IT2205: Programming I

Section 3

Fundamentals of Java Programming (15 hrs)

© 2014, University of Colombo School of Computing

Section 3.1

Data types and variables in Java

© 2014, University of Colombo School of Computing

Data types and variables in Java

- Java Is a Strongly Typed Language
  - It is important to state at the outset that Java is a strongly typed language.
  - Indeed, part of Java's safety and robustness comes from this fact.
  - Let's see what this means. First, every variable has a type, every expression has a type, and every type is strictly defined.
  - Second, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.

Data types and variables in Java Cont...

- Java Is a Strongly Typed Language cont…
  - 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.

© 2014, University of Colombo School of Computing

Data types and variables in Java Cont...

- The primitive types
  - Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean. The primitive types are also commonly referred to as simple types, and both terms will be used in this book. These can be put in four groups:
  - Integers This group includes byte, short, int, and long, which are for whole-valued signed numbers.
  - Floating-point numbers This group includes float and double, which represent numbers with fractional precision.

Data types and variables in Java Cont...

- The primitive types cont…
  - Characters This group includes char, which represents symbols in a character set, like letters and numbers.
  - Boolean This group includes boolean, which is a special type for representing true/false values.
  - One can use these types as-is, or to construct arrays or your own class types. Thus, they form the basis for all other types of data that you can create.

Data types and variables in Java Cont…

- The primitive types cont…
  - The primitive types represent single values—not complex objects. Although Java is otherwise completely object-oriented, the primitive types are not. They are analogous to the simple types found in most other non–object-oriented languages. The reason for this is efficiency. Making the primitive types into objects would have degraded performance too much.
  - The primitive types are defined to have an explicit range and mathematical behavior. Languages such as C and C++ allow the size of an integer to vary based upon the dictates of the execution environment. However, Java is different. Because of Java's portability requirement, all data types have a strictly defined range. For example, an int is always 32 bits, regardless of the particular platform. This allows programs to be written that are guaranteed to run without porting on any machine architecture. While strictly specifying the size of an integer may cause a small loss of performance in some environments, it is necessary in order to achieve portability.

Data types and variables in Java Cont…

- The primitive types cont…
  - One can use these types as-is, or to construct arrays or your own class types. Thus, they form the basis for all other types of data that you can create.
  - The primitive types represent single values—not complex objects. Although Java is otherwise completely object-oriented, the primitive types are not. They are analogous to the simple types found in most other non–object-oriented languages. The reason for this is efficiency. Making the primitive types into objects would have degraded performance too much.
  - The primitive types are defined to have an explicit range and mathematical behavior. Languages such as C and C++ allow the size of an integer to vary based upon the dictates of the execution environment. However, Java is different. Because of Java's portability requirement, all data types have a strictly defined range. For example, an int is always 32 bits, regardless of the particular platform. This allows programs to be written that are guaranteed to run without porting on any machine architecture. While strictly specifying the size of an integer may cause a small loss of performance in some environments, it is necessary in order to achieve portability.

© 2014, University of Colombo School of Computing

Data types and variables in Java Cont…

- Primitive types cont…

  Integers

  – Java defines four integer types: byte, short, int, and long. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers. Many other computer languages support both signed and unsigned integers. However, Java's designers felt that unsigned integers were unnecessary. Specifically, they felt that the concept of unsigned was used mostly to specify the behavior of the high-order bit, which defines the sign of an integer value. Java manages the meaning of the high-order bit differently, by adding a special "unsigned right shift" operator. Thus, the need for an unsigned integer type was eliminated.

  – The width of an integer type should not be thought of as the amount of storage it consumes, but rather as the behavior it defines for variables and expressions of that type. The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them.

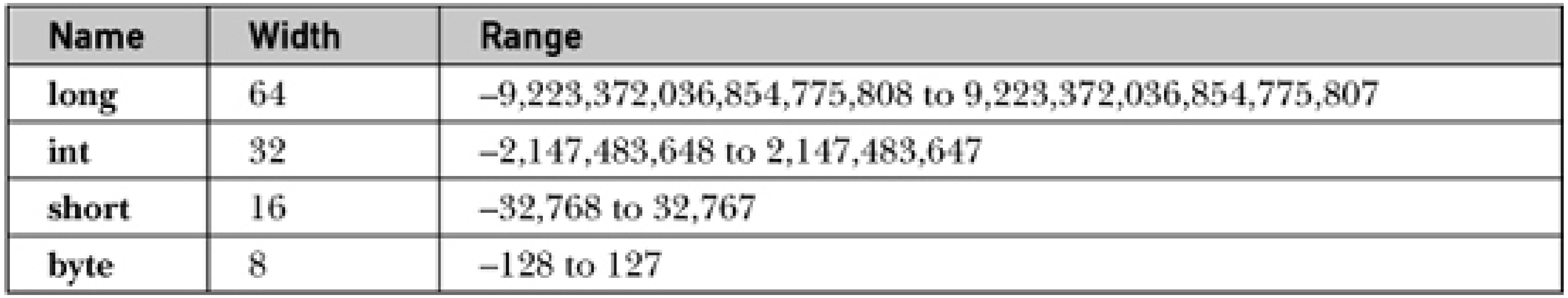
© 2014, University of Colombo School of Computing

Data types and variables in Java Cont…

- Primitive types cont…

Integers

– The width and ranges of these integer types vary widely, as shown in this table:



Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

© 2014, University of Colombo School of Computing

Data types and variables in Java Cont…

- Primitive types cont…

Let’s look at each type of integer.(byte)

– The smallest integer type is byte. This is a signed 8-bit type that has a range from –128 to 127. Variables of type byte are especially useful when you’re working with a stream of data from a network or file. They are also useful when you’re working with raw binary data that may not be directly compatible with Java’s other built-in types.

– Byte variables are declared by use of the byte keyword. For example, the following declares two byte variables called b and c:

```
byte b, c;
```

Data types and variables in Java Cont…

- Primitive types cont…

Let’s look at each type of integer.(short)

– **short is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the least-used Java type. Here are some examples of short variable declarations:**

```
short s;
short t;
```

(A note to remember: do any operation on s and t and assign it to a short value)

Data types and variables in Java Cont…

- Primitive types cont…

  Let's look at each type of integer.(int)

  – **The most commonly used integer type is int. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647.**

  – **In addition to other uses, variables of type int are commonly employed to control loops and to index arrays.**

  – **Although you might think that using a byte or short would be more efficient than using an int in situations in which the larger range of an int is not needed, this may not be the case.**

  – **The reason is that when byte and short values are used in an expression they are promoted to int when the expression is evaluated**

  – **Therefore, int is often the best choice when an integer is needed.**

Data types and variables in Java Cont…

- Primitive types cont…

  Let’s look at each type of integer.(int)

  - **The most commonly used integer type is int. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647.**
  - **In addition to other uses, variables of type int are commonly employed to control loops and to index arrays.**
  - **Although you might think that using a byte or short would be more efficient than using an int in situations in which the larger range of an int is not needed, this may not be the case.**
  - **The reason is that when byte and short values are used in an expression they are promoted to int when the expression is evaluated**
  - **Therefore, int is often the best choice when an integer is needed.**

© 2014, University of Colombo School of Computing

Data types and variables in Java Cont…

- Primitive types cont…

Let's look at each type of integer.(long)

- **long is a signed 64-bit type and is useful for those occasions where an int type is not large enough to hold the desired value.**
- **The range of a long is quite large.**
- **This makes it useful when big, whole numbers are needed.**

Data types and variables in Java Cont…

- Primitive types cont…

Let's look at each type of integer.(long)

- **long is a signed 64-bit type and is useful for those occasions where an int type is not large enough to hold the desired value.**
- **The range of a long is quite large.**
- **This makes it useful when big, whole numbers are needed.**

