3.1. Data Types and Variables in Java

IT1406 - Introduction to Programming

Level I - Semester 1

3.1. 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.
- 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.

3.1. Data types and variables in Java

3.1.1.2. 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.

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.

3.1. Data types and variables in Java

3.1.1.2. The primitive types

- 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 pl
- atform. 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.

3.1. Data types and variables in Java

3.1.1.2. The primitive types

- 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.

3.1. Data types and variables in Java

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
- h 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.

3.1. Data types and variables in Java

3.1.1.2. Primitive types

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

3.1. Data types and variables in Java

3.1.1.2. Primitive types

- 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;

3.1. Data types and variables in Java

3.1.1.2. Primitive types

- 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)

3.1. Data types and variables in Java

3.1.1.2. Primitive types

- 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.

3.1. Data types and variables in Java

3.1.1.2. Primitive types

- 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.

3.1. Data types and variables in Java

3.1.1.2. Primitive types

- 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.

3.1. Data types and variables in Java

3.1.1.2. Primitive types

- 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.

3.1. Data types and variables in Java

3.1.1.2. Primitive types

- 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 transcendental 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.

3.1. Data types and variables in Java

3.1.1.2. Primitive types

- 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

3.1. Data types and variables in Java

3.1.1.2. Primitive types

- 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;

3.1. Data types and variables in Java

3.1.1.2. Primitive types

- 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;

3.1. Data types and variables in Java

3.1.1.2. Primitive types

- 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.
- 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.

3.1. Data types and variables in Java

3.1.1.2. Primitive types

- 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.
- 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';

3.1. Data types and variables in Java

3.1.1.2. Primitive types

- 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

3.1. Variables and Data Types
Primitive Data Types

3.1.1.2. Primitive types

- 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.

3.1. Variables and Data Types
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 and boolean to store true and false and char to store a single character

type	bits
byte	8
short	16
int	32
long	64

type	bits
float	32
double	64

type	bits
boolean	16
char	16

3.1. Variables and Data Types
Primitive Data Types

- 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**

3.1. Variables and Data Types

3.1.1.2. Primitive types

- Closer look at literals 3.1.1.3 - 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.

3.1. Variables and Data Types

- Closer look at literals 3.1.1.3 - Integer Literals(contd)
- 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, 0x7fffffffffffffL or 9223372036854775807L is the largest long. An integer can also be assigned to a char as long as it is within range.

3.1. Variables and Data Types

- Closer look at literals 3.1.1.3 - Integer Literals(contd)
- 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:

3.1. Variables and Data Types

- Closer look at literals 3.1.1.3 - Integer Literals(contd)

- 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;
- 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;

3.1. Variables and Data Types

- Closer look at literals 3.1.1.3 - 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.

3.1. Variables and Data Types

- Closer look at literals 3.1.1.3 - Floating Point Literals(cntd)
- 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.

3.1. Variables and Data Types

- Closer look at literals 3.1.1.3 - Floating Point Literals(cntd)
- 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.

3.1. Variables and Data Types

- Closer look at literals 3.1.1.3 - 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.

3.1. Variables and Data Types

3.1.1.3. 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.

3.1. Variables and Data Types

3.1.1.3. Character Literals

- 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

3.1. Variables and Data Types

3.1.1.3. 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.

3.1. Variables and Data Types

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

private int name = “Thisara Nuwantha”

3.1. Variables and Data Types

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.

3.1. Variables and Data Types

Declaring Variables

- 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.

3.1. Variables and Data Types

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:

3.1. Variables and Data Types

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);
•   }
• }
```

3.1. Variables and Data Types

Dynamic Initialization

- 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

3.1. Variables and Data Types

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.

3.1. Variables and Data Types

The Scope and Lifetime of Variables

- 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.

3.1. Variables and Data Types

The Scope and Lifetime of Variables

- 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.

3.1. Variables and Data Types

The Scope and Lifetime of Variables

- 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);
  }
}
```

3.1. Variables and Data Types

The Scope and Lifetime of Variables

- 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;

3.1. Variables and Data Types

The Scope and Lifetime of Variables

- 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

3.1. Variables and Data Types

The Scope and Lifetime of Variables

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

3.1. Variables and Data Types

The Scope and Lifetime of Variables

- 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
```

3.1. Variables and Data Types

The Scope and Lifetime of Variables

- 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!
    }
  }
}
```

3.1. Variables and Data Types

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.

3.1. Variables and Data Types

Type Conversion and Casting

- 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.

3.1. Variables and Data Types

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;
```

3.1. Variables and Data Types

- 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.

3.1. Variables and Data Types

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.

3.1. Variables and Data Types

Automatic Type Promotion in Expressions

- 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.

3.1. Variables and Data Types

Automatic Type Promotion in Expressions

- 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.

3.1. Variables and Data Types

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:

3.1. Variables and Data Types

The Type Promotion Rules

```
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);
  }
}
```

3.1. Variables and Data Types

The Type Promotion Rules

- 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.