well as metric
[3] // units and then outputs the results
[4]
[5] // application class that contains the main method
[6] public class DemoProgram
[7] {
[8]     public static void main(String[] args)
[9]     {
[10]         // the constant conversion factor inches to
[11]         // centimeters
[12]         final double IN2CM = 2.54;
[13]
[14]         // variables for height, feet, inches, and
[15]         // centimeters
[16]         int height = 74, foot, inch;
[17]         double centimeter;
```

# Program Listing

```
[18]  
[19]    // convert height to feet and inches  
[20]    foot = height / 12;  
[21]    inch = height % 12;  
[22]    centimeter = height * IN2CM;  
[23]  
[24]    // display the output  
[25]    System.out.println("Height of " + foot +  
[26]        " foot " + inch + " in metric is "  
[27]        + centimeter + " cm");  
[28]    }  
[29] }
```

```
Height of 6 foot 2 in metric is 187.96 cm
```

# Comments

- A *single-line* comment starts with the character sequence "//" and continues to the end of the line.
- A *multi-line* comment includes all of the text that is enclosed within the pair of delimiters "/\*" and "\*/".
- A third form is a Javadoc comment that creates HTML documentation.

# Keywords and Identifiers

- *Keywords* are words that have special predefined meaning in the Java language.
  - They may not be used as programmer-defined names for a class or other components in the program.Examples: *class*, *static*
- The name of a class, a method, or a variable is called an *identifier*.
  - An identifier is a series of characters consisting of letters (a...z, A...Z), digits (0...9), underscores (\_), and dollar signs (\$)An identifier may not begin with a digit.



# Declaring and Using Variables

- A *variable* is the program's name for a memory location that holds data.
- A variable must have an associated type that indicates whether the data is an integer, a real number, a character, or other kind of value.

```
int height = 74, foot, inch;  
double centimeter;
```

# Declaring and Using Variable

- Java has two kinds of types
  - primitive
  - reference

A reference type is associated with an object. The type of an object is a user-defined class.

Primitive types are predefined in the Java language and designate variables that have simple integer or real number values, character values, or logical values (true or false).

# Console Output

- Java provides the predefined stream `System.out` for console output.

Use the stream with methods *print* and *println* to display strings and other information in a console window.

# Console Output

The form of the output is provided by a string that is built with elements separated by the '+' character.

The elements may include quoted strings (string literals), variables, and constants.

```
System.out.println("Height of " + foot + " foot " + inch +  
                    " in metric is " + centimeter + " cm");
```

# Java Programming Environment

- Programs are initially written as a sequence of Java statements and then translated by a compiler into *bytecode*.
- Bytecode contains machine instructions for a hypothetical computer.
- To execute the bytecode, Java provides an interpreter, called a ***Java Virtual Machine*** (JVM) that simulates the bytecode on the local computer.

# Java Programming Environment

- The most basic environment is a *command line environment* that uses a separate application from the *Java Software Development Kit (SDK)* for each task.
  - To compile the program, use the command "javac" with a file name that includes the extension ".java".
  - The command converts the source code into intermediate bytecode instructions. The resulting file has the extension ".class".

# Java Programming Environment

`javac compiles "DemoProgram.java" to bytecode "DemoProgram.class"`

`> javac DemoProgram.java`

- To execute the program, use the command "java" with the name of the ".class" file.

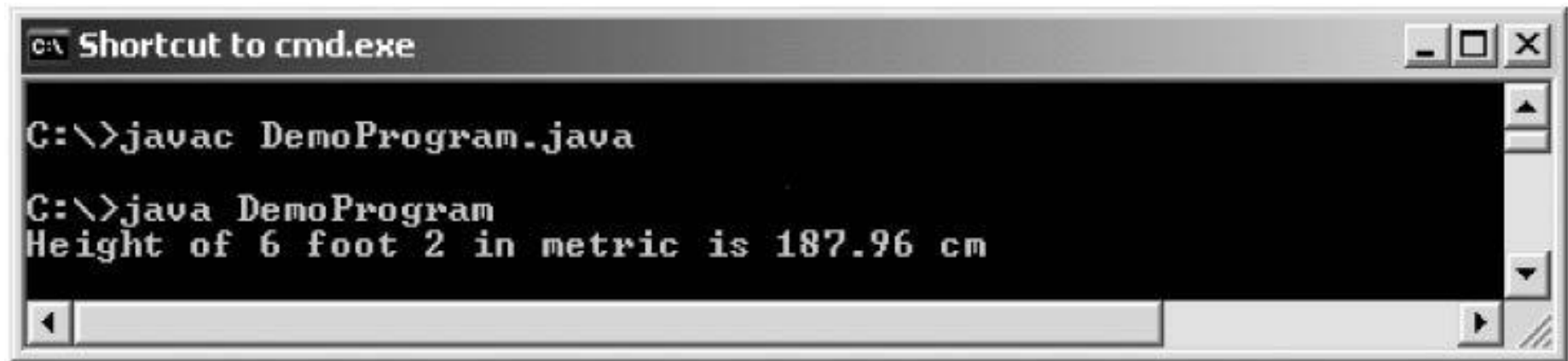
The extension is not included.

- The application loads the Java Virtual Machine for the system and executes the bytecode instructions

`execute the bytecode in file "DemoProgram.class"`

`> java DemoProgram`

# Java Programming Environment