© 2014, University of Colombo School of Computing

Data types and variables in Java Cont…

- Primitive types cont…

  Let’s look at each type of floating point types

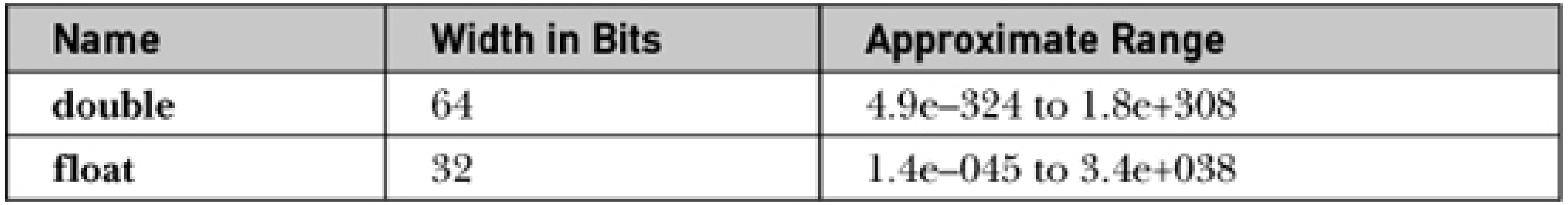
  - **Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root, or transcendentals such as sine and cosine, result in a value whose precision requires a floating-point type.**
  - **Java implements the standard (IEEE–754) set of floating-point types and operators.**
  - **There are two kinds of floating-point types, float and double, which represent single- and double-precision numbers, respectively.**

Data types and variables in Java Cont…

- Primitive types cont…

  Let’s look at each type of floating point types

  – **Width and ranges of float and double are shown here:**



Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

© 2014, University of Colombo School of Computing

Data types and variables in Java Cont…

- Primitive types cont…
  - **Let's look at each type of floating point types(float)**

The type float specifies a single-precision value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type float are useful when you need a fractional component, but don't require a large degree of precision. For example, float can be useful when representing dollars and cents.

Here are some example float variable declarations:

```
float hightemp, lowtemp;
```

Data types and variables in Java Cont…

- Primitive types cont…
  - Let's look at each type of floating point types(double)

Double precision, as denoted by the double keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as sin( ), cos( ), and sqrt( ), return double values. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, double is the best choice.

Here are some examples of the usage of double date type:

```
double pi, r, a;
```

Data types and variables in Java Cont…

- Primitive types cont…

Let's look at the character type (char)

- **In Java, the data type used to store characters is char. However, C/C++ programmers beware: char in Java is not the same as char in C or C++. In C/C++, char is 8 bits wide.**
- **This is not the case in Java. Instead, Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more.**
- **For this purpose, it requires 16 bits. Thus, in Java char is a 16-bit type. The range of a char is 0 to 65,536. There are no negative chars. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255.**

Data types and variables in Java Cont…

- Primitive types cont…

Let's look at the character type (char) cont…

- **Since Java is designed to allow programs to be written for worldwide use, it makes sense that it would use Unicode to represent characters. Of course, the use of Unicode is somewhat inefficient for languages such as English, German, Spanish, or French, whose characters can easily be contained within 8 bits. But such is the price that must be paid for global portability.**
- **In Java, the data type used to store characters is char. However, C/C++ programmers beware: char in Java is not the same as char in C or C++. In C/C++, char is 8 bits wide.**
- **This is not the case in Java. Instead, Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more.**

Data types and variables in Java Cont…

➢ Primitive types cont…

Let's look at the character type (char) cont…

- **For this purpose, it requires 16 bits. Thus, in Java char is a 16-bit type. The range of a char is 0 to 65,536. There are no negative chars. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255.**
- **Since Java is designed to allow programs to be written for worldwide use, it makes sense that it would use Unicode to represent characters. Of course, the use of Unicode is somewhat inefficient for languages such as English, German, Spanish, or French, whose characters can easily be contained within 8 bits. But such is the price that must be paid for global portability.**

```
Eg: char ch1, ch2;      ch1 = 88; // code for X
    ch2 = 'Y';
```

Data types and variables in Java Cont…

- Primitive types cont…

Let’s look at the character type (char)

- **Although char is designed to hold Unicode characters, it can also be used as an integer type on which you can perform arithmetic operations. For example, you can add two characters together, or increment the value of a character variable.**

Eg:
```
char ch1;
ch1 = 'X';
System.out.println("ch1 contains " + ch1);
ch1++; // increment ch1
```

Data types and variables in Java Cont…

- Primitive types cont…

Let’s look at the boolean type

- **Java has a primitive type, called boolean, for logical values. It can have only one of two possible values, true or false.**
- **This is the type returned by all relational operators, as in the case of a < b. boolean is also the type required by the conditional expressions that govern the control statements such as if and for.**

Data types and variables in Java Cont…

Primitive Data Types

- There are four primitive data types that store integers. These differ in the number of memory bits they are allocated in memory.
- Similarly, there are two primitive data types that store reals.

type	**bits**
byte	8
short	16
int	32
long	64

type	**bits**
float	32
double	64

© 2014, University of Colombo School of Computing

Data types and variables in Java Cont...

Primitive Data Types Cont...

- The number of bits used to store an integer or a real determines the range of values the variable is capable of storing.

type	range
byte	**-128** to **127**
short	**-32,768** to **32,767**
int	**-2,147,483,648** to **2,147,483,647**
long	**-9,223,372,036,854,775,808** to **9,223,372,036,854,775,807**

type	range
float	**4.9e-324** to **1.8e+308**
double	**1.4e-45** to **3.4e-038**

Data types and variables in Java Cont…

- Closer look at literals - Integer Literals
  - Integers are probably the most commonly used type in the typical program. Any whole number value is an integer literal. Examples are 1, 2, 3, and 42. These are all decimal values, meaning they are describing a base 10 number.
  - There are two other bases which can be used in integer literals, octal (base eight) and hexadecimal (base 16). Octal values are denoted in Java by a leading zero.
  - Normal decimal numbers cannot have a leading zero. Thus, the seemingly valid value 09 will produce an error from the compiler, since 9 is outside of octal’s 0 to 7 range. A more common base for numbers used by programmers is hexadecimal, which matches cleanly with modulo 8 word sizes, such as 8, 16, 32, and 64 bits. You signify a hexadecimal constant with a leading zero-x, (0x or 0X). The range of a hexadecimal digit is 0 to 15, so A through F (or a through f) are substituted for 10 through 15.

Data types and variables in Java Cont…

- Closer look at literals - Integer Literals cont…
  - Integer literals create an int value, which in Java is a 32-bit integer value. Since Java is strongly typed,
  - you might be wondering how it is possible to assign an integer literal to one of Java's other integer types, such as byte or long, without causing a type mismatch error. Fortunately, such situations are easily handled. When a literal value is assigned to a byte or short variable, no error is generated if the literal value is within the range of the target type.
  - An integer literal can always be assigned to a long variable. However, to specify a long literal, you will need to explicitly tell the compiler that the literal value is of type long. You do this by appending an upper- or lowercase L to the literal. For example, 0x7fffffffffffffffL or 9223372036854775807L is the largest long. An integer can also be assigned to a char as long as it is within range.

