Data Types, Variables and Arrays

Java Is a Strongly Typed Language

- ***Every variable has a type, every expression has a type, and every type is strictly defined.***
- ***All assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.***
- There are no automatic coercions or conversions of conflicting types as in some languages.
- The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

Variable Declarations

- Like most compiled languages, variables must be declared before they can be used.
- Variables are a symbolic name given to a memory location.
- All variables have a type which is enforced by the compiler
  - However, Java does support polymorphism which will be discussed later
- The general form of a variable declaration is:

```
type variable-name [= value][,variable-name[= value]];
```

- Examples:

```
int total;
float xValue = 0.0;
boolean isFinished;
String name;
```

note the semicolon

initialization

Types in Java

- In a variable declaration, the *type* can be:
  - a fundamental data type
  - a class
  - an array
- Java has 8 fundamental data types.
- Fundamental data types are not Object-Oriented. They are included as part of the language primarily for efficiency reasons.
- The eight types are: byte, char, short, int, long, float, double, and boolean.

Primitive Data Types and Operations

Type	Precision	Default Value
byte	**8 bits**	**0**
short	**16 bits**	**0**
int*	**32 bits**	**0**
long	**64 bits**	**0**
char	16 bits	\u0000
float	**32 bits**	**+0.0f**
double	**64 bits**	**+0.0d**
boolean	**-**	**false**

Integral Data Types

- 4 types based on integral values: byte, short, int, long

- All numeric types are signed. There are NO unsigned types in Java.
  - Integrals are stored as 2's compliment.

Type	Size	Range
byte	8 bits	-128 through +127
short	16 bits	-32768 through +32767
int	32 bits	-2147483648 through +2147483647
long	64 bits	-9223372036854775808 through +9223372036854775807

Floating point Data Types

- 2 types based on floating point values: float and double
- Storage conforms to IEEE 754 standard
- Floating point numbers are not accurate. They are an approximation
- floats store 7 significant digits. doubles store 15.

Type	Size	Range
float	32 bits	$-3.4 * 10^{38}$ through $+3.4 * 10^{38}$
double	64 bits	$-1.7 * 10^{308}$ through $+1.7 * 10^{308}$

Character data type

- The char type defines a single character
- In many other programming languages, character types are 8-bits (they store ASCII values). In Java, character types are 16-bits.
- Java characters store characters in *unicode* format.
- Unicode is an international character set which defines characters and symbols from several different world languages.
  - Unicode includes ASCII at its low range (0-255)
- Characters can be converted to integers to perform mathematical functions on them.


```
// Demonstrate char data type.
class CharDemo {
  public static void main(String args[]) {
    char ch1, ch2;

    ch1 = 88;  // code for X
    ch2 = 'Y';

    System.out.print("ch1 and ch2: ");
    System.out.println(ch1 + " " + ch2);
  }
}
```

This program displays the following output:
ch1 and ch2: X Y


```
// char variables behave like integers.
class CharDemo2 {
  public static void main(String args[]) {
    char ch1;

    ch1 = 'X';
    System.out.println("ch1 contains " + ch1);

    ch1++; // increment ch1
    System.out.println("ch1 is now " + ch1);
  }
}
```

Boolean data type

- The boolean type defines a truth value: true or false.
- booleans are often used in control structures to represent a condition or state.
- booleans CANNOT be converted to an integer type.


```
// Demonstrate boolean values.
class BoolTest {
  public static void main(String args[]) {
    boolean b;

    b = false;
    System.out.println("b is " + b);
    b = true;
    System.out.println("b is " + b);
    // a boolean value can control the if statement
    if(b) System.out.println("This is executed.");

    b = false;
    if(b) System.out.println("This is not executed.");

    // outcome of a relational operator is a boolean value
    System.out.println("10 > 9 is " + (10 > 9));
  }
}
```

- **a boolean value is output by println( ), "true" or "false" is displayed.**

- **the value of a boolean variable is sufficient, by itself, to control the if statement.**

There is no need to write an if statement like this:

```
if(b == true) ...
```

- **the outcome of a relational operator, such as <, is a boolean value.**

This is why the expression 10 > 9 displays the value "true." Further, the extra set of parentheses around 10 > 9 is necessary because the + operator has a higher precedence than the >.

