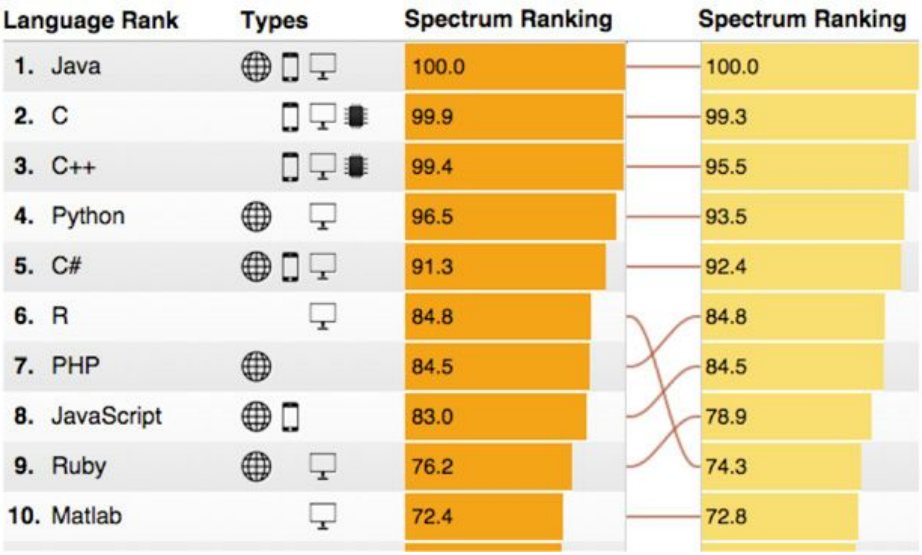
Reading 1: Intro To Java

Getting comfortable with the basics of programming in Java.

1.1 What is Java?

In this course, you will learn not only how to think like a computer scientist and solve problems algorithmically, but you will also learn the fundamentals of the Java programming language.

Java is an object-oriented programming language (If you don't know what object-oriented means right now, never fear! All will be explained eventually). Java is one of the most commonly used languages in the world today, ranking #1 in IEEE Spectrum's 2015 review of the most popular programming languages (see chart below, where the right column is 2014 and the left is 2015). All sorts of real-world projects use Java, from your Android phones and applications, to games like Minecraft.



1.2 Variables

Programming requires us to store and manipulate data. Data can come in the form of numbers, words, or even images. One of the most basic elements in programming that can help us store data is called a variable.

A variable represents a location in your computer's memory in which you can store a value and can access it later. For example, imagine you wanted to store your age as a number in your computer's memory. In Java, you would type:

```
int age = 19;
```

This variable is called "age" and stores the value 19.

Notice how we put the keyword `int` in front of our variable name. In Java, you have to specify what type of variable you are creating (for example, are you storing an integer, a decimal value, a character, etc.?). We will talk more about how to properly declare variables and assign values to them in section **1.4 Data Types**.

1.3 Syntax: Names in Java

Different parts of a program that you write need to have names. You have to name variables in your Java program, but not all names are valid. The following rules will help you give valid names to your variables:

1. All names begin with a letter (A-Z or a-z), $, or _
2. After the first character, names can have any combination of characters
3. You can't use reserved keywords (like `if` or `while` or `int`) for names
4. Names are case sensitive (ex. an identified named **Hello** is considered different than the identified **hello**)

Examples

- Legal names: `age, $alary, _value, __1_value`
- Illegal names: `123abc, -salary`

1.4 Data Types

As you learned in lecture, all information is stored as bits within the computer. A single bit can have the value of 0 or 1. 8 bits together make up 1 byte.

- 8 bits = 1 byte
- 1024 bytes = 1 kibibyte

- 1024 kilobytes = 1 megibyte
- 1024 megabytes = 1 gigibyte

Primitive Data Types

Variables in Java reserve a portion of memory in the computer for a certain amount of data. But how does the computer know how much memory to reserve? Based on the datatype of the variable, the operating system can reserve the right amount of memory for this variable. You can store integers, decimals, characters, and more in variables. In the chart below, you will find a description of the primitive data types used in Java.