© 2014, University of Colombo School of Computing

Data types and variables in Java Cont…

- Closer look at literals - Integer Literals cont…
  - **Beginning with JDK 7, you can also specify integer literals using binary. To do so, prefix the value with 0b or 0B. For example, this specifies the decimal value 10 using a binary literal: int x = 0b1010;**
  - **Among other uses, the addition of binary literals makes it easier to enter values used as bitmasks. In such a case, the decimal (or hexadecimal) representation of the value does not visually convey its meaning relative to its use. The binary literal does.**
  - **Also beginning with JDK 7, you can embed one or more underscores in an integer literal. Doing so makes it easier to read large integer literals. When the literal is compiled, the underscores are discarded.**

    For example, given int x = 123_456_789;
  - **the value given to x will be 123,456,789. The underscores will be ignored. Underscores can only be used to separate digits. They cannot come at the beginning or the end of a literal. It is, however, permissible for more than one underscore to be used between two digits. For example, this is valid: int x = 123___456___789;**

© 2014, University of Colombo School of Computing

Data types and variables in Java Cont…

- Closer look at literals - Integer Literals cont…
  - the value given to x will be 123,456,789. The underscores will be ignored. Underscores can only be used to separate digits. They cannot come at the beginning or the end of a literal. It is, however, permissible for more than one underscore to be used between two digits. For example, this is valid: int x = 123___456___789;
  - The use of underscores in an integer literal is especially useful when encoding such things as telephone numbers, customer ID numbers, part numbers, and so on. They are also useful for providing visual groupings when specifying binary literals. For example, binary values are often visually grouped in four-digits units, as shown here:

```
int x = 0b1101_0101_0001_1010;
```

Data types and variables in Java Cont…

➢ Closer look at literals - Floating Point Literals

– Floating-point numbers represent decimal values with a fractional component. They can be expressed in either standard or scientific notation. Standard notation consists of a whole number component followed by a decimal point followed by a fractional component.

– For example, 2.0, 3.14159, and 0.6667 represent valid standard-notation floating-point numbers. Scientific notation uses a standard-notation, floating-point number plus a suffix that specifies a power of 10 by which the number is to be multiplied. The exponent is indicated by an E or e followed by a decimal number, which can be positive or negative. Examples include 6.022E23, 314159E–05, and 2e+100.

Data types and variables in Java Cont…

➢ Closer look at literals - Floating Point Literals cont…

– **Floating-point literals in Java default to double precision. To specify a float literal, you must append an F or f to the constant. You can also explicitly specify a double literal by appending a D or d. Doing so is, of course, redundant. The default double type consumes 64 bits of storage, while the smaller float type requires only 32 bits.**

– **Hexadecimal floating-point literals are also supported, but they are rarely used. They must be in a form similar to scientific notation, but a P or p, rather than an E or e, is used. For example, 0x12.2P2 is a valid floating-point literal. The value following the P, called the binary exponent, indicates the power-of-two by which the number is multiplied. Therefore, 0x12.2P2 represents 72.5.**

© 2014, University of Colombo School of Computing

Data types and variables in Java Cont...

- Closer look at literals - Floating Point Literals cont...
  - Beginning with JDK 7, you can embed one or more underscores in a floating-point literal. This feature works the same as it does for integer literals, which were just described. Its purpose is to make it easier to read large floating-point literals. When the literal is compiled, the underscores are discarded.
  - For example, given double num = 9_423_497_862.0;the value given to num will be 9,423,497,862.0. The underscores will be ignored. As is the case with integer literals, underscores can only be used to separate digits. They cannot come at the beginning or the end of a literal. It is, however, permissible for more than one underscore to be used between two digits. It is also permissible to use underscores in the fractional portion of the number. For example, double num = 9_423_497.1_0_9;is legal. In this case, the fractional part is .109.

Data types and variables in Java Cont…

- Closer look at literals - Boolean Literals

  – **Boolean literals are simple. There are only two logical values that a boolean value can have, true and false. The values of true and false do not convert into any numerical representation.**

  – **The true literal in Java does not equal 1, nor does the false literal equal 0. In Java, the Boolean literals can only be assigned to variables declared as boolean or used in expressions with Boolean operators.**

© 2014, University of Colombo School of Computing

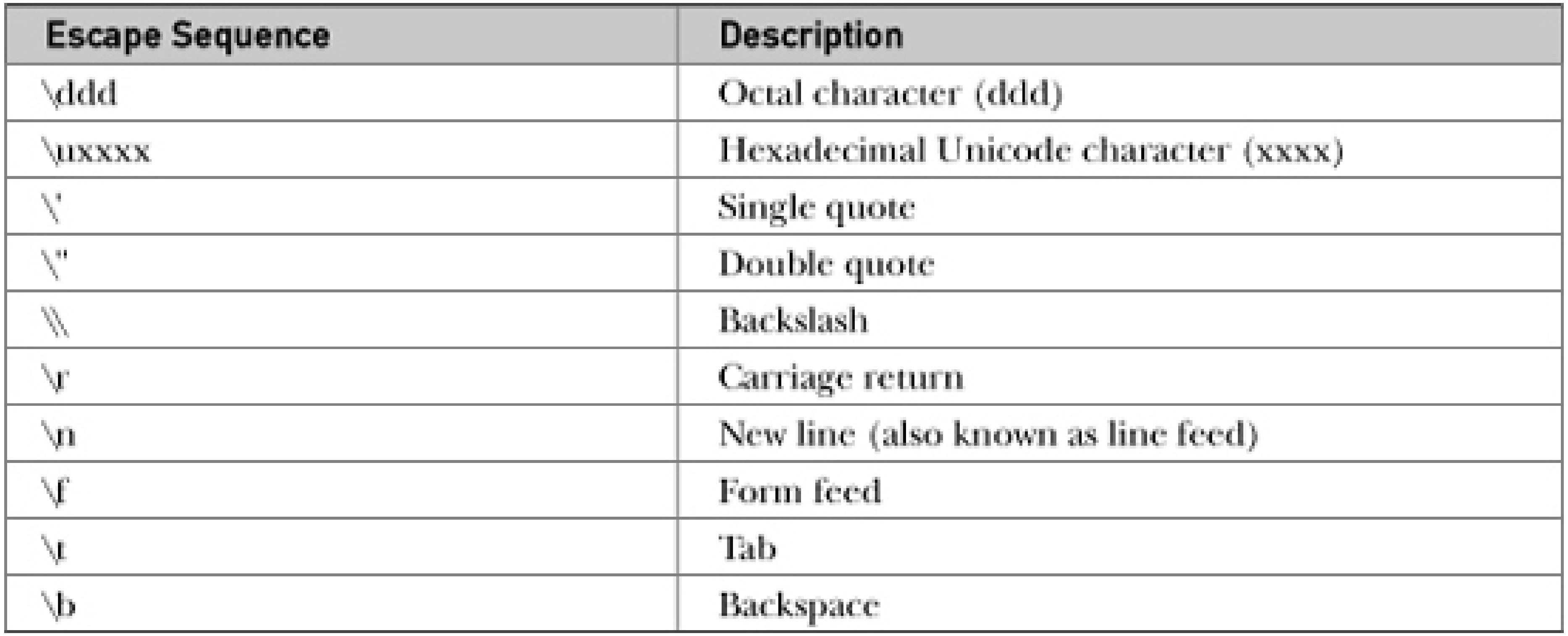
Data types and variables in Java Cont…

➢ Closer look at literals - Character Literals

– **Characters in Java are indices into the Unicode character set. They are 16-bit values that can be converted into integers and manipulated with the integer operators, such as the addition and subtraction operators. A literal character is represented inside a pair of single quotes. All of the visible ASCII characters can be directly entered inside the quotes, such as 'a', 'z', and '@'. For characters that are impossible to enter directly, there are several escape sequences that allow you to enter the character you need, such as '\" for the single-quote character itself and '\n' for the newline character. There is also a mechanism for directly entering the value of a character in octal or hexadecimal. For octal notation, use the backslash followed by the three-digit number. For example, '\141' is the letter 'a'. For hexadecimal, you enter a backslash-u (\u), then exactly four hexadecimal digits. For example, '\u0061' is the ISO-Latin-1 'a' because the top byte is zero. '\ua432' is a Japanese Katakana character.**

Data types and variables in Java Cont…

- Closer look at literals - Character Literals cont…
  - **Escape Sequences**



Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal Unicode character (xxxx)
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace

© 2014, University of Colombo School of Computing

Data types and variables in Java Cont…

- Closer look at literals - String Literals
  - String literals in Java are specified like they are in most other languages—by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are:

    “Hello World”

    “two\nlines”

    “\”This is in quotes\””

As you may know, in some other languages, including C/C++, strings are implemented as arrays of characters. However, this is not the case in Java. Strings are actually object types.

Data types and variables in Java Cont...