```
C:\>javac DemoProgram.java  
  
C:\>java DemoProgram  
Height of 6 foot 2 in metric is 187.96 cm
```

The action of compiling and running demonstration program in a Windows command-line environment.

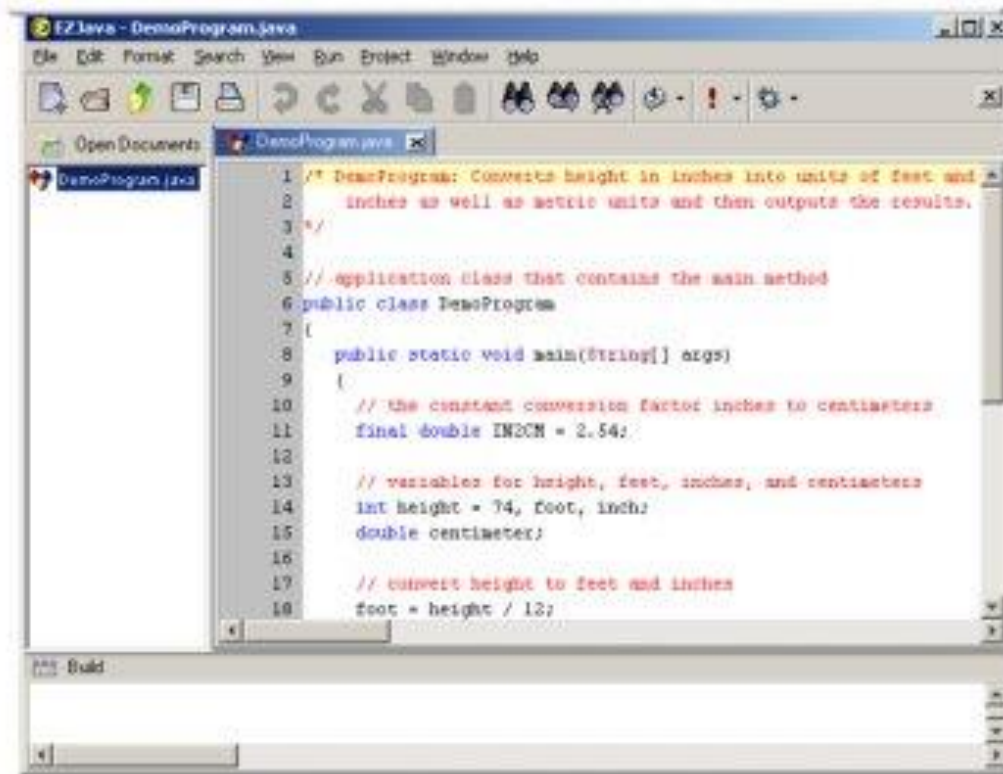


# Integrated Development Environment

- An Integrated Development Environment (IDE) provides tools to support the entire program development and execution process.
  - The tools include
    - an editor for writing programs,
    - debuggers to locate errors,
    - a window to display compiler messages, and
    - a runtime window to execute the program.

Examples: JBuilder from Borland, CodeWarrior from Metrowerks, and the author-supplied EZJava.

# Integrated Development Environment



Using the EZJava editor and commands to compile and run DemoProgram.java with a display window.

# Primitive Number Types

Primitive Types	Size in Bits	Range
byte	8-bit integer	-128 to 127
short	16-bit integer	-32768 to 32767
int	32-bit integer	-2,147,483,648 to 2,147,483,647
long	64-bit integer	(Approx) $-9 \times 10^{18}$ to $9 \times 10^{18}$
float	32-bit real number	(Approx) $-3.4 \times 10^{38}$ to $3.4 \times 10^{38}$
double	64-bit real number	(Approx) $-1.7 \times 10^{308}$ to $1.7 \times 10^{308}$

# Declaration of primitive numeric variables

```
// uninitialized variable of type byte
byte b;
// declaration of two int variables; n is initialized
int m, n = 15500;
// compiler error; range is -128 to 127
byte badByte = 1000;
// uninitialized variable
double slope;
// use fixed-point notation
double x = 14.89;
// use floating-point notation
double y = 2.3e-5;
// the literal is a 32-bit float
float t = 0.0775;
```

# Declaration of primitive numeric variables

```
byte b;
```

```
int m, n = 15500;
```

```
byte badByte = 1000;
```

```
double slope;
```

```
double x = 14.89;
```

```
double y = 2.3e-5;
```

```
float t = 0.0775;
```

# Declaration of primitive numeric variables

```
byte b;  
int m, n = 15500;  
byte badByte = 1000;  
double slope;  
double x = 14.89;  
double y = 2.3e-5;  
float t = 0.0775f;
```

# Declaration of primitive numeric variables

```
// uninitialized variable of type byte
```

```
byte b;
```

```
// declaration of two int variables; n is initialized
```

```
int m, n = 15500;
```

```
// compiler error; range is -128 to 127
```

```
byte badByte = 1000;
```

```
// uninitialized variable
```

```
double slope;
```

# Declaration of primitive numeric variables

```
// use fixed-point notation
```

```
double x = 14.89;
```

```
// use floating-point notation
```

```
double y = 2.3e-5;
```

```
// the literal is a 32-bit float
```

```
float t = 0.0775f;
```



# Java char Type

- Character data includes uppercase and lowercase letters, digits, punctuation marks, and special symbols.
  - The Java primitive type *char* specifies a single character.
  - A character literal is a character enclosed in single quotation marks.

```
// declare the variables ch and letter  
// assign letter the character 'A'  
char ch, letter = 'A'
```

# Java char Type

- Java defines a set of ***escape sequence*** to represent special characters.

Character	Escape Code
backslash	\\
carriage return	\r
double quote	\"
newline	\n
tab	\t

# Java char Type

```
System.out.println("Source file is \"DemoProgram.java\");  
System.out.println("Input c:\\dataIn.dat\\n" +  
                    "Output c:\\dataOut.dat");
```

## Output:

```
Source file is "DemoProgram.java"  
Input c:\\dataIn.dat  
Output c:\\dataOut.dat
```

# Declaring Named Constants

- A program often uses a number or string to define a value that should not be changed.
- For instance, for circle calculations we use the value 3.14159265 for  $\pi$ .
  - Create a named constant using the keyword ***final*** followed by the declaration for an initialized primitive variable.

```
final int LIMIT = 50;  
final double PI = 3.14159265;  
final char DELIMITER = ':';
```

# Arithmetic Operators

- An arithmetic expression combines numeric values and operators.
- The familiar binary operators
  - addition (+)
  - subtraction (-)
  - multiplication (\*)apply to both integer and real numbers.

# Arithmetic Operators

The unary operator

negation (-)

changes the sign of the operand.

With integers, division is in fact long division that evaluates to a quotient and a remainder.

The operator / returns the quotient.

The % operator gives the remainder

Ex :  $(25/3=8, 25\%3=1)$ .

# Assignment Operator

- Java uses the *assignment operator* = to copy the value of an expression on the right-hand side (rhs) into a variable on the left-hand side (lhs).
- An assignment statement is terminated by a semicolon.