Primitive Data Type Chart

`byte`

- 8-bit, signed (meaning that it can be negative or positive) integer
- Min value = -2^7
- Max value = 2^7 -1

`short`

- 16-bit, signed integer
- Min value = -2^15
- Max value = 2^15 -1

`int`

- 32-bit, signed integer
- Min value = -2^31
- Max value = 2^31 -1

`long`

- 64-bit, signed integer
- Min value = -2^63
- Max value = 2^63 -1

Examples

- `byte a = 100;`
- `byte b = -50;`
- `int foo = 12345;`
- `long a = 100000L;`

`float`

- 32-bit, single-precision floating point (i.e. can store decimal values)

`double`

- 64-bit, double-precision floating point

`boolean`

- Represents 1 bit of information
- Only 2 possible values: true and false

`char`

- 16-bit Unicode character
- Used to store any character

Examples

- `float f1 = 234.5f;`
- `double d1 = 123.4;`
- `boolean one = true;`
- `char letterA = 'A';`

1.5 Operators

Operators are symbols in Java that allow us to perform arithmetic or logical operations. Some of the most basic operators are binary operators which take two values and compare them. In the table below you will find some common binary operators.

Note that in all of the examples, we will use the variables

```
int a = 1;
int b = 2;
```

which store integers of value 1 and 2 respectively .

Type	Binary Operator	Example
Additive	+ (addition) - (subtraction)	`a + b` → evaluates to 3 `a - b` → evaluates to -1
Equality	== (equals) != (not equals)	`a == b` → evaluates to **`false`** `a != b` → evaluates to **`true`**
Multiplicative	* (multiplication) / (division) % (modulus)	`a * b` → evaluates to 2 `a / b` → evaluates to 0 `a % b` → evaluates to 1
Relational	> (greater than) < (less than) >= (greater than or equal to) <= (less than or equal to)	`a > b` → evaluates to **`false`** `a < b` → evaluates to **`true`** `a >= b` → evaluates to **`false`** `a <= b` → evaluates to **`true`**

There are a few subtleties regarding binary operators that we haven't explained yet - don't worry! Here are some clarifications to common questions:

- **Q: Why would `a / b = 0` when `a = 1` and `b = 2`?**
  - Notice that the variables a and b were declared with the keyword **`int`**, meaning they can only store whole numbers. Thus, when Java performs integer division, it only stores the quotient (in our case 0) but *not* the remainder.
- **Q: What does the % actually do?**
  - The modulus operator in Java will give you the remainder of integer division. Thus, **`a % b = 1`** when a = 1 and b = 2, since 1 goes into 2 zero times with a remainder of one.
- **Q: What is the difference between = and ==?**
  - In Java, there is a difference between assigning a value to a variable and checking if that variable equals a value.
  - **=** is the assignment operator. This lets us assign values to a variable. For example, we assigned the value of 1 to the variable **a** by writing: **`int a = 1;`**

- **==** is the equality operator. This lets us compare the value of primitive types to see if two values are equal or not. For example, we can compare if `a == 5` (which in our case is false).

1.6 Converting Between Bases

Since everything is stored as bits inside computers, we need to know how to convert from our intuitive number system that is in base 10 (decimal), to the base 2 (binary) or base 16 (hexadecimal) systems often used in computers.

When working in base 10, the furthest right digit is the 1s place, followed by the 10s place, the 100s place, the 1000s place, etc. When working in base 2, recall that the furthest right digit is the 1s place, followed by the 2s place, the 4s place, the 8s place, etc. There is a clear pattern here: when working in base x, the furthest right digit is x^0=1s place. The next is the x^1=x place, then the x^2 place, and so on with powers of x.

BINARY TO DECIMAL

To convert a number from base 2 to base 10, simply think about the value in each digit and add them up. The best way to understand this method is through an example.

