COP 3330, Spring 2013

Primitives, Literals, Strings, Scanner

Instructor : Arup Ghosh

01-14-13

School of Electrical Engineering and Computer Science
University of Central Florida

Primitives

- Almost all variables used in Java are objects, but there are notable exceptions.

- Primitives are eight standard data types which are pretty much identical to C.
  - Similar to C: `int, long, char, float, double`
  - Java-specific : `byte, short, boolean`

- They work almost exactly like their C equivalents.

Primitives: Key Points

- Using primitives with the basic operators
  - Exactly identical to C
  - Logical operators return booleans
- Unlike C, you can declare a variable when you use it
  - No need to do them all at the top.
- Default values
  - Unlike C, uninitialized primitives are not filled with junk values
- The various literals

Default literal types

- All integer literals are treated as `int`s
  - Compiler error if the literal is too big to fit in an int
  - Use the 'L' suffix to denote a `long` literal
- All floating-point literals are treated as `double`s
  - Use the 'F' suffix to denote a `float` literal
- You can use hex and binary for integer literals
- Scientific notation can be used for floating-point literals

Quick overview

- `int`: 32-bit integer
- `long`: 64 bit integer
- `float`: 32 bit real
- `double`: 64 bit real (preferred to floats)

- `char`: 16-bit Unicode character
  - Contrast with the 7-bit C char.

- `boolean`: Limited to true/false values.

- `byte` and `short`: 8 and 16 bit integers respectively

Numeric Data Types in Java

Type	Range	Storage Size
`byte`	-2^7 (-128) to 2^7-1 (+127)	8-bit signed
`short`	-2^{15} (-32768) to $2^{15}-1$ (+32767)	16-bit signed
`int`	-2^{31} (-2147483648) to $2^{31}-1$ (+2147483647)	32-bit signed
`long`	-2^{63} (-92233720368547758O8) to $2^{63}-1$ (+9223372046854775807)	64-bit signed
`float`	Negative range: -3.4028235E+38 to 1.4E-45 Positive range: 1.4E-45 to 3.4028235E+38	32-bit IEEE 754 standard
`double`	Negative range: -1.7976931348623157E+308 to -4.9E-324 Positive range: 4.9E-324 to 1.7976931348623157E+308	64-bit IEEE 754 standard

Literals

- Primitives are not objects created from a class.

- A literal is when you directly represent the value of a primitive in code.
  - Integer literal: `5, 32, 1000035`, etc.
  - Character literal: `‘x’, ‘A’, ‘\u1375’` (Unicode)
  - Boolean literal: `true, false`

- We often initialize primitives with literals:
  - `int x = 42;`
  - `boolean b = false;`
  - `double d = 3.14159;`

Casting

- Casting refers to coercing a value of one type into a different type.

- Java will auto-cast a ‘smaller’ type to a ‘bigger’ one when required.
  - `double d = 3;`

- `double` > `float` > `long` > `int` > `char`

- Casting from a bigger type to a smaller one must be done manually (see sample code in AR 5).

Wrapper Classes

Premitive types	Wrapper classes
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean
void	Void

primitives types and their wrapper class names

```
Integer myRoll = new Integer(50);
```

Details

- This page gives some more technical details on the primitives: http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html
- Pay special attention to the section on floating-point literals.
- There's also a section on using underscores in numerical literals for improved readability (Java 7 only)

Strings

- String is a class in the Java standard library.
- It is somewhat more sophisticated than C strings, which are just souped-up character arrays.
- String objects are what we use for almost all text I/O in Java.
- Look them up in the docs: http://docs.oracle.com/javase/6/docs/api/

Using strings

- Simple to create:
  - `String str = “hello”;`

- In code, text in double quotes is a String literal.
- Strings are the **only** non-primitive type that has literals.
- Mostly for convenience – strings are used so heavily that ease of usage is a necessity.

