COMP6700/2140 Data and Types

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

February 2017

Topics

1. Types and literals
2. Primitive types
3. Objects and references
4. Arrays
5. Strings

Types and Literals — 1

Computer programs operate on values which are sequences of bits. These values can be represented as symbols (variables or constants), or as literals. Symbols must be declared with a type. The type determines the number of bits representing a given symbol in memory and how these bits are interpreted (its value). Java has *primitive types* and *references*.

Each type has literals (set of constant values of this type).

Java is a *strongly typed language* — every symbol must have a declared type.

The eight primitive types:

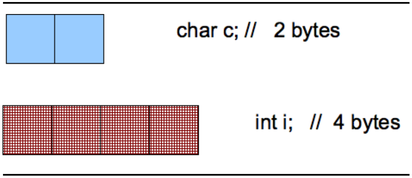
- `boolean` (1-bit, `true/false`)
- `byte` (8-bit signed, range `-128 ... +127`)
- `char` (16-bit unicode, character symbols from *Unicode Table*)
- `short` (16-bit signed, range `-32768 ... +32767`)
- `int` (32-bit signed, range `-2,147,483,648 ... +2,147,483,647`)
- `long` (64-bit signed range `-9,223,372,036,854,775,808 ... +9,223,372,036,854,775,807`)
- `float` (32-bit floating-point number)
- `double` (64-bit floating-point number)

Range of Primitive Values

Each primitive type can hold values within a certain range.

byte, *short*, *integer* and *long* are **signed** (their range includes negative values from -$2^{\text{bitsize}-1}$to +$2^{\text{bitsize}-1}$-1 — one bit is used for the sign, +/-). For instance, the `int` values change from -2,147,483,648 to +2,147,483,647.

Floating point types (`float` and `double`) include subsets of approximations to *real numbers* from very small to very large (discussed in greater detail later).



- *bit* — a single **bi**nary digi**t** with one of two possible values, `0` and `1`
- *byte* — the unit of data consisting of eight bits
- *word* — the unit of data used by a particular processor design (handled as a unit in the processor instruction set); modern general purpose computers have either 32- or 64-bit words

Types and Literals — 2

```
public class PrintNumber { // usage of the primitive type variables
  public static void main( String[] args ) {
      int i = 90000; short s = 20000; // one can have 2 statements on one line
      byte b = -100; double d = 1.5; // but normally it's not good for readability
      long l = 4000000000L; // note the `L` here
      float f = 1.5f; // note the f here
      char c = 'a'; boolean bn = true;
      int i2; i2 = -90000; // delayed initialisation
      System.out.println( "The value of s is " + s ); // display the value
   }
}
```

- Java variables are statically typed: *first* declare a variable and *then* assign it a value. Declaration and assignment can done in the same place, or assignment can be postponed.
- L or l suffix for long integers, and F or f suffix for floats. To avoid confusing the long type flag “l” and the literal value 1, use only the upper-case “L”.
- Range of values is independent of platform, unlike, *eg*, for C/C++ .
- Integer constants (literals of byte, short, integer and long) are strings of either decimal, octal (leading zero, 010) or hexadecimal (0xCAFE) digits. Octal format isn't recommended (confusing syntax!): (decimal)13 = (hexadecimal) 0xD = (binary) 1101. The 2-byte hex literal 0xCAFE if treated as a 32-bit int is 0000 0000 0000 0000 1100 1010 1111 1110 (binary) = 51966 (decimal), but if treated as a 16-bit short is 1100 1010 1111 1110 (binary) = -13570 (decimal).

Floating Point Numbers

In scientific notation, we commonly represent numbers in terms of a significand and an exponent: $x = \alpha \cdot 10^{\beta}$, for example,

$$\textit{mass(Earth)} = 5.97 \cdot 10^{24} \ \textit{kg}.$$

In a similar way, real numbers can be represented on a computer in terms of a binary significand and an exponent of base 2: $x = \alpha \cdot 2^{\beta}$.