Example. Take 101100101_2, which is in binary, and convert it to decimal. Construct a table which shows each digit place and whether there is a 1 or a 0 there.

Place	256	128	64	32	16	8	4	2	1
Digit	1	0	1	1	0	0	1	0	1

If this is confusing, think about a simpler case in the decimal system. For example, for the number 13 in base 10, we know we can write 13 as using one 10 and three 1s to get 1*10 + 3*1 = 13. The same process is followed in binary.

101100101_2 = 1*256 + 0*128 + 1*64 + 1*32 + 0*16 + 0*8 +1*4 + 0*2 + 1*1

= 1*2^8 + 0*2^7 + 1*2^6 + 1*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0

= 357_{10}

So 101100101 in base 2 is equivalent to 357 in base 10.

This same process works for other bases as well. If working in base 5 for instance, you would think of the 1s place, the 5s place, the 25s place, the 125s place, and so on, but the summing up of the digits works just the same.

Your turn!

Here are a few conversions to try out. When you finish, you can go to https://www.tools4noobs.com/online_tools/base_convert/ to verify your answers.

1. Convert 1101 from base 2 to base 10.
2. Convert 111111 from base 2 to base 10.[1]
3. Convert 2102 from base 3 to base 10.

DECIMAL TO BINARY

A lot of people find converting from decimal to another base harder than the other way. Below are two different ways to think about the process. Again, we will use an example to demonstrate the methods. Consider 357_{10} and convert it to binary.

Method 1:

Find the largest power of 2 which is less than 357. If you're still playing the game 2048[2], you'll know that 2^8=256, while 2^9=512, which is too big. Therefore, we know that we want a 256 as a component of our number. Recall from the previous section that we added the different places together, so when working backwards, we subtract. 357-256=101. Repeat the process. The highest power of 2 which is less than 101 is 2^6=64, so we want to include a 64 in our number. 101-64=37. 2^5=32 is less than 37, so we want a 32. 37-32=5. 2^2=4 is less than 5, so we want a 4. 5-4=1. Finally, there is only 1 left, so we want a 1 in the 1s place. So now we have a 256, a 64, a 32, a 4, and a 1. We fill in the other places with 0s. We have no 128, 16, 8, or 2. Putting these in order in a neat table, we get:

Place	256	128	64	32	16	8	4	2	1
Digit	1	0	1	1	0	0	1	0	1

You may have already noticed that this is the same number we used in the binary to decimal conversion above. All that's left is to put the digits together to get the number 101100101_2. We got back to the number we started with in the previous example, verifying again that $101100101_2 = 357_{10}$.

Method 2:

The conversion from decimal to binary is simply a series of divisions by 2 (because we want

[1] Interesting note: when a binary number is all 1s, it is 1 less than a power of 2. Similarly, 99 is 1 less than 10^2 and 99999 is 1 less than 10^5.

[2] If you haven't played this game, here's where to play it: http://gabrielecirulli.github.io/2048/

base 2). With each division, keep track of both the quotient (the integer result of the division) and the remainder. Keep a list of the quotients and remainders. Again, start with 357_{10} and convert to binary. The first step is to divide 357 by 2, yielding a quotient of 178 and a remainder of 1. Next, divide 178 by 2. Repeat this process until the quotient is 0 or 1. Here is the table for the number 357.

Step	Quotient	Remainder
1	178	**1**
2	89	**0**
3	44	**1**
4	22	**0**
5	11	**0**
6	5	**1**
7	2	**1**
8	**1**	**0**

Now simply read from the table to find the number in binary. Start with the very last quotient first (**IMPT**: do not forget to include the final quotient!), followed by the remainders, *from bottom to top*. These numbers are bolded in the table. This method gets the result 101100101_2, which is again consistent with the previous conversions.

Come up with some of your own conversion examples to practice. If you really want to get fancy, try converting decimal to other bases, like base 4. The process stays the same, with division by 4 instead, and the stopping case being when the quotient is less than 4.