Literal Definitions

- Integrals can be defined in decimal, octal, or hexadecimal
  - Integrals can be long or int (L or l)
  - Integrals are int by default
- Floating point numbers can be defined using standard or scientific notation
  - double by default
  - F indicates float
- Single characters are defined within single quotes
  - can be defined as unicode
  - can be "special" character (eg. '\n')
- Strings are defined by double quotes.

Decimal: 0 1 10 56 -35685
Octal: 01 056 07735
Hex: 0x1 0x6F 0xFFFF
long: 7L 071L 0x4FFL

Standard: 3.14 9.9 -37.1
Scientific: 6.79e29
float: 7.0F -3.2F

'c' '\n' '\r' '\025' \u34F6

"this is a String."

Valid Control Character

- Valid control character are:
  - \b backspace
  - \t horizontal tab
  - \n linefeed
  - \f formfeed
  - \r carriage return
  - \” double quote
  - \’ single quote
  - \\ backslash


Java

```
class Shuffle1 {

  public static void main(String [] args) {
```

```
if (x > 2) {
  System.out.print("a");
}
```

```
int x = 3;
```

```
if (x == 1) {
  System.out.print("d");
  x = x - 1;
}
```

```
if (x == 2) {
  System.out.print("b c");
}
```

```
x = x - 1;
System.out.print("-");
```

```
while (x > 0) {
```

```
File Edit Window Help Sleep
% java Shuffle1
a-b c-d
```

Variable/Identifier names

- Java has a series of rules which define valid variable names and identifiers.
  - Identifiers can contain letters, numbers, the underscore (_) character and the dollar sign character($)
  - Identifiers must start with a letter, underscore or dollar sign.
  - Identifiers are case sensitive
  - Identifiers cannot be the same as reserved Java keywords.

Tips for good variable names

- Use a naming convention
- Use names which are meaningful within their context
- Start Class names with an Upper case letter. Variables and other identifiers should start with a lower case letter.
- Avoid using _ and $.
- Avoid prefixing variable names (eg. _myAge, btnOk)
  - This is often done in languages where type is not strongly enforced.
  - If you do this in Java, it is often an indication that you have not chosen names meaningful within their context.
- Separate words with a capital letter, not an underscore (_)
  - myAccount, okButton, aLongVariableName
  - avoid: my_account, ok_button, and a_long_variable_name

Variable/Identifier names

_myName

_total

___total5

My-Name

total#

$total36_51$

myName

total

total5

default

total5$

1myName

Identifiers
Valid / Invalid ???

AvgTemp	count	a4	$test	this_is_ok

2count	high-temp	Not/ok

Reserved Words in Java

boolean
byte
char
short
int
long
float
double
void

false
null
true

abstract
final
native
private
protected
public
static
synchronized
transient
volatile

break
case
catch
continue
default
do
else
finally
for
if
return
switch
throw
try
while

class
extends
implements
interface
throws

import
package

instanceof
new
super
this

byvalue
cast
const
future
generic
goto
inner
operator
outer
rest
var

reserved for future use.

Variables

- **Declaring a Variable**
- In Java, all variables must be declared before they can be used. The basic form of a variable
- declaration is shown here:
- *type identifier [= value],[identifier [= value] ...] ;*


```
int a, b, c;                // declares three ints, a, b, and c.

int d = 3, e, f = 5;        // declares three more ints,
                            // initializing d and f.

byte z = 22;                // initializes z.

double pi = 3.14159;        // declares an approximation of pi.

char x = 'x';               // the variable x has the value 'x'.
```

Dynamic Initialization


```
// Demonstrate dynamic initialization.
class DynInit {
    public static void main(String args[]) {
      double a = 3.0, b = 4.0;

      // c is dynamically initialized
      double c = Math.sqrt(a * a + b * b);

      System.out.println("Hypotenuse is " + c);
    }
}
```

The Scope and Lifetime of Variables

- In Java, the two major scopes are those **defined by a class and those defined by a method.**
- the class scope has several unique properties and attributes that do not apply to the scope defined by a method
- The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope

variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope.