- Variables
  - The variable is the basic unit of storage in a Java program.
  - A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime.

Scope type identifier initializer

Data types and variables in Java Cont...

- Declaring Variables
  - 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] ...];

  - The type is one of Java's atomic types, or the name of a class or interface.
  - The identifier is the name of the variable.
  - You can initialize the variable by specifying an equal sign and a value.
  - Keep in mind that the initialization expression must result in a value of the same (or compatible) type as that specified for the variable. To declare more than one variable of the specified type, use a comma-separated list.

© 2014, University of Colombo School of Computing

Data types and variables in Java Cont…

- Declaring Variables cont…
  - Here are several examples of variable declarations of various types. Note that some include an initialization.

```
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'.
```

  - The identifiers that you choose have nothing intrinsic in their names that indicates their type. Java allows any properly formed identifier to have any declared type.

Data types and variables in Java Cont...

- Dynamic Initialization
  - Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.
  - For example, here is a short program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides:

```
// 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);
  }
}
```

Data types and variables in Java Cont…

- Dynamic Initialization cont…
  - In the program three variables—a, b, and c—are declared. The first two, a and b, are initialized by constants. However, c is initialized dynamically to the length of the hypotenuse (using the Pythagorean theorem). The program uses another of Java's built-in methods, sqrt( ), which is a member of the Math class, to compute the square root of its argument. The key point here is that the initialization expression may use any element valid at the time of the initialization, including calls to methods, other variables, or literals

Data types and variables in Java Cont…

- The Scope and Lifetime of Variables
  - So far, all of the variables used have been declared at the start of the main( ) method. However, Java allows variables to be declared within any block.
  - A block is begun with an opening curly brace and ended by a closing curly brace.
  - A block defines a scope. Thus, each time you start a new block, you are creating a new scope.
  - A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

Data types and variables in Java Cont…

- The Scope and Lifetime of Variables cont…
  - Many other computer languages define two general categories of scopes: global and local.
  - However, these traditional scopes do not fit well with Java's strict, object-oriented model.
  - While it is possible to create what amounts to being a global scope, it is by far the exception, not the rule.
  - In Java, the two major scopes are those defined by a class and those defined by a method. Even this distinction is somewhat artificial.
  - However, since the class scope has several unique properties and attributes that do not apply to the scope defined by a method, this distinction makes some sense.

Data types and variables in Java Cont…

- The Scope and Lifetime of Variables cont…
  - 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.
  - As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification. Indeed, the scope rules provide the foundation for encapsulation.
  - Scopes can be nested. 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.

Data types and variables in Java Cont…

- The Scope and Lifetime of Variables cont…
  - To understand the effect of nested scopes, consider the following program:

```
// 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);
  }
}
```

Data types and variables in Java Cont…

- The Scope and Lifetime of Variables cont…
  - As the comments indicate, the variable x is declared at the start of main( )'s scope and is accessible to all subsequent code within main( ). Within the if block, y is declared. Since a block defines a scope, y is only visible to other code within its block. This is why outside of its block, the line y = 100; is commented out. If you remove the leading comment symbol, a compile-time error will occur, because y is not visible outside of its block. Within the if block, x can be used because code within a block (that is, a nested scope) has access to variables declared by an enclosing scope.
  - Within a block, variables can be declared at any point, but are valid only after they are declared. Thus, if you define a variable at the start of a method, it is available to all of the code within that method. Conversely, if you declare a variable at the end of a block, it is effectively useless, because no code will have access to it. For example, this fragment is invalid because count cannot be used prior to its declaration:

```
// This fragment is wrong!
count = 100; // oops! cannot use count before it is declared!
int count;
```

Data types and variables in Java Cont...

- The Scope and Lifetime of Variables cont…
  - Here is another important point to remember: variables are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope.
  - 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

© 2014, University of Colombo School of Computing

Data types and variables in Java Cont…

- The Scope and Lifetime of Variables cont…

```
// 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); // this always prints -1
      y = 100;
      System.out.println("y is now: " + y);
    }
  }
}
```

Data types and variables in Java Cont…

- The Scope and Lifetime of Variables cont…
  - The output generated by this program is shown here:

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

Data types and variables in Java Cont…

- The Scope and Lifetime of Variables cont…
  - As you can see, y is reinitialized to −1 each time the inner for loop is entered. Even though it is subsequently assigned the value 100, this value is lost.
  - One last point: Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope. For example, the following program is illegal:

```
// This program will not compile
class ScopeErr {
  public static void main(String args[]) {
    int bar = 1;
    {              // creates a new scope
      int bar = 2;  // Compile-time error - bar already defined!
    }
  }
}
```

Data types and variables in Java Cont…

- Type Conversion and Casting
  - It is fairly common to assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically.
  - For example, it is always possible to assign an int value to a long variable.
  - However, not all types are compatible, and thus, not all type conversions are implicitly allowed.
  - For instance, there is no automatic conversion defined from double to byte. Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use a cast, which performs an explicit conversion between incompatible types. Let’s look at both automatic type conversions and casting.

© 2014, University of Colombo School of Computing

Data types and variables in Java Cont…

- Type Conversion and Casting cont…
  - 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. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required.
  - For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, there are no automatic conversions from the numeric types to char or boolean. Also, char and boolean are not compatible with each other.
  - As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long, or char.

© 2014, University of Colombo School of Computing

Data types and variables in Java Cont…

- Type Conversion and Casting cont…

  Casting incompatible types

  – Although the automatic type conversions are helpful, they will not fulfill all needs. For example, what if you want to assign an int value to a byte variable?

  – This conversion will not be performed automatically, because a byte is smaller than an int. This kind of conversion is sometimes called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type.

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

    (target-type) value

  – Here, target-type specifies the desired type to convert the specified value to. For example, the following fragment casts an int to a byte. If the integer's value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte's range.

```
int a;
byte b;
// ...
b = (byte) a;
```

Data types and variables in Java Cont…

- Type Conversion and Casting cont…

  A different type of conversion will occur when a floating-point value is assigned to an integer type: truncation. As you know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost. For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated. Of course, if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.

Data types and variables in Java Cont...

- Automatic Type Promotion in Expressions
- In addition to assignments, there is another place where certain type conversions may occur: in expressions. To see why, consider the following. In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand. For example, examine the following expression:

```
byte a = 40;
byte b = 50;
byte c = 100;
```

- int d = a * b / c;The result of the intermediate term a * b easily exceeds the range of either of its byte operands. To handle this kind of problem, Java automatically promotes each byte, short, or char operand to int when evaluating an expression. This means that the subexpression a*b is performed using integers—not bytes. Thus, 2,000, the result of the intermediate expression, 50 * 40, is legal even though a and b are both specified as type byte.

Data types and variables in Java Cont…

- Automatic Type Promotion in Expressions cont…
  - As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, this seemingly correct code causes a problem:

```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```

  - The code is attempting to store 50 * 2, a perfectly valid byte value, back into a byte variable. However, because the operands were automatically promoted to int when the expression was evaluated, the result has also been promoted to int. Thus, the result of the expression is now of type int, which cannot be assigned to a byte without the use of a cast. This is true even if, as in this particular case, the value being assigned would still fit in the target type.

Data types and variables in Java Cont…

- Automatic Type Promotion in Expressions cont…
  - In cases where you understand the consequences of overflow, you should use an explicit cast, such as

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

  - which yields the correct value of 100.

Data types and variables in Java Cont...

- The Type Promotion Rules
  - Java defines several type promotion rules that apply to expressions.
  - They are as follows: First, all byte, short, and char values are promoted to int, as just described. Then, 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 are double, the result is double.
  - The following program demonstrates how each value in the expression gets promoted to match the second argument to each binary operator:

Data types and variables in Java Cont...

➢ The Type Promotion Rules cont...