```
int m, n;  
m = 8;           // rhs is the integer literal 8  
n = m * 2;       // rhs is an expression which evaluates to 16  
n = m = 25;      // assigns 25 as value of m and then the value  
                  // of m to n
```

# Compound Assignment Operators

Compound assignment:      lhs <op>= rhs

Simple assignment:          lhs = lhs <op> rhs

Example: Assume m = 14 and n = 3.

m += 5;	// m = m + 5;	m is 19
n += m - 2;	// n = n + (m - 2);	n is 15
n *= 2;	// n = n * 2;	n is 6
m /= 5;	// m = m / 5;	m is 2
m %= 5;	// m = m % 5;	m is 4



# Increment and Decrement Operators

- Java provides unary operators ++ and -- for the increment and decrement operations respectively.

```
count++;           // increment operator
```

If the value of count is initially 8 before the statement is executed, the resulting value is 9.

```
count--;           // decrement operator
```

From initial value 8, the result becomes 7;

# Operator Precedence and Associativity

- In Java, each operator has a *precedence level* and the compiler generates code to execute operators in the order of their precedence.

```
int m = 40, n;  
n = -m + 14 % 4;
```

First execute negation:       $n = -40 + 14 \% 4$

Second execute remainder:       $= -40 + 2$        $// \ 14 \% 4 = 2$

Third execute addition:       $= -38$        $// \ -40 + 2 = -38$

implicitly  $n = ((-m) + (14 \% 4));$

# Operator Precedence and Associativity

Operator precedence and associativity for arithmetic and assignment operators

Level	Operator	Operation	Associativity
0	=	Assignment	R to L
1	+	Addition	L to R
	-	Subtraction	L to R
2	*	Multiplication	L to R
	/	Division	L to R
	%	Remainder	L to R
3	+	Unary plus	R to L
	-	Unary minus	R to L

```
int m = 9, n = 5, p, q;  
p = q = 4 * m / n;
```

First execute multiplication:

```
p=q=36/n // 4 * m = 4 * 9 = 36
```

Second execute division:

```
p=q=7 // 36 / n = 36 / 5 = 7
```

Third execute assignment (R to L):

```
q=7 // assign 7 to q
```

Fourth execute assignment (R to L):

```
p=q // assign q = 7 to p
```

# Type Conversions

- Conversions between one primitive type and another are classified as
  - *widening conversions* or
  - *narrowing conversions*.
- Widening conversions go from one data type to another that uses the same or more memory space to store the values.

# Type Conversions

- Narrowing conversions go from one type to another that uses a smaller space to store values.
- The result is often a loss of both the magnitude and precision of the value.
- The ordering of the widening conversions for numeric types is

`byte -> short -> int -> long -> float -> double`

# Arithmetic Promotion and Casting

- Arithmetic *promotion* occurs automatically.

`4 + 6.5` is evaluated as `4.0 + 6.5 = 10.5`

- A *cast* is an explicit directive to the compiler indicating that a conversion should occur. The format for a cast places the desired type in parentheses in front of the operand, which may be variable, literal, or complex expression.

**(type) operand**

```
int total = 5;  
// avg1 = 1, avg2 = 1.25  
double avg1 = total/4, avg2 = (double) total/4;
```

# Assignment Conversion

- Assignment conversion occurs when the expression on the right side of an assignment statement has a different type than the left hand side.
- Assignment accomplishes only widening conversion from the right-hand side to the left-hand side.
- Otherwise, the right-hand side must be explicitly cast.

```
int m = 15, n = 65;  
double x;  
char ch;
```

```
x = m;           // with widening conversion, x is 15.0  
ch = (char) n;   // explicit casting, ch is 'A'
```

# Java Comparison Operators

## Java comparison operators with equivalent math notation

Mathematics Notation:	=
Java Notation:	==
Meaning:	Equal to
Java Example:	<code>n % 2 == 0</code>
Mathematics Notation:	≠
Java Notation:	!=
Meaning:	Not equal to
Java Example:	<code>response != 'Y'</code>
Mathematics Notation:	<
Java Notation:	<
Meaning:	Less than
Java Example:	<code>4 &lt; 6</code>



# Java Comparison Operators

## Java comparison operators with equivalent math notation

Mathematics Notation:	$\leq$
Java Notation:	<code>&lt;=</code>
Meaning:	Less than or equal to
Java Example:	<code>age &lt;= 21</code>
Mathematics Notation:	$>$
Java Notation:	<code>&gt;</code>
Meaning:	Greater than
Java Example:	<code>balance &gt; 0</code>
Mathematics Notation:	$\geq$
Java Notation:	<code>&gt;=</code>
Meaning:	Greater than or equal to
Java Example:	<code>ch &gt;= 'A'</code>

# Boolean Operators

<u>Logical Operator</u>	<u>Java Operator</u>
AND	&&
OR	
NOT	!

- `P && Q` is true provided both P and Q are true; it is false in all other cases.  
Example: `'a' <= ch && ch <= 'z'`  
`// true if ch is a lowercase letter`
- `P || Q` is true if P is true or if Q is true; it is false only when both P and Q are false  
Example: `n % 2 == 0 || n == 5`  
`// true if n is even or n is 5`
- `!P` is the logical opposite of P. It is true when P is false and false when P is true  
Example: `!(n % 5 == 0)`  
`// true if n is not divisible by 5`

# Boolean Operators

- With *short-circuit evaluation*, the computer evaluates the operands from left to right and stops as soon as the logical value of the entire expression is known.

In the following expression, the division is not done if x is 0.0.