Thus, protecting it from unauthorized access and/or modification. (encapsulation)

<u>Scopes can be nested</u>.

For example, each time you create a block of code, you are creating a new, nested scope.

When this occurs, the outer scope encloses the inner scope. This means that ***objects declared in the outer scope will be visible to code within the inner scope***. However, the reverse is not true. ***Objects declared within the inner scope will not be visible outside it.***


```
// Demonstrate block scope.
class Scope {
  public static void main(String args[]) {
    int x; // known to all code within main

    x = 10;
    if(x == 10) { // start new scope
      int y = 20; // known only to this block

      // x and y both known here.
      System.out.println("x and y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Error! y not known here

    // x is still known here.
    System.out.println("x is " + x);
  }
}
```

- **Within a block, variables can be declared at any point, but are valid only after they are declared.**
- **variables are created when their scope is entered, and destroyed when their scope is left.**
- Therefore, variables declared within a method will not hold their values between calls to that method.
- Also, a variable declared within a block will lose its value when the block is left. Thus, **the lifetime of a variable is confined to its scope.**
- If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered.


```
// Demonstrate lifetime of a variable.
class LifeTime {
  public static void main(String args[]) {
    int x;

    for(x = 0; x < 3; x++) {
      int y = -1; // y is initialized each time block is entered
      System.out.println("y is: " + y);

      y = 100;
      System.out.println("y is now: " + y);
    }
  }
}
```


```
class ScopeErr {
  public static void main(String args[])

 {
    int bar = 1;
    {
      int bar = 2;
    }
  }
}
```

Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope.

Constant Values

- A variable can be made constant by including the keyword *final* in its declaration.
- By convention, the names of variables defined as final are UPPER CASE.
- Constants allow for more readable code and reduced maintenance costs.
- Final variables must be initialized upon declaration.

```
final int MAX_BUFFER_SIZE = 256;
final float PI=3.14159;
```

Type Conversion and Casting

Java's Automatic Conversions

- When one type of data is assigned to another type of variable, an ***automatic type conversion*** will take place if the following two conditions are met:

- • *The two types are compatible.*
- • *The destination type is larger than the source type*.

- When these two conditions are met, a *widening conversion takes place*

Widening Conversions

- A widening conversion occurs when a value stored in a smaller space is converted to a type of a larger space.
  - There will never be a loss of information
- Widening conversions occur automatically when needed.

Original Type	Automatically converted to:
byte (8 bits)	char, short, int, long, float or double
char (16 bits)	int, long, float, or double
short (16 bits)	int, long, float, or double
int (32 bits)	long, float, double
float (32 bits)	double

•For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other.

•there are no automatic conversions from the numeric types to **char or boolean. Also, char and boolean are not compatible with each other.**

•Java also performs an automatic type conversion when storing a literal integer constant into variables of type **byte, short, long, or char.**

Casting Incompatible Types

- To create a conversion between two incompatible types, you must use a cast.
- an explicit type conversion.
- It has this general form:

(target-type) value

- **int to a byte** reduced to modulo byte's range.
- **Float to a int** *truncation.*
  - Information may be lost
  - Never occurs automatically. Must be explicitly requested by the programmer using a cast.

Narrowing Conversions

Original Type	Narrowing conversions to:
char (16 bits)	byte or short
short (16 bits)	byte or char
int (32 bits)	byte, char, or short
long	byte, char, short, or int
float (32 bits)	byte, char, short, int, or long
double (32 bits)	byte, char, short, int, long, or float


```
class Conversion {
  public static void main(String args[]) {
    byte b;
    int i = 257;
    double d = 323.142;

    System.out.println("\nConversion of int to byte.");
    b = (byte) i;
    System.out.println("i and b " + i + " " + b);

    System.out.println("\nConversion of double to int.");
    i = (int) d;
    System.out.println("d and i " + d + " " + i);

    System.out.println("\nConversion of double to byte.");
    b = (byte) d;
    System.out.println("d and b " + d + " " + b);
  }
}
```