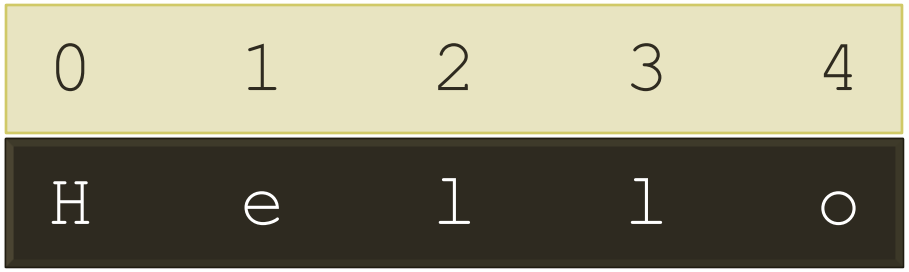
String concatenation

- Concatenation: Use the '+' operator
  - `String a = "Hello";`
  - `String b = "World";`
  - `String c = a+b;`
  - `System.out.println(c);  // Prints HelloWorld`

- This is another example of special treatment for strings.
  - Other than primitives, no other type has an overloaded operator ('+' in this instance).

String indexing

- Strings are indexed just like C strings.



- Here is a String of length 5.
  - First character has index 0
  - The last has index 4 (5-1).

The charAt method

- Java doesn't allow the [ ] operator for indexing into strings.
  - Works in C because strings are arrays. Not so in Java.

- Instead, use the charAt() method.
  - `String s = "Hello";`
  - `char c = s.charAt(4); // Now c contains 'o'`
  - `System.out.println(s.charAt(1)); // Prints 'e'`

- The `charAt()` method takes an `int` (the index) as parameter and returns a `char` (the character at that index)

Some more String methods

- `int length()`: Returns the length of the string
- `int indexOf(ch)`: Returns the first occurrence of the character ch in the string
- `String substring(int start, int end)`: Returns the substring starting at index start and ending just before index end.
- A full list is in the docs at: http://docs.oracle.com/javase/6/docs/api/

Strings: Key Points

- Declared like a primitive, except the type is `String`.
- The `'+'` and `'+='` operators are overloaded for concatenation.
- String literals are just text in double quotes.
- An uninitialized String object has the value null.
  - This is true for all objects, not just Strings.
- Trying to use a null object causes an error.

String methods

- Use the dot operator on a String variable to call a method.

- For instance, finding the length of a string named `s`:
  - `s.length();`

- WRONG!
  - `length(s)`
  - `length()`
  - `strlen(s);`

- This way you call the method *on* the object s.
  - Methods can also be called on a literal directly.

String methods

- Some string methods have parameters.
  - `String greeting = “Hello World!”;`
  - `String piece = greeting.substring(2, 9);`
- Translation: “Set `piece` to be the substring of `greeting` starting at index 2 and ending just before index 9.”
- So now `piece` contains the text `“llo Wor”`

The compareTo method.

- We can compare two strings to find their relative lexicographic order.
  - This is basically dictionary ordering generalized to include numbers, punctuation, etc.

- To compare String `a` to String `b`, do:
  - `int c = a.compareTo(b);`

- Essentially, "`a`, compare yourself to `b`".

- The returned value is:
  - Positive if `a` comes after `b` in lexicographic order
  - Negative if `a` comes before `b`
  - 0 if `a` and `b` are the same.

Strings: Immutability

- Once created, a String object is immutable – it cannot be modified.

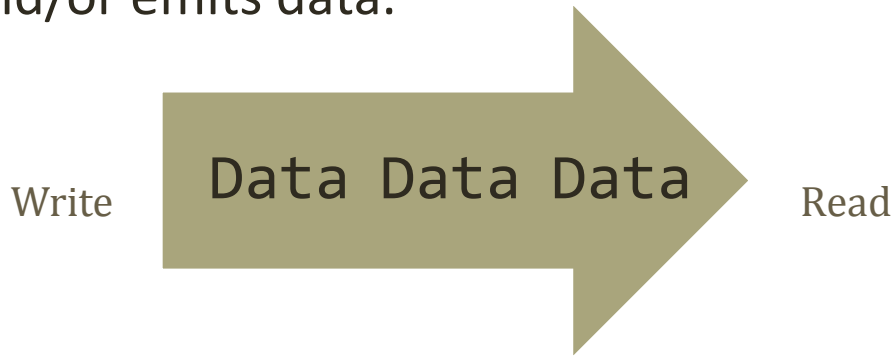
- Actions like concatenation, substring, etc. actually create a *new* string each time.

- This can get slow if you do a lot of operations.
  - E.g., concatenating 10,000 Strings can get slow.

- The `StringBuilder` class is a mutable String.
  - Sadly it doesn't do literals, concatenation with '+', etc.
  - But it's way faster for these situations.

Streams and I/O

- A stream is an abstraction for some sequential process that absorbs and/or emits data.