```
x != 0.0 && 20/x < 1
```

# if-Statement

- A *code block* is a group of one or more statements that are combined to carry out a single task.
- Instructions within the block are set off by braces and may contain declarations.
- In the case of a single statement, the braces may be omitted.
- A variable declared in a block can only be used in the block. We say that the block defines the *scope* of the variable.

# if-Statement

*Block Syntax:*

```
{  
    Declarations      // declaration of variables  
    Statement1      // sequence of statements  
    ...  
    Statementn  
}
```

- The simplest form of an if-statement uses a logical expression as a condition and executes code block code<sub>T</sub> if the expression evaluates to true.

*Syntax:*

```
if (condition)  
    CodeT
```

# if-Statement

```
if (m < n)
{
    int temp;    // the exchange needs temporary storage

    temp = m;    // hold m in temporary storage
    m = n;       // copy n to m
    n = temp;    // copy original value of m to n
}
```

# if-Statement

- The most common form of an if-statement has the program choose from among two options.
- The form of the if-statement uses the reserved word *else* to designate the second option.
- The statement, called an *if-else statement*, tests the condition and executes  $\text{Code}_T$  if the condition is true and  $\text{Code}_F$  otherwise (if it is false).

## Syntax:

```
if (condition)
     $\text{Code}_T$  // if true, execute this code
else
     $\text{Code}_F$  // if false, execute this code
```

# if-Statement

- A course grade is 'P' or 'F' indicating pass or failure. The instructor passes a student when the average score is 70 or above.

```
if (avgScore >= 70)
    grade = 'P';
else
    grade = 'F';
```



# Nested if-Statements

- An *if-else* statement can contain any sort of statements within its code blocks.
- In particular, it can constitute a nested if-statement by placing if-else statements within the blocks.

# Nested if-Statements

A lumber company prices sheets of plywood based on their grade F (finished) or U (utility).

In addition, the company offers customers a reduced price on a sheet if they buy in quantity.

The following table lists the price of a sheet of plywood based on grade and quantity

Plywood price (per sheet)		
	Utility	Finished
Quantity 1-9	12.50	18.75
Quantity 10+	11.25	17.25

# Nested if-Statements

```
// first test for the grade; nested if-else
// statements for each grade test the quantity
if (grade == 'U')
    if (quantity < 10)
        // option: U grade, quantity 1-9
        totalCost = n * 12.50;
    else
        // option: U grade, quantity 10+
        totalCost = n * 11.25;
else
    if (quantity < 10)
        // option: F grade, quantity 1-9
        totalCost = n * 18.75;
    else
        // option: F grade, quantity 10+
        totalCost = n * 17.25;
```

# Nested if-Statements

```
if (grade == 'U')
    if (quantity < 10)

        totalCost = n * 12.50;
    else

        totalCost = n * 11.25;
else
    if (quantity < 10)

        totalCost = n * 18.75;
    else

        totalCost = n * 17.25;
```

# Nested if-Statements

```
if (grade == 'U')  
    if (quantity < 10)  
        totalCost = n * 12.50;  
    else  
        totalCost = n * 11.25;  
else  
    if (quantity < 10)  
        totalCost = n * 18.75;  
    else  
        totalCost = n * 17.25;
```

# Nested if-Statements

```
if (grade == 'U') {  
    if (quantity < 10)  
        totalCost = n * 12.50;  
    else  
        totalCost = n * 11.25;  
}  
else {  
    if (quantity < 10)  
        totalCost = n * 18.75;  
    else  
        totalCost = n * 17.25;  
}
```

# Multiway if/else Statements

- In many applications, we want a selection statement to specify branches into a number of different options.
- This can be done with nested if-statements but without indenting to list the choices.
- The format creates a multiway if-else statement that better describes how we view the options.

# Multiway if/else Statements

```
if (avgScore >= 90)
    grade = 'A';
else if (avgScore >= 80)
    grade = 'B';
else if (avgScore >= 70)
    grade = 'C';
else if (avgScore >= 60)
    grade = 'D';
// could be a simple else statement
else if (avgScore < 60)
    grade = 'F';
```



# Multiway if/else Statements

```
if (avgScore >= 90)
    grade = 'A';
else if (avgScore >= 80)
    grade = 'B';
else if (avgScore >= 70)
    grade = 'C';
else if (avgScore >= 60)
    grade = 'D';
// could be a simple else statement
else grade = 'F';
```

# Conditional Expression Operator

- Java provides an alternative to the "if-else" statement using the *conditional expression operator* (?:).
- The syntax for the operator requires three operands.
- The first is a boolean expression and the other two are expressions that are set off with separators "?" and ":" respectively.

*Syntax:*     condition ? expression<sub>T</sub> : expression<sub>F</sub>

# Conditional Expression Operator

```
max = (x >= y) ? x : y;
```

is equivalent to

```
if (x >= y)
    max = x;
else
    max = y;
```

# switch-Statement

- The *switch-statement* is a special form of multiway selection that transfers control to one of several statements depending on the value of an integer or character expression.
  - If no match occurs, then control passes to the *default* label if it exists or to the first statement following the switch block.
  - When a statement sequence concludes, control continues with the next statement sequence.
  - Java provides a *break* statement, which forces a branch out of the switch statement.

# switch-Statement

## Syntax:

```
switch (selector expression)
{
    case constant1:      Statement for constant1
                          break;
                          . . . . .
    case constantn:      Statement for constantn
                          break;

    default:             Statement if no case matches
                          selector
                          break;
}
```

# Switch-Statement Example

```
switch(coinValue)
{
    case 1:    // Note: two or more case options included with
    case 5:    // a single statement
    case 10:
    case 25: System.out.println(coinValue + " cents " +
        "is a standard coin");
        break;
    case 50: System.out.println(coinValue + " cents " +
        "is a special coin");
        break;
    default: System.out.println("No coin for " + coin +
        " cents");
        break;
}
```

For coinValue = 25, the output is "25 cents is a standard coin"  
For coinValue = 50, the output is "50 cents is a special coin"  
For coinValue = 15, the output is "No coin for 15 cents"

# switch-Statement

If a break statement is not placed at the end of a case-option, the runtime system will execute instructions in the next case-option.

For instance, assume the example includes no break statements and coinValue is 25.

A run would produce output that includes the case 50 option and the default option.

Output:

```
25 cents is a standard coin
25 cents is a special coin
No coin for 25 cents
```

# Switch-Statement Example

```
switch(coinValue)
{
    case 1:    // Note: two or more case options included with
    case 5:    // a single statement
    case 10:
    case 25: System.out.println(coinValue + " cents " +
                               "is a standard coin");

    case 50: System.out.println(coinValue + " cents " +
                               "is a special coin");

    default: System.out.println("No coin for " + coin +
                               " cents");
}
```

## Output