```
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);
  }
}
```

Data types and variables in Java Cont…

- The Type Promotion Rules cont…
  - Let’s look closely at the type promotions that occur in this line from the program:

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

  - In the first sub expression, f * b, b is promoted to a float and the result of the sub expression is float. Next, in the sub expression i/c, c is promoted to int, and the result is of type int. Then, in d * s, the value of s is promoted to double, and the type of the sub expression is double. Finally, these three intermediate values, float, int, and double, are considered. The outcome of float plus an int is a float. Then the resultant float minus the last double is promoted to double, which is the type for the final result of the expression.

© 2014, University of Colombo School of Computing

Section 3.2

Operators

© 2014, University of Colombo School of Computing

Operators

- An operator is a function that has a special symbolic name and is invoked by using that symbol with an expression.
- Java has several kinds of operators:
  - Arithmetic Operators
    - These are used to perform standard arithmetic operations performed on numbers
  - Assignment Operator
    - This is used to assigns a variable with a value
  - Logical (Boolean) Operators
    - These are used to perform standard logical operations on Boolean values
  - Bitwise Operators
    - These perform operations on the individual bits of integers
  - Conditional Operator
    - This acts as a shorthand for if-then-else

Operators
Arithmetic Operators

Symbol	Meaning	Example	
+	Addition	3+4	=7
-	Subtraction	15-7	=8
*	Multiplication	7*8	=56
/	Division	15/3	=5
%	Modulus	59%7	=3

© 2014, University of Colombo School of Computing

Operators
Assignment Operator

- Assignment Operator (=)
- For example:

```
int a;
a = 5;
```

- Right side of Assignment operator is evaluated first and then this value is assigned to the Left side of the Assignment operator

Operators
Logical (Boolean) Operators

Symbol	Meaning	Example	
&&	Logical AND	true && false	= false
||	Logical OR	true || false	= true
!	Logical NOT	!true	= false

Operators
Bitwise Operator

Symbol	Meaning	Example	
&	Bitwise AND	0 & 1	= 0
|	Bitwise OR	0 | 1	= 1
^	Bitwise XOR	^0	= 1
<<	Left Shift	1<<4	= 16
>>	Right Shift	256>>4	= 64
~	Bitwise Compliment		

© 2014, University of Colombo School of Computing

Operators
Assignment Shortcuts

- Often in programming, we have to apply some operator to some variable and then assign the resultant value back to the same variable. For example,

```
a = a + 5;
```

- Note, here we have two operations, one addition (**a + 5**) and one assignment (**a = ...**).
- Java provides shortcut operators that combine the two operations.

```
a += 5;
```

- Similarly we have, **-=**, ***=**, **/=**, **%=**, **&=**, **|=** etc.
- Very often we need to perform additions or subtractions by one (increments or decrements), that is expressions of the form **a = a + 1** or **a = a - 1**. These can be done by the Increment Operator and Decrement Operator **++** and **--**.

```
a++;
a--;
```

Operators
Relational Operator

Symbol	Meaning	Example	
==	Equal to	2==3	= false
!=	not Equal to	2!=3	= true
>	Greater than	2>3	= false
<	Less than	2<3	= true
>=	Greater than or Equal	2>=3	= false
<=	Less than or Equal	2<=3	= true

© 2014, University of Colombo School of Computing

Operators
Conditional Operator

- Conditional Operator (?:)
- This operator has the form:
  - **(condition)?(value if true):(value if false)**
  - The condition is evaluated and if it is true **value if true** is returned, otherwise **value if false** is returned.
- For example, if **a** and **b** were integers, the following would return the maximum of **a** and **b**.
  - **(a>b)?a:b**
- This is a short hand for “if-then-else”

Operators
Operator Precedence

- Consider the following expression
  - **6+2*3**
- This consists of two operations, one addition and one multiplication.
- The value of the expression depends on which of these operations is performed first.
  - If addition is performed first we get:
    6+2*3 = (6+2)*3 = 8*3 = 24
  - If multiplication is performed first we get:
    6+2*3 = 6+(2*3) = 6+6 = 12
- To avoid ambiguity and confusion, java defines a clear order in which operators are evaluated. This is known as **operator precedence.**
- According to operator precedence, the multiplication operator (*) has higher precedence than the addition operator (+). Hence, the correct evaluation of **6+2*3** is **12.**

Operators
Operator Precedence cont..

- The complete operator precedence is as follows.
- Operators at the top have higher precedence.
- Operators at the same level have equal precedence

Highest

. [] ( )	
++ -- ! ~	Increment Operators
new	
* / %	Arithmetic Operators
+ -	
<< >> >>>	Bitwise Shift Operators
< > <= >=	Comparison Operators
== !=	
&	Bitwise Operators
^	
|	
&&	Logical Operators
||	
?:	Conditional Operator
= += - = etc.	Assignment Operators
&= |= <<= etc.	

Lowest

Section 3.3

Flow Control

© 2014, University of Colombo School of Computing

Flow Control

- **Control flow** refers to the order in which statements are executed in an algorithm.
- Usually, an algorithm executes **sequentially**; that is the first statement executes, then the second and so on. However, it is often useful to be able to alter this flow.
- A **flow control statement** is a statement that changes the order of execution of subsequent statements.
- We will be talking about 3 types of flow control statements
  - Selection Statements
  - Iterative Statements
  - Jump Statements

Selection Statements

- A Selection Statement, evaluates some expression, and depending on the expression’s value, selects one of several possible sets of statements to execute.

- In Java, we have two types of selection statements:
  - **If**
  - **switch**

Selection Statements Cont…
If Statements

- An If Statement has the form:

```
if (<Boolean expression>)
{
    <statement1.1>;
    <statement1.2>;
    …
}
else
{
    <statement2.1>;
    <statement2.2>;
    …
}
```

- If **<Boolean expression>** is true, then **<statement1.1>, <statement1.2>** etc. are executed; else **<statement2.1>, <statement2.2>** etc. are executed. The "else" part is optional if there is nothing to be executed if **<Boolean expression>** is false.

© 2014, University of Colombo School of Computing

Selection Statements Cont...
If Statements cont...

- If only one statement needs to be executed, the curly brackets can be omitted.

```
if (<Boolean expression>)
   <statement1.1>;
else
   <statement2.1>;
```

Selection Statements Cont...
If Statements cont...

- If statements can be nested inside other if statements to give multiple if statements.

```
if (<Boolean expression1>)
   <statement1.1>;
else if (<Boolean expression2>)
   <statement2.1>;
else if (<Boolean expression3>)
   <statement3.1>;
else
   <statement4.1>;
```

Selection Statements Cont…
Switch Statements

- An Switch Statement has the form:

```
switch (<Expression>)
{
    case <value 1>:
            <statement1.1>;
            <statement1.2>;
    case <value 2>:
            <statement2.1>;
            <statement2.2>;
    case <value 3>:
            <statement3.1>;
            <statement3.2>;
    case <value 4>:
            <statement4.1>;
            <statement4.2>;
            …
    default:
            <statementD.1>;
            <statementD.2>;
}
```

- The expression **<Expression>** must evaluate to a byte, int or a char.

Selection Statements Cont…
Switch Statements cont…

- **<Expression>** is compared with each of the case values **<value 1>, <value 2>, <value 3>** etc.
- If a match is found, the statements that correspond to the first match, and all the other case statements further down are executed

  For example if **<Expression> is equal to <value 2>, <statement2.1>, <statement2.2>** etc. are executed; then, **<statement3.1>, <statement3.2>** etc. are executed, and so on.
- If a match is not found, default statements **<statementD.1>, <statementD.2>** etc. are executed.
- The default statements are optional. If it is not there, case statement completes without doing anything.

Selection Statements Cont...

```
// Demonstrate if-else-if statements.
class IfElse {
 public static void main(String args[]) {
  int month = 4; // April
  String season;

  if(month == 12 || month == 1 || month == 2)
   season = "Winter";
  else if(month == 3 || month == 4 || month == 5)
   season = "Spring";
  else if(month == 6 || month == 7 || month == 8)
   season = "Summer";
  else if(month == 9 || month == 10 || month == 11)
   season = "Autumn";
  else
   season = "Bogus Month";

  System.out.println("April is in the " + season + ".");
 }
}
```

Selection Statements Cont...