Automatic Type Promotion in Expressions

- examine the following expression:

byte a = 40;

byte b = 50;

byte c = 100;

int d = a * b / c;

- *Java automatically promotes each byte, short, or char operand to int when evaluating an expression*

examine the following expression

```
byte b = 50;
b = b * 2;
```

ERROR WHY?

byte b = 50;

b = (byte)(b * 2);

The Type Promotion Rules

- all byte, short, and char values are promoted to int
- if one operand is a long, the whole expression is promoted to long.
- If one operand is a float, the entire expression is promoted to float.
- If any of the operands is double, the result is double


```
class Promote  {
  public static void main(String args[]) {
    byte b = 42;
    char c = 'a';
    short s = 1024;
    int i = 50000;
    float f = 5.67f;
    double d = .1234;

    double result = (f * b) + (i / c) - (d * s);

    System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));

    System.out.println("result = " + result);
  }
}
```


Java

Arrays

Definition:

An array is a group/collection of variables of the same type that are referred to by a common name and an index

Examples:

- Collection of numbers
- Collection of names
- Collection of suffixes

Examples

Array of numbers:

10	23	863	8	229

Array of names:

Sholay	Shaan	Shakti

Array of suffixes:

ment	tion	ness	ves

Analogy

Array is like a pen box with fixed no. of slots of same size.

Syntax

Declaration of array variable:

data-type variable-name[];

eg. *int marks[];*
This will declare an array named ‘marks’ of type ‘int’. But no memory is allocated to the array.

Allocation of memory:

variable-name = new data-type[size];

eg. *marks = new int[5];*
This will allocate memory of 5 integers to the array ‘marks’ and it can store upto 5 integers in it. ‘new’ is a special operator that allocates memory.

Syntax…

Accessing elements in the array:

Specific element in the array is accessed by specifying name of the array followed the index of the element.

All array indexes in Java start at zero.

variable-name[index] = value;

eg. *marks[0] = 10;*

marks[2] = 863;

Example

STEP 1 : (Declaration)
int marks[];
marks → null

STEP 2: (Memory Allocation)
marks = new int[5];

marks →

0	0	0	0	0

marks[0] marks[1] marks[2] marks[3] marks[4]

STEP 3: (Accessing Elements)
marks[0] = 10;

marks →

10	0	0	0	0

marks[0] marks[1] marks[2] marks[3] marks[4]

Program


```
class try_array {

        public static void main(String args[]) {
                int marks[];
                marks = new int[3];

                marks[0] = 10;
                marks[1] = 35;
                marks[2] = 84;

System.out.println("Marks obtained by 2nd student=" + marks[1]);

        }

}
```

Alternative Syntax

Combined declaration & memory allocation:

data-type variable-name[] = new data-type[size];

eg. *int marks[] = new int[5];*

This will declare an int array ‘marks’ and will also allocate memory of 5 integers to it.

Alternative Syntax

Combined declaration, allocation & assignment:

data-type variable-name[] = {comma-separated values};

eg. *int marks[] = {10, 35, 84, 23, 5};*

This will declare an int array ‘marks’, will allocate memory of 5 integers to it and will also assign the values as-