```
25 cents is a standard coin
25 cents is a special coin
No coin for 25 cents
```



# Boolean Type

- The boolean type is a primitive type like int, double, and char.
- The type has values *true* and *false* which can be used in program statements.
- Like the other primitive types, you can have constants and variables of boolean type.
- A variable is typically used as a flag to indicate the status of a condition in an algorithm or as a name for a boolean expression.

# Boolean Type

```
final boolean VALID_ID = true;
```

```
// isEnrolled is a flag that indicates the  
// registration status of a student in a course;  
// The flag is set to false if the student drops  
// the course  
boolean isEnrolled = true;
```

```
// variables are names for a boolean expression;  
// the value of the variable is the value of the  
// boolean expression  
boolean isLowercase = 'a' <= ch && ch <= 'z';  
boolean onProbation = gpa < 2.0;
```

```
boolean isLeapYear = (year % 4 == 0 && year % 100 != 0) ||  
                     (year % 400 == 0);
```

# Boolean Type

```
boolean haveInsufficientFunds = balance < checkAmt;  
boolean isSenior = age >= 60;
```

```
if (haveInsufficientFunds)  
    fee = 20.00;  
else  
    if (isSenior)  
        fee = 0.50;  
    else  
        fee = 1.00;
```

# while Loop

- A *while loop* is the most general form of loop statement. The while statement repeats its action until the controlling condition (loop test) becomes false.

*Syntax:*    while (logical expression)  
                  {body}

Sum successive even integers  $2 + 4 + 6 + \dots$  until the total is greater than 250.

```
int i, sum = 0;
i = 2;           // initial even integer for the sum
while (sum <= 250) // loop test; check current value of sum
{
    sum += i;     // add integer to sum
    i += 2;       // update i to next even integer
}
```

# do/while Loop

- The *do/while* loop is similar to a while loop except that it places the loop test at the end of the loop body.
- A do/while loop executes at least one iteration and then continues execution so long as the test condition remains true.

*Syntax:*

```
do
{
    body
} while (logical expression);
```

# do/while Loop

```
int count = 10;
do
{
    System.out.print(count + " ");
    count--;                // decrement count
}
while (count > 0);          // repeat until count is 0

System.out.println("Blast Off!!!");
```

Output: 10 9 8 7 6 5 4 3 2 1 Blast Off!!!

# for Loop

- The *for-statement* is an alternative loop structure for a counter-controlled while statement.
- The format of the for-loop has three separate fields, separated by semicolons.
- The fields enable a programmer to initialize variables, specify a loop test, and update values for control variables.

Syntax: `for (init statement; test condition; update statement)  
    body`

# for Loop

```
int i, sum = 0;
```

```
while LOOP
```

```
=====
```

```
init:    i = 1;
```

```
test:    while (i <= 10)
```

```
{
```

```
    sum += i;
```

```
update:  i++;
```

```
}
```

```
for LOOP
```

```
=====
```

```
for (i = 1; i <= 10; i++)
```

```
    sum += i;
```



# Break Statement

- Within a loop body, a *break* statement causes immediate exit from the loop to the first statement after the loop body.
- The break allows for an exit at any intermediate statement in the loop.

*Syntax:*   **break**;

```
while (true)
{
    <read data from the file>
    if (eof)
        break;
    <process data from this input>
}
```

# Arrays

- An *array* is a fixed-size collection of elements of the same data type that occupy a block of contiguous memory locations.
- Like any other variable, an array declaration includes the name of the array and the data type for the elements in the sequence.
- To specify that the variable is an array, add a pair of square brackets immediately after the data type.

```
int[] intArr;  
Time24[] tarr;
```

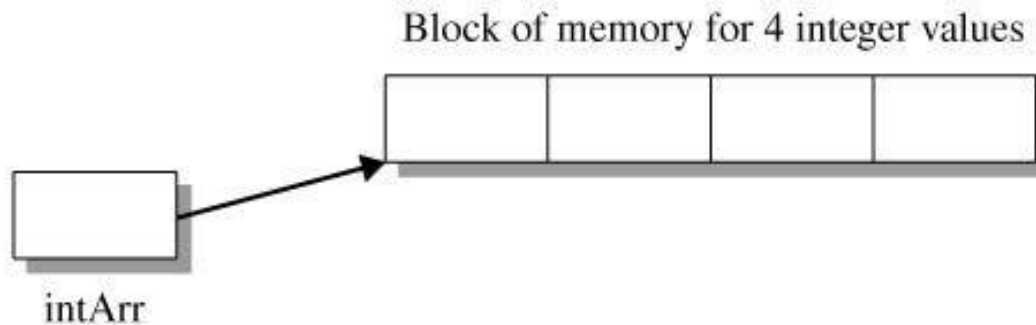
# Arrays

- An array is a different kind of entity called a *reference variable*.
- It contains a value which is a reference (address) to the first location in the associated block of memory.
  - A simple declaration of an array assigns it the value *null*. The array does not point to any actual memory locations.
  - To allocate a block of memory for the array and assign the address of the block to the array variable, use the *new* operator.

# Arrays

- The syntax for the new operator includes the data type and the number of elements in the array enclosed in square brackets.
- For instance, the following statement allocates four integer elements.

```
intArr = new int[4];
```



The figure distinguishes the array reference variable from the memory that holds the array elements.

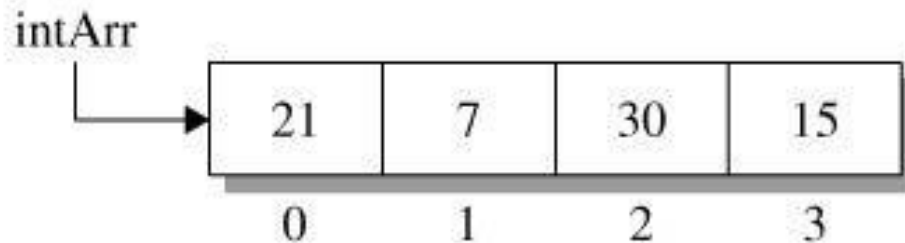
# Arrays