```
// A simple example of the switch.
class SampleSwitch {
  public static void main(String args[]) {
    for(int i=0; i<6; i++)
      switch(i) {
        case 0:
          System.out.println("i is zero.");
          break;
        case 1:
          System.out.println("i is one.");
          break;
        case 2:
          System.out.println("i is two.");
          break;
        case 3:
          System.out.println("i is three.");
          break;
        default:
          System.out.println("i is greater than 3.");
      }
  }
}
```

Selection Statements Cont…

// A simple example of the switch.

```
class SampleSwitch {
 public static void main(String args[]) {
  for(int i=0; i<6; i++)
   switch(i) {
    case 0:
     System.out.println("i is zero.");
     break;
    case 1:
     System.out.println("i is one.");
     break;
    case 2:
     System.out.println("i is two.");
     break;
    case 3:
     System.out.println("i is three.");
     break;
    default:
     System.out.println("i is greater than 3.");
   }
  }
}
```

Selection Statements Cont...

- The **break** statement is optional. If you omit the **break**, execution will continue on into the next **case**. It is sometimes desirable to have multiple **case**s without **break** statements between them. For example, consider the following program:

```
// In a switch, break statements are optional.
class MissingBreak {
 public static void main(String args[]) {
  for(int i=0; i<12; i++)
   switch(i) {
    case 0:
    case 1:
    case 2:
    case 3:
    case 4:
     System.out.println("i is less than 5");
     break;
    case 5:
    case 6:
    case 7:
    case 8:
    case 9:
     System.out.println("i is less than 10");
     break;
    default:
     System.out.println("i is 10 or more");
   }
 }
}
```

Selection Statements Cont…

JDK 7, you can use a string to control a **switch** statement. For example,// Use a string to control a switch statement.

```
class StringSwitch {
  public static void main(String args[]) {
    String str = "two";
    switch(str) {
      case "one":
        System.out.println("one");
        break;
      case "two":
        System.out.println("two");
        break;      case "three":
        System.out.println("three");       break;     default:
                System.out.println("no match");       break;   } }
```

Iterative Statements
For loops

- A For loop has the form:

```
for (<Initialization statement>; <Boolean Expression>; <Increment statement>)
{
    <statement1.1>;
    <statement1.2>;
    ...
}
```

- First **<Initialization statement>** is executed.
- Next **<Boolean Expression>**is evaluated.
  - If **<Boolean Expression>** is true, **<statement1.1>, <statement1.2>** etc. and **<Increment statement>** are executed. The **<Boolean Expression>**is evaluated again, and the loop continues.
  - If **<Boolean Expression>** is false, executing the iterative statement is concluded.

© 2014, University of Colombo School of Computing

Iterative Statements Cont...
For loops cont...

- For example, the following loop finds the sum of the numbers 0, 1, 2, ...99.

```
int sum = 0;
for (int i = 0; i<100; i=i+1)
{
   sum = sum +i;
}
```

Iterative Statements Cont...
While loops

- A While loop has the form:

```
while (<Boolean Expression>)
{
    <statement1.1>;
    <statement1.2>;
    ...
}
```

- First **<Boolean Expression>**is evaluated.
  - If **<Boolean Expression>** is true, **<statement1.1>, <statement1.2>** etc. are executed. The **<Boolean Expression>**is evaluated again, and the loop continues.
  - If **<Boolean Expression>** is false, executing the iterative statement is concluded.

© 2014, University of Colombo School of Computing

Iterative Statements Cont... While loops cont...

- For example, the following loop finds the sum of the numbers 0, 1, 2, ...99.

```
int sum = 0;
int i=0;
while (i<100)
{
   sum = sum +i;
   i = i+1;
}
```

Iterative Statements Cont...
Do While loops

- A Do While loop has the form:

```
do
{
    <statement1.1>;
    <statement1.2>;
    ...
}
while (<Boolean Expression>);
```

- First **<statement1.1>, <statement1.2>** etc. are executed.

- Next, the **<Boolean Expression>**is evaluated.

  – If **<Boolean Expression>** is true, **<statement1.1>, <statement1.2>** etc. are executed. The **<Boolean Expression>**is evaluated again, and the loop continues.

  – If **<Boolean Expression>** is false, executing the iterative statement is concluded.

© 2014, University of Colombo School of Computing

Iterative Statements Cont...
Do While loops cont...

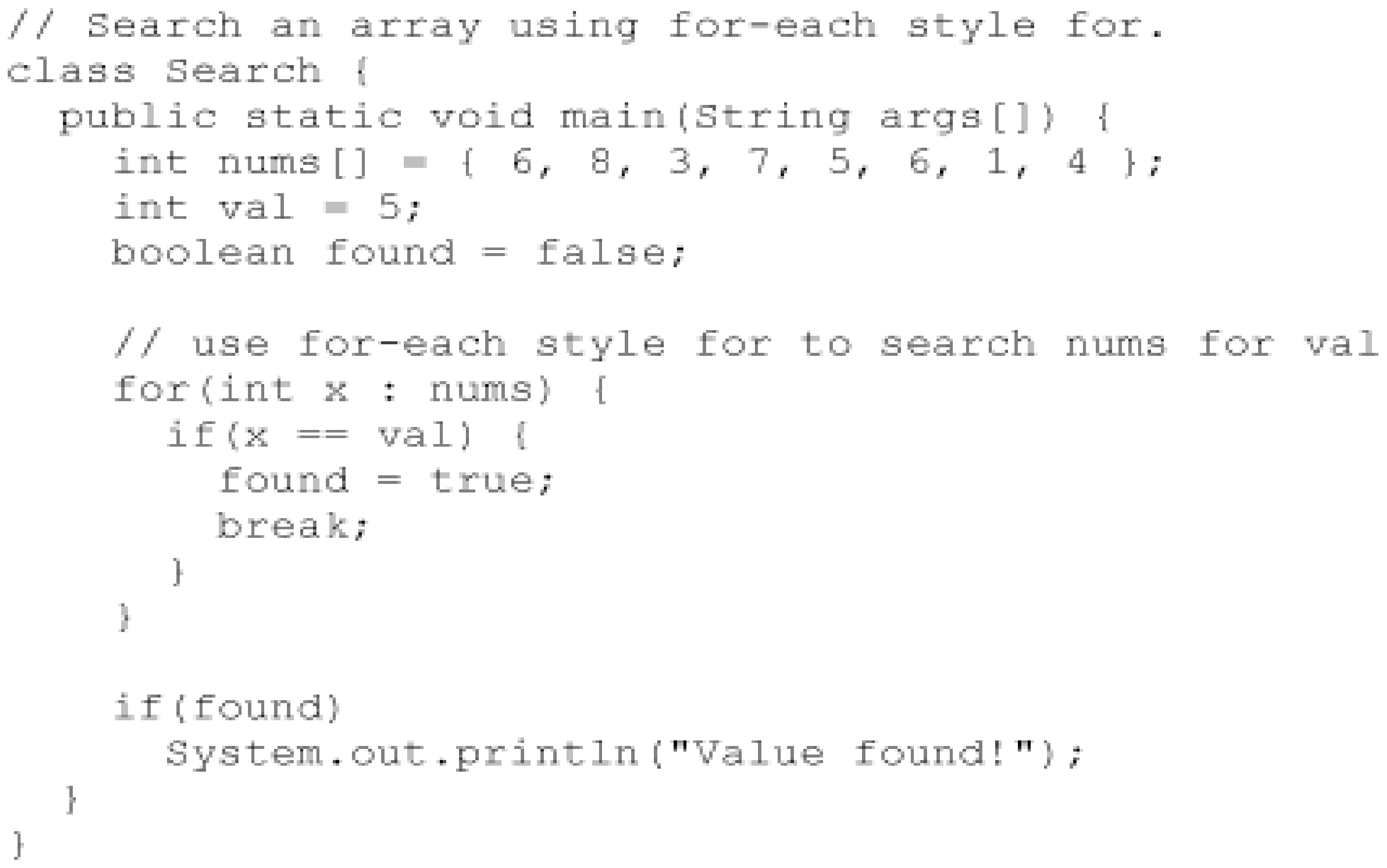
- For example, the following loop finds the sum of the numbers 0, 1, 2, ...99.