The decimal point 'floats' left or right so that there is only one digit in front of the point. For all non-zero numbers, the first digit is a one. The exponent determines the magnitude of the number.

$$\textit{(binary)}\ 1101 = 1.101 \cdot 10^{11} \leftrightarrow \textit{(decimal)}\ 13 = 1.625 \cdot 2^{3}$$

Floating point numbers of a fixed size (float = 32 bits; double = 64 bits) cannot represent all real numbers within their range. Some numbers have an exact representation, but most numbers must be approximated.

Floating Point Numbers (2)

- For FP literals either decimal or hexadecimal format is used:
  - `18.=1.8e1=.18E2=180.0e-1` — decimal point is optional; `e` or `E` is optional, and if used is followed by optional signed integer
  - `0x12p0=0x1.2p4=0x.12P+8=0x120p-4` — the binary exponent, `p` or `P`, is mandatory; it's followed by an optionally signed integer; the exponent represents scaling by two raised to a power (the shown number is decimal `18.0`)
- `float` has ~7 decimal digits of precision, the trailing suffix "f" is mandatory. `double` has ~16 decimals of precision, the trailing suffix "d" is optional. Normally, use double unless you have a good reason.
- FP literals include *two zeros*, `1d * 0 = 0.0` (positive) and `-1d * 0 = -0.0` (negative); `0.0==-0.0` evaluates to `true`, but some expressions evaluate to different values: `1d/0.0=`∞, yet `1d/(-0.0)=`$-\infty$
- 64-bit IEEE 754 standard stipulates the number of bits used for significand and exponent: 52 and 11 (plus one bit for the sign +/-). There are special floating point values:
  1. NEGATIVE_INFINITY, POSITIVE_INFINITY, MAX_VALUE, MIN_VALUE (to denote overflows and underflows)
  2. `NaN`, "Not a Number" (examples: $0/0$, $\sqrt{-1}$), *Float* and *Double* classes provide a method `isNaN()` to test a FP value. `NaN` is the only value, for which `x == y` evaluates to `false` if both double's `x` and `y` are equal to `NaN`.

Types and Literals — 3

Numeric literals of “difficult” integer types can be (since Java SE 7) created using notations which are more readable for humans:

- Numeric constants (that is, one of the integer primitive types) may now be expressed as binary literals

  ```
  int x = 0b1100110;
  ```

- Underscores may be used in integer constants to improve readability:

  ```
  long anotherLong = 2_147_483_648L;
  int bitPattern = 0b0001_1100_0011_0111_0010_1011_1010_0011;
  ```

Type Compatibility

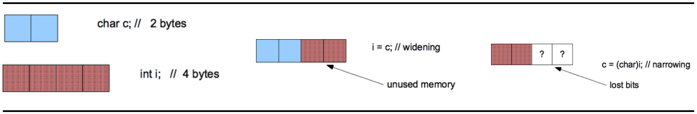
A variable of one type may be assigned a value of a different type:

long m; int n = 2; m = n;

This is a *type conversion*: `int`→ `long`.

Java is a *strongly typed language* — it checks for type compatibility at compile time in almost all cases — all “fishy” assignments are forbidden. To allow assignment of types with compatibility only known *at run time*, or to explicitly force conversion for primitive types that would otherwise lose range (*double* → *float*), Java provides the *cast* operator: `var1 = (type1)var2;`

- A “narrower” type, represented by a fewer number of bits, is *implicitly widened*, cast is not required
- A “wider” type must be *explicitly narrowed* to control the change of value: (“high” bits are chopped off).