marks →

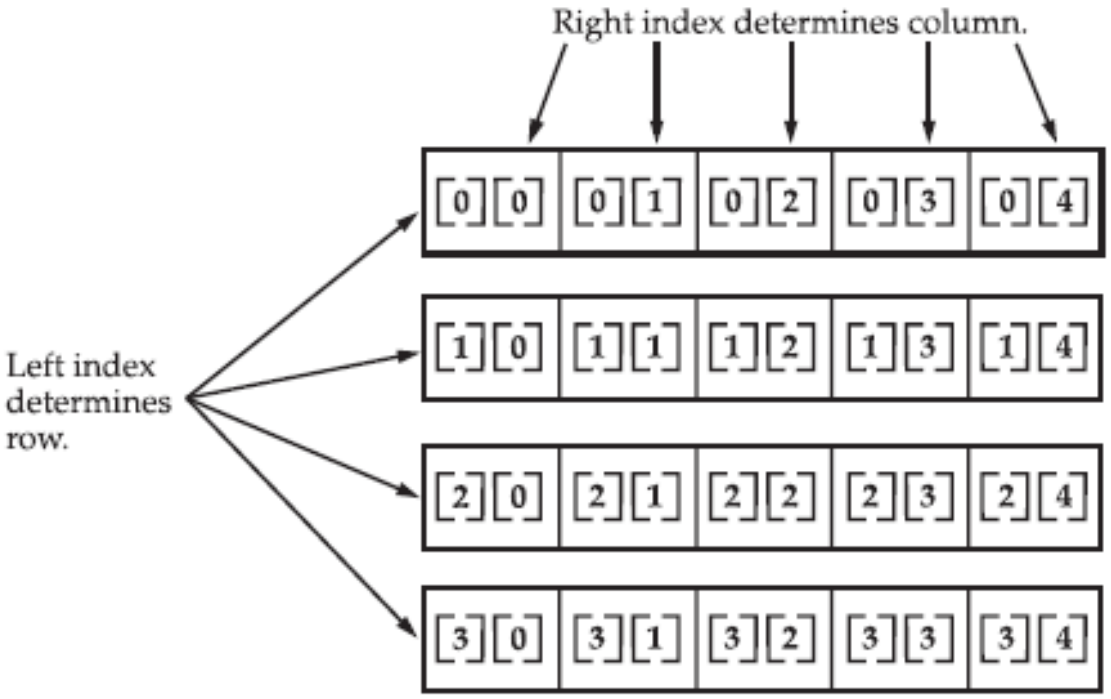
10	35	84	23	5

marks[0] marks[1] marks[2] marks[3] marks[4]

- Java strictly checks to make sure you do not accidentally try to store or reference values outside of the range of the array.
- The Java run-time system will check to be sure that all array indexes are in the correct range.

Multidimensional Arrays

- In Java, *multidimensional arrays are actually arrays of arrays*
- ***int twoD[ ][ ] = new int[4][5];***
- This allocates a 4 by 5 array and assigns it to twoD.
- Internally this matrix is implemented as an ***array of arrays of int***



Given: int twoD [ ] [ ] = new int [4] [5] ;


```
class TwoDArray {
  public static void main(String args[]) {
    int twoD[][]= new int[4][5];
    int i, j, k = 0;

    for(i=0; i<4; i++)
      for(j=0; j<5; j++) {
        twoD[i][j] = k;
        k++;
      }

    for(i=0; i<4; i++) {
      for(j=0; j<5; j++)
        System.out.print(twoD[i][j] + " ");
      System.out.println();     }   }
}
```

- When you allocate memory for a multidimensional array, you need only specify the memory for the first **(leftmost) dimension**.
- You can allocate the remaining dimensions separately.
- For example, this following code allocates memory for the first dimension of **twoD when it is declared. It allocates the second dimension manually.**

int twoD[ ][ ] = new int[4][ ];

twoD[0] = new int[5];

twoD[1] = new int[5];

twoD[2] = new int[5]; *twoD[3] = new int[5];*


```
class TwoDAgain {
  public static void main(String args[]) {
    int twoD[][] = new int[4][];
    twoD[0] = new int[1];
    twoD[1] = new int[2];
    twoD[2] = new int[3];
    twoD[3] = new int[4];

    int i, j, k = 0;
```

```
for(i=0; i<4; i++)
      for(j=0; j<i+1; j++) {
        twoD[i][j] = k;
        k++;
      }

    for(i=0; i<4; i++) {
      for(j=0; j<i+1; j++)
        System.out.print(twoD[i][j] + " ");
      System.out.println();
    }
  }
}
```

The array created by this program looks like this:

[0][0]			
[1][0]	[1][1]		
[2][0]	[2][1]	[2][2]	
[3][0]	[3][1]	[3][2]	[3][3]

possible to initialize multidimensional arrays


```
class Matrix {
  public static void main(String args[]) {
    double m[][] = {
      { 0*0, 1*0, 2*0, 3*0 },
      { 0*1, 1*1, 2*1, 3*1 },
      { 0*2, 1*2, 2*2, 3*2 },
      { 0*3, 1*3, 2*3, 3*3 }
    };
    int i, j;

    for(i=0; i<4; i++) {
      for(j=0; j<4; j++)
        System.out.print(m[i][j] + " ");
      System.out.println();
    }
  }
}
```


```
// Demonstrate a three-dimensional array.
class ThreeDMatrix {
  public static void main(String args[]) {
    int threeD[][][] = new int[3][4][5];
    int i, j, k;

    for(i=0; i<3; i++)
      for(j=0; j<4; j++)
        for(k=0; k<5; k++)
        threeD[i][j][k] = i * j * k;

    for(i=0; i<3; i++) {
      for(j=0; j<4; j++) {
        for(k=0; k<5; k++)
          System.out.print(threeD[i][j][k] + " ");
        System.out.println();
      }
      System.out.println();
    }   }   }
```

This program generates the following output:

```
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  2  3  4
0  2  4  6  8
0  3  6  9  12
```

```
0  0  0  0  0
0  2  4  6  8
0  4  8  12  16
0  6  12  18  24
```

- **Alternative Array Declaration Syntax**
- There is a second form that may be used to declare an array:

Type[ ] var-name;

```
int a1[] = new int[3];
int[] a2 = new int[3];
```

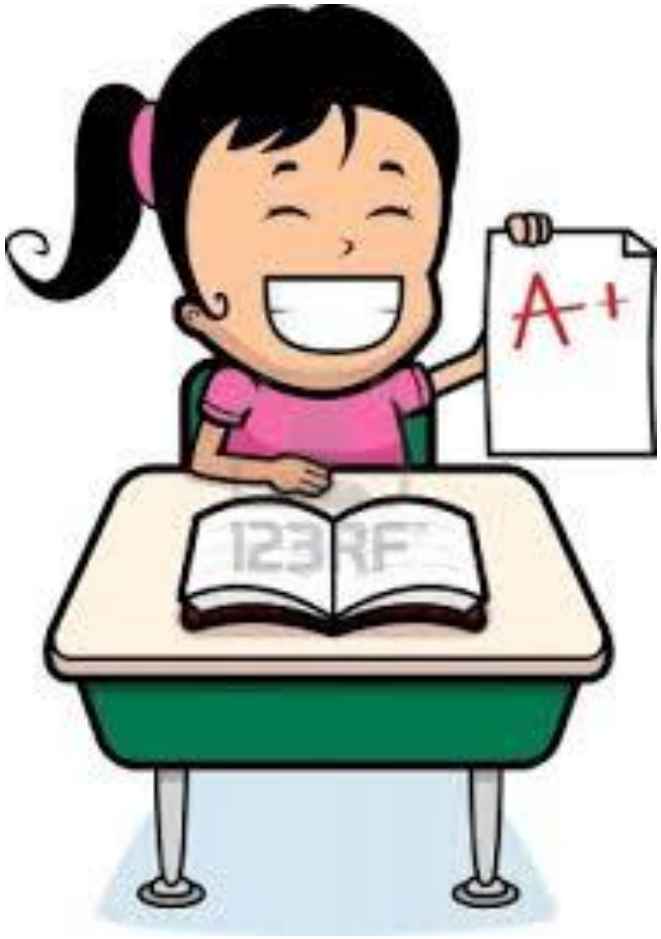
```
char twod1[][] = new char[3][4];
char[][] twod2 = new char[3][4];
```

```
int[] nums, nums2, nums3; // create three arrays
```

String

- The **String type is used to declare string variables.**
- **can declare arrays of strings.**
- A quoted string constant can be assigned to a **String variable.**
- **A variable of type String can** argument to **println( )**

```
String str = "this is a test";
System.out.println(str);
```


A+


Java