```
int[] intArr = new int[4];           // array of 4 integers
char[] charArr = new char[80];       // array of 80 characters
double[] rainfall = new double[12];  // array of 12 real numbers
```

- Assume  $n$  is the size of the array. Access to an individual element in the array is provided by an index in the range 0 to  $n-1$ .
- The index denotes the position of the element in the sequence.
- The element at index  $i$  is denoted by  $\text{arr}[i]$ . The sequence of elements in the array is  
 $\text{arr}[0], \text{arr}[1], \text{arr}[2], \dots, \text{arr}[n-2], \text{arr}[n-1]$

# Arrays

- For instance, let `intArr` be an array of four integers 21, 7, 30, and 15.
- The figure displays the sequence as a contiguous block of memory with an index listed below each cell.



# Arrays

- The range of valid indices is 0 to  $n-1$  where  $n$  is the size of the array.
- An attempt to access an element outside of the valid index range results in a runtime error.
- Once an array is created, a program may access its size using the expression *arr.length*.
- Java defines length as a variable associated with the array.
- Hence, with any array, the valid index range is 0 to *arr.length* - 1.

# Arrays

- A loop provides efficient sequential access to the elements in an array.
- Let the loop control variable serve as an index.
- For instance, the following statements declare an array `arr` of 100 integer elements and use a for-loop to initialize the array with values 1, 2, ..., 100.

```
int[] arr = new int[100]; // declare array and allocate space

for (i=0; i < arr.length; i++)
    arr[i] = i+1;
```



# Arrays

- An array can be initialized when it is declared.
- After the array name, use "=" followed by an *array initializer list* which is a comma-separated sequence of values enclosed in braces.
- There is no need to use the operator **new** since the compiler uses the size of the initializer list to allocate memory.

```
int[] intArr = {21, 7, 30, 15};  
String[] day = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri",  
"Sat"};
```

# Arrays

- Java provides an "*enhanced for*" statement that allows for a read-only scan of an array without using indices.
- The syntax includes a declaration of a variable of the array type and the array name separated by a colon (:).

Syntax: `for (Type varName : arrayName)`  
          `. . .`

```
int [] intArr = {6, 1, 8, 4, 9};  
int sum;
```

```
for (int eltValue : intArr)  
    sum += eltValue;
```

# Two-Dimensional Arrays

- A two-dimensional array is a table with access specified by row and column indices.
- Rather than using one pair of square brackets to denote an array, the two-dimensional array uses two pairs of brackets.
- We often refer to a two-dimensional array of numbers as a matrix.

```
int[][] mat; // declare two-dimensional reference variable
// allocate 8 integer locations organized in 2 rows, 4 columns
mat = new int[2][4];
```

Elements of mat are accessed using double-bracket notation

`mat[i][j]` where  $0 \leq i < 2$ ,  $0 \leq j < 4$

# Two-Dimensional Arrays

- The following table displays the elements in mat.
- Nested for-loops scan the elements and compute the sum of the elements.

	0	1	2	3
0	20	5	30	0
1	-40	15	100	80

```
int row, col, sum = 0;
```

```
for (row = 0; row < 2; row++)  
    for (col = 0; col < 4; col++)  
        sum += mat[row][col];
```

# Two-Dimensional Arrays

- A program may access an entire row of the matrix using a row index in brackets.
- The resulting structure is a 1-dimensional array.

```
// mat[0] is the first row in the matrix.  
// as a one-dimensional array,  
// it has an associated length variable  
columnSize = mat[0].length;    // value 4
```

# Two-Dimensional Arrays

- Like a one-dimensional array, you can use an initializer list to allocate a matrix with initial values.

```
int[][] mat = {{20, 5, 30, 0}, {-40, 15, 100, 80}};
```

# Java Methods

- A *method* is a collection of statements that perform a task.
- A method can accept data values, carry out calculations, and return a value.
- The data values serve as input for the method and the return value is its output.

# Java Methods

- The declaration of a method begins with its signature, which consists of modifiers, a return type, a method name, and a parameter list that specifies the formal parameters.
- The code that implements the method is a block called the *method body*.

```
modifiers returnType methodName (Type1 var1, Type2 var2, . . . )  
{  
    // method body  
}
```



# Java Methods

- We are already familiar with the method `main()`.
- The method name is *main* and the return type is `void` since it does not return a value.
- The parameter list is a single variable specifying an array of strings.
- The method has the modifier "*public static*" which indicates that it is associated with the main application class.

```
public static void main(String[] args)
{
    // body is the main program
}
```

# Predefined Methods

- Java provides a collection of methods that implement familiar mathematical functions for scientific, engineering or statistical calculations.
- The methods are defined in the Math class.

# Predefined Methods

// Power Function

```
public static double pow(double x, double y);    //  $x^y$ 
```

// Square Root Function

```
public static double sqrt(double x);
```

// Trigonometric Functions:

// trigonometric cosine

```
public static double cos(double x);
```

// trigonometric sine

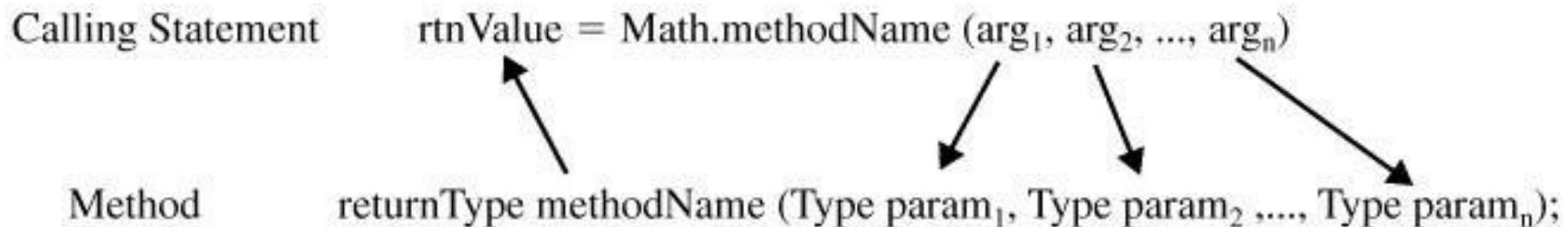
```
public static double sin(double x);
```

// trigonometric tangent