```
int sum = 0;
int i=0;
do
{
   sum = sum +i;
   i = i+1;
}
while (i<100);
```

Applying the Enhanced for

- Since the for-each style **for** can only cycle through an array sequentially, from start to finish, you might think that its use is limited, but this is not true. A large number of algorithms require exactly this mechanism. One of the most common is searching. For example, the following program uses a **for** loop to search an unsorted array for a value. It stops if the value is found.

Applying the Enhanced for Cont...



```
// Search an array using for-each style for.
class Search {
  public static void main(String args[]) {
    int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };
    int val = 5;
    boolean found = false;

    // use for-each style for to search nums for val
    for(int x : nums) {
      if(x == val) {
        found = true;
        break;
      }
    }

    if(found)
      System.out.println("Value found!");
  }
}
```

Jump Statements

- Jump statements move the point of execution in a program from one place to another place.

- In java we have 3 types of Jump Statements.
  - Break Statement
  - Continue Statement
  - Return Statement

Jump Statements Cont... Break Statement

- A Break Statement jumps out of a loop and effectively bypasses the loop condition.
- For example, the following loop finds the sum of the numbers 0, 1, 2, ...55.

```
int sum = 0;
int i=0;
while (i<100)
{
    sum = sum +i;
    i = i+1;
    if (i==55)
        break;
}
```

© 2014, University of Colombo School of Computing

Jump Statements Cont...
Continue Statement

- A Continue Statement jumps out of the current iteration of a loop.
- For example, the following loop finds the sum of the numbers 0, 1, 2, ...99 excluding the numbers that are multiples of 5.

```
int sum = 0;
for (int i = 0; i<100; i=i+1)
{
    if (i%5==0)
        continue;
    sum = sum +i;
}
```

Jump Statements Cont… Return Statement

- A Return Statement terminates the current method (function) and jumps to the place immediately after the function call.

© 2014, University of Colombo School of Computing

Jump Statements Cont…
Labeled loops

- With a label, program controls transfer to the statement just after the enclosing statement or block of statements carrying the same label.

```
for (int i=0; i<3; i++)
 {
   resume:
   for (...............)
    {
       ........
       if (........) break resume;
    }
 }
```

Section 3.4

Arrays

© 2014, University of Colombo School of Computing

Arrays

- Arrays provide a way of storing a list of variables of the same data type, one after the other

- Arrays must be declared. For example, an array of 10 integers might be declared as:
    - **int[] a = new int[10];**

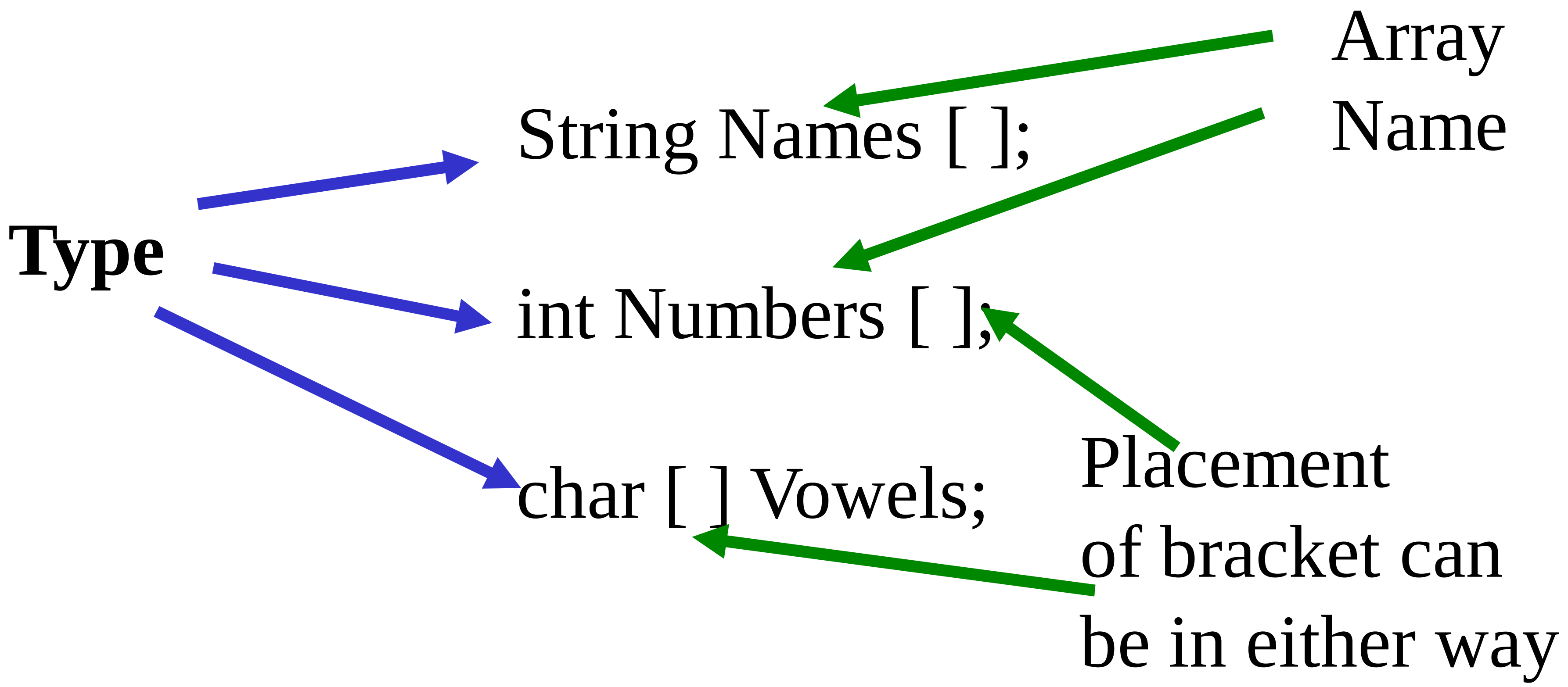
- Elements of an array can be accessed by indicating the index (position) of the element in the array inside square brackets. Indexing (Position numbering) begins with 0.

- For example, to access the first element in an array, **a[0]**, the second element **a[1]**, and so on.

Arrays Cont...

Declaring array Variables

– Arrays are declared using enclosing square brackets.



```
String Names [ ];

int Numbers [ ];

char [ ] Vowels;
```

Arrays Cont...

Declaring array Variables cont...

– By placing the bracket before the Array Name

  • we can declare multiple arrays of same type in the same line.

```
int [ ] firstArray, secondArray ;
```

Both are arrays of
Type int

Arrays Cont...

Declaring array Variables cont …

- By placing the bracket after the Array Name
  - we can declare variables and arrays of same Type in the same line.