HEXADECIMAL AND BINARY

First, what is hexadecimal? Hexadecimal, or hex for short, is simply base 16. Before going into the conversions, you first need to understand hex. We only have names for 10 digits, 0-9, so a 16 digit system needs 6 additional digit names. To solve this problem, hex uses the letters A-F to represent the numbers 10-15. Here is a chart showing the digits of the hexadecimal system, as well as the corresponding binary numbers:

Decimal	Hex	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Notice anything? Each hex number is exactly 1 digit and each binary number uses up to 4 digits. The next number, 16, would overflow in both bases, requiring 2 digits in hex (10) and 5 digits in binary (10000). This is not a coincidence! It is a direct result of the fact that 2^4=16. While conversion between binary and hex can be done using the methods above, this special case allows us to use trick to make the conversion easier.

For every digit of hex, there are exactly 4 digits of binary. To convert A09 from hex to binary, for instance, simply convert **A** to **1010**, **0** to **0000**, and **9** to **1001**, then put them together to get **101000001001** in binary.

To go the other direction, simply start at the right (this is very important!) and take groups of 4 and convert them each. To convert **110110000110**, convert **0110** to **6**, **1000** to **8**, and **1101** to **D**, then put them together to get **D86**. But wait! What about cases where the number of binary digits isn't a multiple of 4? Not to fear. Simply add 0s to the left of the number until there are a multiple of 4 digits. This doesn't change the number (10 and 010 are the same thing). Note: if you remember to do this check first, it won't matter whether you go right to left or left to right, but starting from the right will force you to notice the problem by the time you arrive at the furthest left digits.

OCTAL AND BINARY

Octal is simpler to wrap your head around than hex because there are no new digits needed. Instead, the digits are just 0-7. The reason we had the trick for binary and hex was because 2^4=16. Similarly, we have a trick for binary and octal because 2^3=8, so the same processes are used, but with groups of 3 digits at a time. You can make a table of the 8 simple binary to octal conversions and come up with examples on your own to practice.

To convert from octal to binary, take each digit in order and convert it to 3 digits of binary, then put them all together. To convert from binary to octal, pad the left side with 0s (if necessary) to get a multiple of 3 digits, then convert each set of three binary digits to a single octal digit.

1.7 Coding Disasters[3]

We want these readings to not only help guide you through this course, but also show you that what you are learning has real-world applications and potentially huge impacts!

[3] Otherwise known as, "Why you should pay attention in this class."

For this first reading, we're going to kick things off with what NOT to do when coding -- and why it's really important to get have a rock-solid programming foundation.

NASA Mariner Rocket Destroyed - Carrying a space probe and headed for Venus, the Mariner 1 rocket was diverted from its flight path and destroyed only 5 minutes after liftoff. Why did this happen? When transcribing a handwritten formula into code, a programmer made a single typo in a variable name and the software treated normal variations of flight velocity as if they were serious, sending the rocket off course. The total cost of this tiny error? A heaping $18.5 million.

Lethal Radiation - The Therac-25, a Canadian radiation therapy machine, malfunctioned and delivered lethal radiation doses to patients, killing 3 people and critically injuring 3 others. The bug at fault was a race condition (don't worry about the subtleties of what a race condition is -- you'll learn more about this in CMSC132), which occurred when two threads (I promise you'll learn this too!) of the program tried to access and change the same variable.

Ariane Rocket Explodes - The European rocket, Ariane 5, was destroyed only seconds into its first launch due to an overflow error. The guidance computer tried to convert the sideways rocket velocity from 64-bits to 16-bits. However, since the 64-bit number was too big to fit into a 16-bit number, an overflow error occurred and the guidance system shut down.

1.8 Contributors

Amelia Bateman and Andrea Bajcsy

Sources

- http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages
- http://www.tutorialspoint.com/java/index.htm
- https://docs.oracle.com/javase/tutorial/java/nutsandbolts
- http://w3processing.com/index.php?subMenuItemId=135
- http://www.purplemath.com/modules/numbbase.htm
- http://www.devtopics.com/20-famous-software-disasters/

Last Modified

January 2, 2016