```
public static double tan(double x);
```

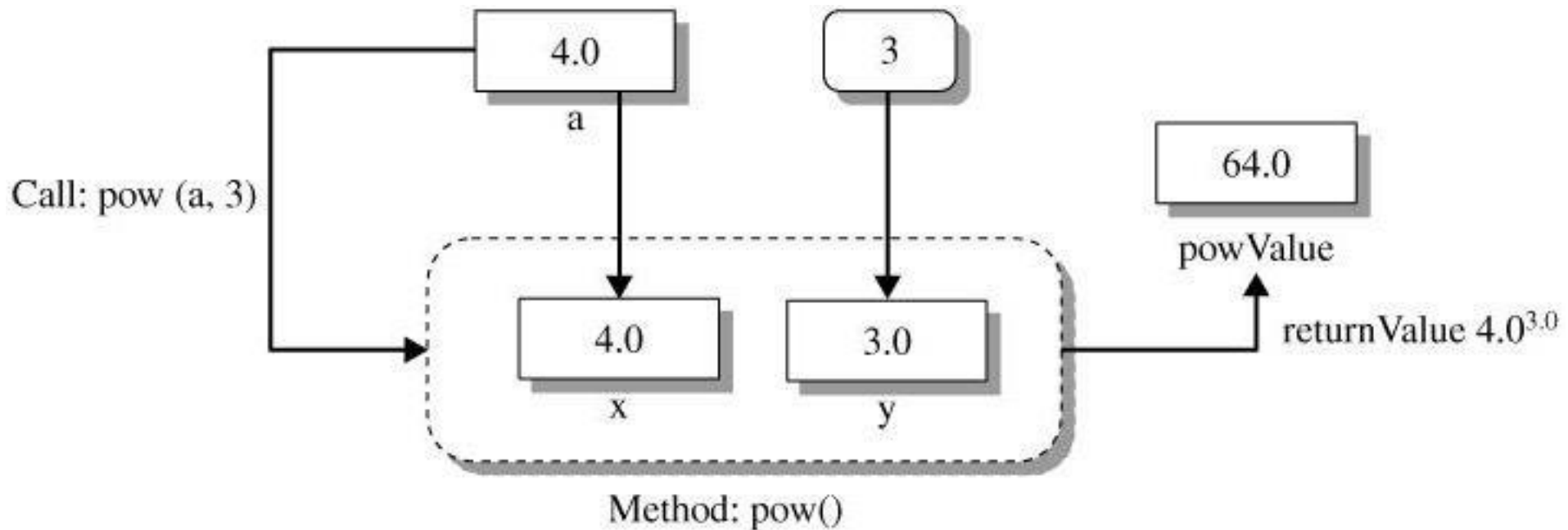
# Predefined Methods

- To access a Java method defined in the Math class, a calling statement must access the method using the syntax *Math.method(arguments)*.
- If not void, the return value may be used in an assignment statement or as part of an expression.



# Predefined Methods

```
// variables for one argument and the return value  
double a = 4.0, powValue;  
// call pow() from Math class; integer 3 is promoted to a double.  
// assign powValue the return value 64.0  
powValue = Math.pow(a, 3);
```



# Predefined Methods

- The Math class defines the constant  $\pi$  as the named constant Math.PI.
- A trigonometric function takes an angle in radians. The expression  $(\text{angle} * (\pi/180))$  converts from degrees to radians.

```
double theta = 0.0, angle = 30;  
System.out.println("cosine of 0 radians is " + Math.cos(theta));  
System.out.println("sine of 30 degrees is " +  
    Math.sin(angle*(Math.PI/180)));
```

```
cosine of 0 radians is 1.0  
sine of 30 degrees is 0.5
```

# User Defined Methods

- In a method, we provide the return value with a statement that uses the reserved word *return* and the value.
- The data type of the return value must be compatible with the return type in the method header.
- The general form of the statement is  
    return expression;

# User Defined Methods

- The method `annuity()` returns the value of an annuity.

```
public static double annuity(double principal, double rate, int nyears)
{
    return principal * Math.pow(1+rate,nyears);
}
```



# User Defined Methods

- A return statement can be used anywhere in the body of the method and causes an exit from the method with the return value passed back to the calling statement.
  - When a method has a returnType other than void, use a return statement with a return value.
  - When the method has a void returnType, the method will return after the last statement. You may force a return using the syntax **return;**

# User Defined Methods

- The method `printLabel()` takes a string argument for a label along with string and integer arguments for the month and year and outputs the label and the month and year separated by a comma (',').

```
public static void printLabel(String label,  
                               String month, int year)  
{  
    System.out.println(label + " " month + ", " + year);  
  
    // a return statement provides explicit exit  
    // from the method; typically we use an implicit  
    // return that occurs after executing the last  
    // statement in the method body  
    return;  
}
```

# User Defined Methods

```
// month and year for purchase
String month = "December";
int year = 1991;

// principal invested and interest earned per year
double principal = 10000.0, interestRate = 0.08,
    annuityValue;

// number of years for the annuity
int nyears = 30;

// call the method and store the return value
annuityValue = annuity(principal, interestRate, nyears);

// output label and a summary description of the annuity
printLabel("Annuity purchased:", month, year);
System.out.println("After " + nyears + " years, $" +
    principal + " at " + interestRate*100 +
    "% grows to $" + annuityValue);
```

# User Defined Methods

Annuity purchased: December, 1991

After 30 years, \$10000.0 at 8.0% grows to \$100626.57

# Arrays as Method Parameters

- A method can include an array among its formal parameters.
- The format includes the type of the elements followed by the symbol "[]" and the array name.