```
int  firstArray[ ], justAVariable ;
```

Array of Type int   Variable of Type int

Arrays Cont...

- if the return type is an array object,
  - the square brackets can go after the return type or after the parameter list

int [ ] **SortedList** (int List [ ]);

OR

int **SortedList** (int List [ ]) [];

Arrays Cont...

Creating Array Objects

– Array Objects can be created using two methods.
  - Using the **new** operator
  - Directly Initializing the contents

Number of elements

String firstName [ ] = **new** String [5];

String firstName [ ] = { “Kamal”, “Amal” , “Nimal
“Saman”, “Sunil” } ;

© 2014, University of Colombo School of Computing

Arrays Cont...

Accessing Array elements

– After initializing, Array elements can be accessed using subscript expression ([ ]) .

Subscript starts with 0 ends with 4

firstName [subscript];

Array with 5 elements

Example:
firstName[0] will give “Kamal”
firstName[1] will give “Amal”
firstName[2] will give “Nimal”

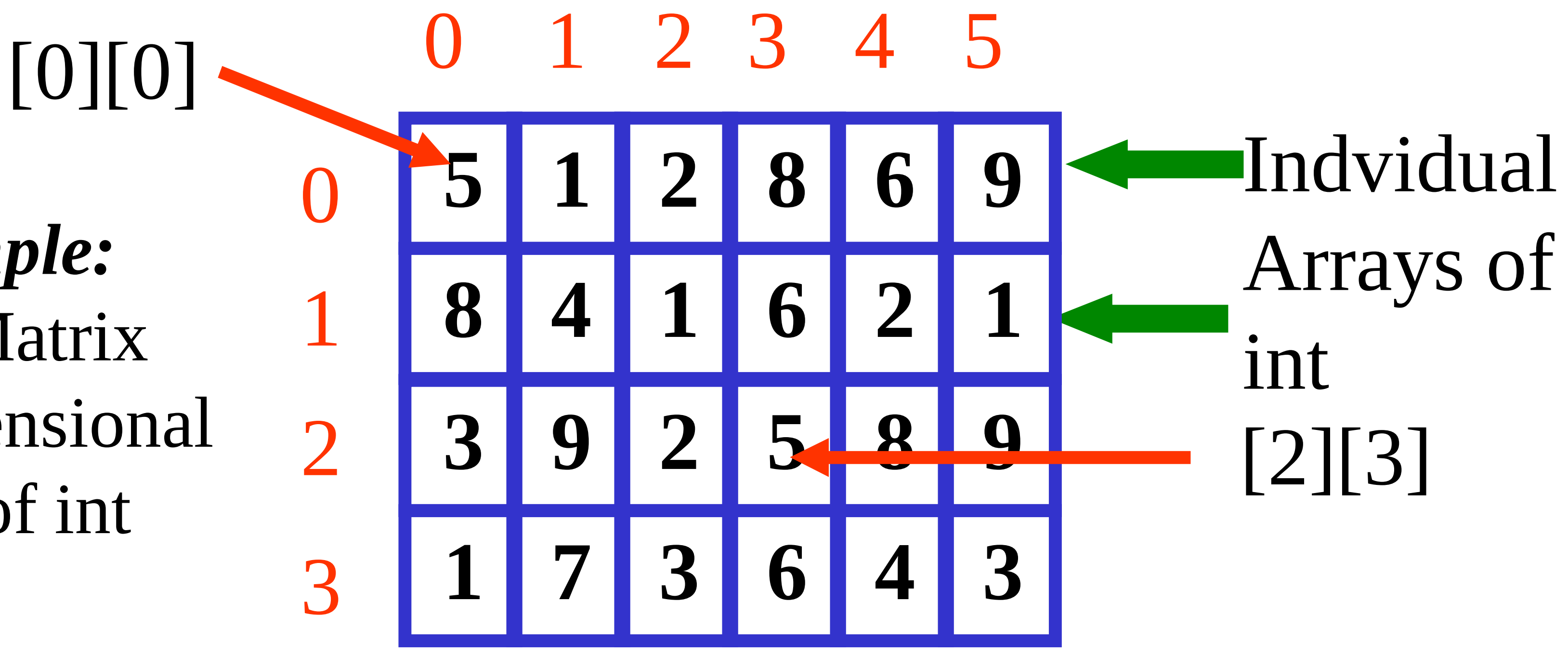
© 2014, University of Colombo School of Computing

Arrays Cont...

Multi Dimensional Arrays

– In Java Multi Dimensional Arrays are just Arrays of Arrays.

Exapmple:
twoDMatrix
2 Dimensional
Array of int



$$
\begin{array}{c|cccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 5 & 1 & 2 & 8 & 6 & 9 \\
1 & 8 & 4 & 1 & 6 & 2 & 1 \\
2 & 3 & 9 & 2 & 5 & 8 & 9 \\
3 & 1 & 7 & 3 & 6 & 4 & 3
\end{array}
$$

Arrays Cont...

Multi Dimensional Arrays cont…

– Can be created in either way as follows

```
int twoDMatrix [ ][ ] = new int [4][6];

int twoDMatrix [ ][ ] ={ {5,1,2,8,6,9},
                         {8,4,1,6,2,1},
                         {3,9,2,5,8,9},
                         {1,7,3,6,4,3} };
```

Note:
twoDMatrix.length=4
(Number of rows)
twoDMatrix[0].length=6
(lenth of first row Array)
twoDMatrix[1].length=6

Arrays Cont...

Multi Dimensional Arrays Cont..

– Since Java Multi Dimensional Arrays are Arrays of Arrays,
  - It is possible to create ***Ragged Arrays***

```
int twoDMatrix [ ][ ] ={ {5,1,2,8,6,9}
                         {8,4,1,1}
                         {3,9,2,5,8}
                         {1,7,3,6,4,3} };
```

Variable, row array lenghths
twoDMatrix[1].length=4
(lenth of second row Array)
twoDMatrix[2].length=5
(lenth of third row Array)

Arrays Cont...

Multi Dimensional Arrays Cont …

– Alternatively
  - It is possible to declare the same ***Ragged Array*** as follows:

```
int twoDMatrix [ ][ ] = new  int [4][ ];
twoDMatrix [0] = new  int [6];
twoDMatrix [1] = new  int [4];
twoDMatrix [2] = new  int [5];
twoDMatrix [3] = new  int [6];
```

Size of second Dimension is not Specified

Arrays Cont...

Multi Dimensional Arrays Cont…

- Each element can be accessed using two subscripts

```
twoDMatrix [0][0];
twoDMatrix [0][1];
```

- Each element can be changed as follows

```
twoDMatrix [0][0] = 6;
twoDMatrix [0][1] = 8;
```

© 2014, University of Colombo School of Computing

Arrays Cont...

Accessing Array elements (continued)

– Java **Run Time** will check to verify that the Array bounds are not exceeded

– Each array object has a property called **length** which will **yield the size** of Array

Example

firstName[5] will throw an **Exception**

firstName.length will yield 5

maximum subscript is always **firstName.length -1**

Arrays Cont...

Changing Array elements

- To change an Array Element,
  - just use an assignment statement after the Array Access Expression

Example

```
        firstName[3] = “Kamala”;
```

now the element 3 will contain value “Kamala”

Arrays Cont...

Changing Array elements

- To change an Array Element,
  - just use an assignment statement after the Array Access Expression

Example

```
firstName[3] = “Kamala”;
```

now the element 3 will contain value “Kamala”

Command-Line Arguments

- A Java application can accept any number of arguments from the command line. This allows the user to specify configuration information when the application is launched.
- The user enters command-line arguments when invoking the application and specifies them after the name of the class to be run. For example, suppose a Java application called Sort sorts lines in a file. To sort the data in a file named friends.txt, a user would enter:

```
java Sort friends.txt
```

- When an application is launched, the runtime system passes the command-line arguments to the application's main method via an array of Strings. In the previous example, the command-line arguments passed to the Sort application in an array that contains a single String: "friends.txt".

Command-Line Arguments Cont...

- The Echo example displays each of its command-line arguments on a line by itself:

```
public class Echo {
    public static void main (String[] args) {
        for (String s: args) {
            System.out.println(s);
        }
    }
}
```

Command-Line Arguments Cont...

➢ Parsing Numeric Command-Line Arguments

- If an application needs to support a numeric command-line argument, it must convert a String argument that represents a number, such as "34", to a numeric value. Here is a code snippet that converts a command-line argument to an int:

```
int firstArg;
if (args.length > 0) {
try {
        firstArg = Integer.parseInt(args[0]);
    } catch (NumberFormatException e) {
        System.err.println("Argument" + args[0] + " must be an integer.");
        System.exit(1);
    }
}
```

© 2014, University of Colombo School of Computing

The End of Section 03