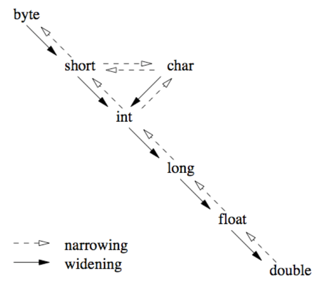


- The explicit cast is also needed if two types use an equal number of bits, but their memory layout is different, *e.g.* conversion `double` → `long` requires a translation from a 64-bit

Type Conversions and Casts — 1

Hierarchy of widening and narrowing conversions

- *a* → *b* means that a variable of the type *b* is assigned a value held by a variable of the type *a*;
- *widening* is an implicit type conversion, cast isn't needed since the doesn't change;
- *narrowing* needs the explicit cast (because the value *can* change of assignment).



Type Conversions and Casts — 2

Casts (including the effect of losing the fractional parts of floats) are illustrated by the following code:

```
public class ConversionTest {
  public static void main(String[] args) {
    int i1 = 10;
    float f1 = i1;
    double d1 = i1;
    double d2 = 1.5;
    float f2 = (float) d2;
    int i2 = (int) d2;
    // call print method to display the values
  }
}
```

Add printing statements for all the variables involved, compile and run this code to see the values generated by conversion.

What do you get when you print the value of i2?

Conversion, Rounding and Finite Precision

What is the value of `n` below?

-
  ```
  double f = 4.35;
  int n = (int) ( 100 * f );
  ```

Conversion, Rounding and Finite Precision

What is the value of `n` below?

-
  ```
  double f = 4.35;
  int n = (int) ( 100 * f );
  ```

-
  ```
  System.out.println(n); // prints 434: Wrong!
  ```

  What's wrong? In the binary number system, there is no exact representation for the number 4.35, it is actually stored as `4.349999999...` so `100 * f = 434.99999...` (etc). When cast to an integer, the fractional component is dropped (not rounded!), thus `n = 434`. The proper way to do rounding is to use a function call from the `Math` class:

  ```
  double f = 4.35; int n = (int)Math.round( 100 * f );
  System.out.println( n ); // prints 435! (Good)
  ```

  Rounding errors are a fact of life with floating point numbers.

Characters

- **Characters** are represented by primitive type `char`.
- Strings are not characters — strings are instances of the class *String*. But it may help to think of them as sequences of characters (which is what they are in term of values).
- *Character literals* appear between *single* quotes: `'a'`, string literals appear between *double* quotes: `"a"` (`'a'` and `"a"` are literal values of different type!).
- The character literals can be expressed either by a single symbol, or by an escape sequence inside the pair of single quotes, including an *octal character constant* (`\ddd`, where each `d` is one of `0-7`, but the code cannot exceed `\377`).

Unicode

- `char` literals are what you can find in the Unicode character table (used to be — and still is for C — the plain old ASCII with 128 entries; not for Java!): characters (Latin and other scripts), digits and many special symbols.
- Any valid Unicode character can be represented using ASCII symbols only with an *escape sequence* `\uxxxx`, where *x* is a hexadecimal digit (0,...,F). The number of `u` symbols can be more than one. (BTW, the Unicode character mapping escape sequence can be used anywhere inside Java code as a part of an identifier, not only in `char` and `String` literals). Here is a portability problem for Java: despite you can manipulate all Unicode characters in your code, you may not be able display them because many terminal and other output devices are yet to adopt the Unicode standard.
- Some special ASCII characters:
  - `\n = \u000A` (new line)
  - `\t = \u0009` (tab)
  - `\\ = \u005C` (backslash)

Try printing \a (bell), what happens? (*Hint* there is no literal constant for it, one has to use its unicode value, `\u0007`).

Arithmetic on Chars

Two `chars` can be added together — careful!

- first they are converted to `ints` (ie, the characters are replaced by their *code values*)
- then the sum is calculated and the result value is also `int`
- to get the character back, this sum should be cast back to `char`: `char c = (char)(a + b)`.

This also works for other arithmetic operations — try it!

