Data Types, Variables

Java Is a Strongly Typed Language

- First, every variable 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.
- 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.

The Primitive Types

Integers

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

Floating-Point Types

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

Characters

- 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.
- In Java char is a 16-bit type
- The range of a char is 0 to 65,536
- The standard set of characters known as ASCII still ranges from 0 to 127 as always

```java
// Demonstrate char data type. class CharDemo { public static void main(String args[]) { char ch1, ch2; ch1 = 88; // code for X ch2 = 'Y'; System.out.print("ch1 and ch2: "); System.out.println(ch1 + " " + ch2); } }
```

Java

```
ch1 and ch2: X Y
```

Plain Text

- Notice that ch1 is assigned the value 88, which is the ASCII (and Unicode) value that corresponds to the letter X. As mentioned, the ASCII character set occupies the first 127 values in the Unicode character set. For this reason, all the “old tricks” that you may have used with characters in other languages will work in Java, too.

Booleans

- 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.

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. These elements are examined next.

Declaring a variable

```java
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'.
```

Java

Dynamic Initialization

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

Java

- Here, three local 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.

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.

- As explained earlier, 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.
- In Java, the two major scopes are those defined by a class and those defined by a method.
- Class scope to be discussed in later chapters
- **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.
- 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.

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

Java

- 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:

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

Java

- **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 is entered. For example, consider the next program:

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

Java

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

Java

- 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:**

```java
// 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! } } }
```

Java

Type Conversion and Casting

- If you have previous programming experience, then you already know that 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**.
- 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.

Java’s Automatic Conversions

- When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:
  - The two types are compatible
  - The destination type is larger than the source type
- When these two conditions are met, a widening conversion takes place. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required

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?
- 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
```

Java

- 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;
```

Java

- A different type of conversion will occur when a floating-point value is assigned to an integer type: **truncation**
- 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.

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:

```java
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.
- Thus, 2,000, the result of the intermediate expression, 50 * 40, is legal even though a and b are both specified as type byte.
- As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, this seemingly correct code causes a problem:

```java
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.
- In cases where you understand the consequences of overflow, you should use an explicit cast, such as below which yields the correct value of 100.

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