- For instance, a network connection can be viewed as a stream.

- Streams are the basic input/output abstraction in Java

Three standard streams

- Standard Input (stdin): For user input
  - Whatever the user types is written to stdin, and the program can read it from there.
- Standard Output (stdout): Output to screen
  - By writing to stdout, we can display data on screen.
- Standard Error (stderr): A separate output stream for error messages
  - Just like stdout, mostly just useful if we want to redirect it elsewhere.

How Java handles this

- Java has a preexisting object corresponding to each stream.
- These are located in the System class.
  - System.out: stdout
  - System.in: stdin
  - System.err: stderr
- By calling their methods, we can interact with the streams they represent.

PrintStream

- `System.out` is an object of class `PrintStream`
- A `PrintStream` provides the ability to conveniently print data from an output stream
  - By itself, a stream would only display a sequence of bits
- The methods `print()`, `println()` and `printf()` belong to this class
- `System.err` behaves exactly the same way.

InputStream

- `System.in` is an object of class `InputStream`
- We can use it to read user input.
- InputStreams are cumbersome to deal with directly, so we use the Scanner class instead.

Scanner

- A Scanner object hooks up to a source of data (like an input stream) and *parses* the stuff that it emits.
  - Parsing: Translating bit strings into various data types.

- So if an input stream spat out 32 bits, Scanner can convert that into an int for us.

- It has methods like:
  - `nextInt()`: Reads the next integer
  - `nextDouble()`: Reads the next double
  - `next()`: Reads the next word (String)

Creation

- `Scanner sc = new Scanner(System.in);`

- This is a general pattern for creating objects.
  - Strings are an exception, since they have literals.

- Translation:
  - "Create a Scanner named sc hooked up to System.in".

- The new operator creates a new object of the Scanner class.
  - More on this in a later lecture.

Usage

- Having created a Scanner object, we use its methods to read from the stream.

- Read a number the user typed in:
  - `int x = sc.nextInt();`

- See if the user has typed in an int:
  - `boolean typedInt = sc.hasNextInt();`

- Read a word the user typed in:
  - `String word = sc.next();`

Tokenization

- By default the Scanner decides where input begins and ends based on whitespace: spaces, tabs, etc.
  - Using delimiters this way is called tokenization.

- E.g., if the user types in 3 4.0 hello (note the spaces)
  - `int a = sc.nextInt();  // a = 3`
  - `double b = sc.nextDouble(); // b = 4.0`
  - `String c = sc.next();  // c = “hello”;`

- Instead of a single space, you can type multiple spaces, tabs, whatever – it knows what to do.

Sample Programs

- Sample programs using primitives and Strings
  - Subtraction.java
  - StringExample.java

Summary

- Read through the sample programs for examples
  - Pay attention to the underlying patterns
- The eight primitives are the only non-object types in Java
  - They have literals, as does the String type.
- Java strings are full objects, though they get some special treatment as far as language features go.
- Java Strings are immutable
  - Their methods create new objects instead of modifying the original.
  - Object creation is slow, so if your program does a million string operations…

Console Input Using the `Scanner` Class

- While there are several ways to enter data into a Java program while it is executing, one simple way is to use the `Scanner` class.
- Java uses `System.out` to refer to the standard output device (default is your terminal screen), and `System.in` to refer to the standard input device (default is your keyboard).
- To perform console output, you simply use the `println` method to display either a primitive value or a string to the screen. (Remember: `print` and `println` are identical except that `println` moves the cursor to the next line after displaying the string.)
- Console input is not directly supported in Java, but you can use the `Scanner` class to create an object to read input from `System.in` as follows:

```
Scanner input = new Scanner(System.in);
```

Console Input Using the `Scanner` Class

Method	Description
`nextByte()`	Reads an integer of the `byte` type
`nextShort()`	Reads an integer of the `short` type
`nextInt()`	Reads an integer of the `int` type
`nextLong()`	Reads an integer of the `long` type
`nextFloat()`	Reads a number of the `float` type
`nextDouble()`	Read a number of the `double` type
`next()`	Reads a string that ends before a whitespace. A whitespace character is ' ', '\t', '\f', '\r', or '\n'.
`nextLine()`	Reads a line of characters (i.e., a string ending with a line separator)

Methods In `Scanner` Class

A full list is in the docs at: http://docs.oracle.com/javase/6/docs/api/