Study these programs:

- CharTest.java
- CharTest1.java
- CharTest2.java

Non-Primitive Data Types

As well as primitive types, in an object-oriented language like Java, there are… objects.

- objects are composed of (potentially) several items of data called *fields*, which can be primitive values or references to other objects
- objects can also have operations associated with them called *methods*
- objects can be introduced into a program as literal values (e.g. literal strings like `"Java has become trendy again"`), but more commonly they are represented by references which are declared to have a particular type

The objects (of any type/class) are collectively referred to as *reference type* data.

The literal `null` is a special reference literal; it represents an invalid or uninitialized object. `null` does not have a type of its own, but it can be cast to any reference type.

Arrays

An array is special reference type, representing an ordered collection of data elements indexed by an integer from 0 (first element) up to `length - 1`. All elements have the *same* type.

The declaration and instantiations can be done differently:

```
//standard instantiation with elements set to default values
int[] ia = new int[3];
// creation of an array with initialisation of its element values (no new!)
int[] ia = { 2, 3, 5, 7, 11, 13 };
 // next C-like syntax is also legal, but don't do it!
int ia[] = { 2, 3, 5, 7, 11, 13 };
// as above, but note absence of the array size! (array size is inferred)
int[] ia =  new int[] { 17, 19, 23, 29, 31, }; // last comma is optional
```

Dimension of the array variables in the type declaration is omitted. The symbol `new` is the creation operator (used when an object is created). When it's called, an array is instantiated (*ie*, an array *object* is created) and its dimension is set. As every reference, `ia` can be later assigned to a different array of a different size (example ArrayTest.java).

- `arr[i]` returns the *i*-th element in the array `arr`, and `str.charAt(i)` returns the character at the *i*-th position.
- An arrays has the `length` *field* which tells you how long it is (ie, how many elements it contains)

Strings

`String` is another special reference type, representing a string of text. A value of the `String` type — a *literal string* — is created inside the program as a sequence of characters enclosed in the matching pair of **double quotes** (the double quotes are **NOT** part of the string data).

All string literals in Java programs, such as `"abc"`, are objects of the class `String`, which is a part of the Java standard library (Java SE API). Strings are *immutable* (constant) — their values cannot be changed after creation.

```
String str1 = "abc"; // using a literal string to initialise a variable
char data[] = {'a', 'b', 'c'};
String str2 = new String(data); // using String class constructor
System.out.println(str1.equals(str2)); // testing equality of two strings
// should print "true", the objects str1 and str2 are identical
// But, str1 and str1 references were independently assigned, therefore
System.out.println(str1 == str2); // should print "false"
```

Characters which make up a string object can be read by using the method `somestring.charAt(i)`, where the *index* `i` is the position of the character inside the string (left-to-right, the **first character has index** 0!).

String Methods

The operations we just saw performed on strings — (str1.equals(str2)) and (somestring.charAt(i)) — are **methods** which can be *invoked* on a string object. Methods are operations which objects are programmed to perform.

Methods are characterised by a name followed by a pair of parentheses which may or may not include literal values or variable names in a form of a comma separated list — these are method (actual) parameters, they are used in carrying out the computation which method represents.

The *String* class includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase, *etc* (string operations are important, and *String* is a large class).

```java
System.out.println("abc"); String cde = "cde";
System.out.println("abc" + cde); String c = "abc".substring(2,3);
String d = cde.substring(1, 2);
char c1 = cde.charAt(0);
```

Further Reading

Cay Hortsmann *Core Java for the Impatient* Ch. 1.2, 1.5

David Goldberg *What Every Computer Scientist Should Know About Floating-Point Arithmetic*
http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

Josh Haberman *Floating Point Demystified*
http://blog.reverberate.org/2014/09/what-every-computer-programmer-should.html

Henry S. Warren *Hacker's Delight* 2ed, Addison-Wesley, 2013, ISBN-13: 987-0-321-84268-8
http://www.hackersdelight.org/