```
returnType methodName (Type[] arr, ...)
```

# Arrays as Method Parameters

- The method `max()` returns the largest element in an array of real numbers.
- The method `signature` includes an array of type `double` as a parameter and a return type `double`.

# Arrays as Method Parameters

```
public static double max(double[] arr)
{
    // assume arr[0] is largest
    double maxValue = arr[0];

    // scan rest of array and update
    // maxValue if necessary
    for (int i = 1; i < arr.length; i++)
        if (arr[i] > maxValue)
            maxValue = arr[i];

    // return largest value which is maxValue
    return maxValue;
}
```

# Arrays as Method Parameters

```
public static double max(double[] arr)
{
    double maxValue = arr[0];

    for (int i = 1; i < arr.length; i++)
        if (arr[i] > maxValue)
            maxValue = arr[i];

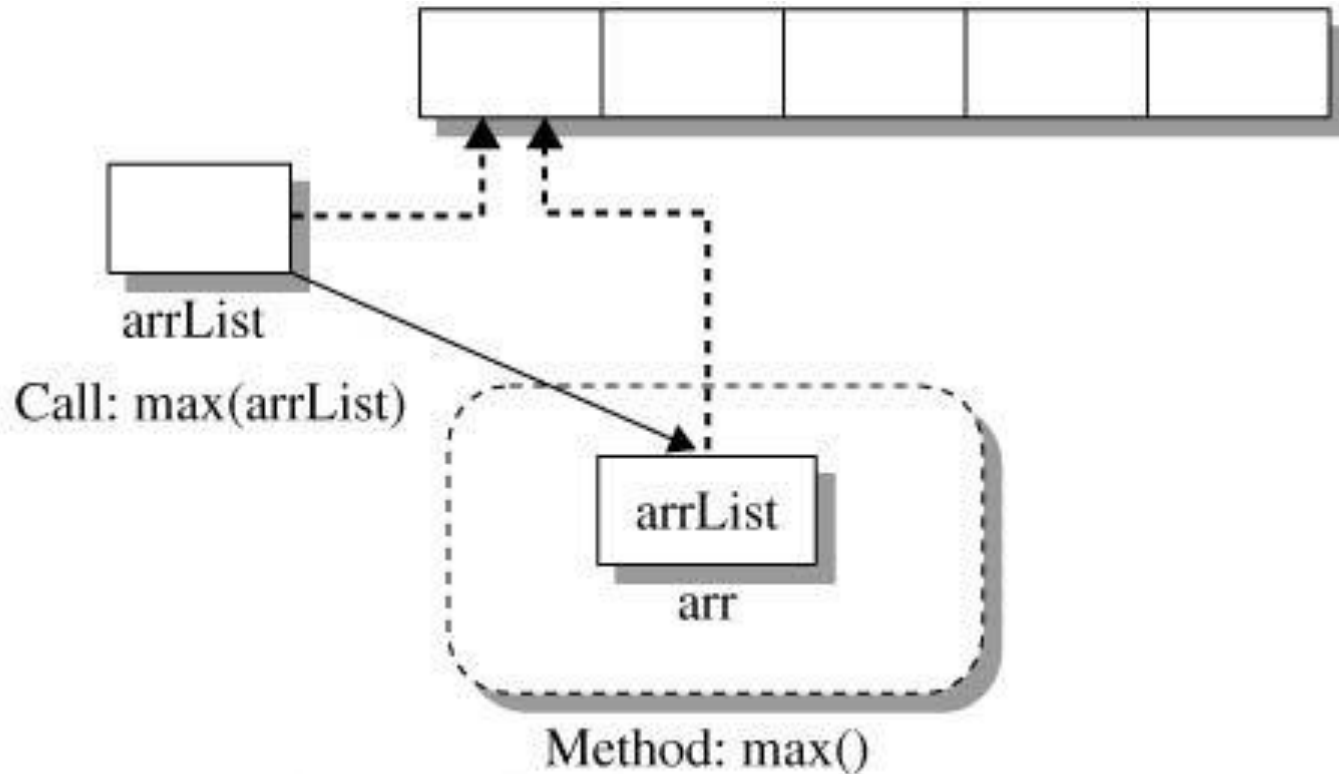
    return maxValue;
}
```



# Arrays as Method Parameters

- In a declaration of an array, the name is a reference that designates the address of the block of memory that holds the array elements.
- When a method has an array parameter, call the method by passing the name of an existing array argument.
- This has the effect of passing a reference to the method identifying the sequence of elements.

# Arrays as Method Parameters



The algorithm for `max()` simply performs a read-only scan of the elements in the array to determine the maximum value.

# Arrays as Method Parameters

- The algorithm for `max()` performs a read-only scan of the elements.
- It is possible to update the array passed to a method.
- Since the array parameter is pointing at the array allocated in the calling program, an update modifies this array and the change remains in effect after returning from the method.

# Arrays as Method Parameters

- The method `maxFirst()` finds the largest element in the tail of an array beginning at index `start` and exchanges it with the element at the index `start`.
- The method has no return value.

# Arrays as Method Parameters

```
public static void maxFirst(int[] arr, int start)
{
    // maxValue and maxIndex are the value and
    // location of the largest element that is
    // identified during a scan of the array
    int maxValue = arr[start], maxIndex = start, temp;

    // scan the tail of the list beginning at index
    // start+1 and update both maxValue and maxIndex
    // so that we know the value and location of the
    // largest element
    for (int i = start+1; i < arr.length; i++)
        if (arr[i] > maxValue)
        {
            maxValue = arr[i];
            maxIndex = i;
        }
}
```

# Arrays as Method Parameters

```
// exchange arr[start] and arr[maxIndex]
temp = arr[start];
arr[start] = arr[maxIndex];
arr[maxIndex] = temp;
}
```

# Arrays as Method Parameters

```
public static void maxFirst(int[] arr, int start)
{
    int maxValue = arr[start], maxIndex = start, temp;

    for (int i = start+1; i < arr.length; i++)
        if (arr[i] > maxValue)
        {
            maxValue = arr[i];
            maxIndex = i;
        }

    temp = arr[start];
    arr[start] = arr[maxIndex];
    arr[maxIndex] = temp;
}
```

# Program A.1

```
// main application class
public class ProgramA_1
{
    public static void main(String[] args)
    {
        int[] intArr = {35, 20, 50, 5, 40, 20, 15, 45};
        int i;

        // scan first n-1 positions in the array where
        // n = intArr.length; call maxFirst() to place
        // largest element from the unsorted tail of
        // the list into position i
        for (i = 0; i < intArr.length-1; i++)
            maxFirst(intArr, i);
    }
}
```



# Program A.1

```
// display the sorted array
for (i = 0; i < intArr.length; i++)
    System.out.print(intArr[i] + " ");
System.out.println();
}

<include code for maxFirst()>
}
```

Run:

50 45 40 35 20 20 15 5