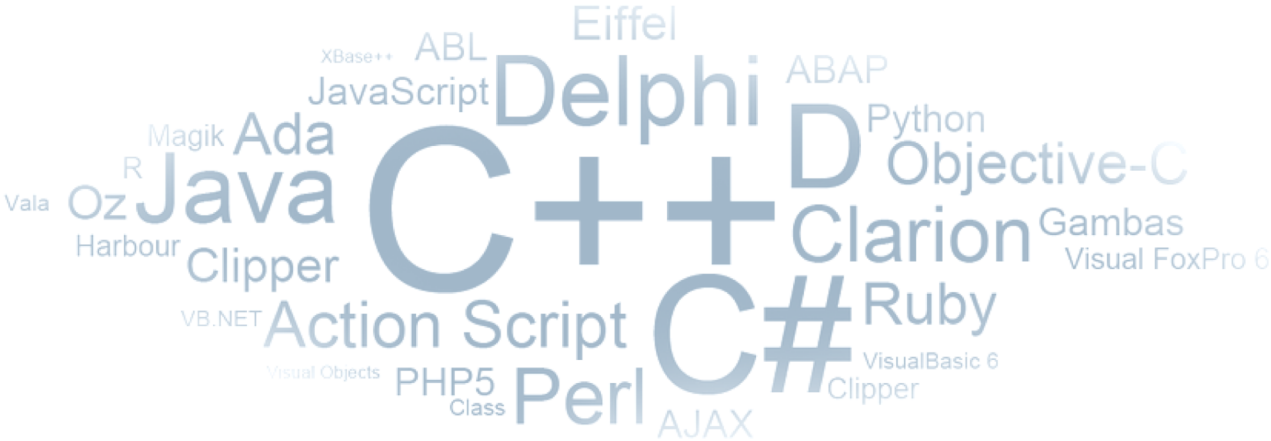
CIS 351-Data Structure
Jan 16, 2020

Dr. Farzana Rahman

Syracuse University



What is Abstract Data Type

Primitive data types – contain only data at a time

ex. int num = 0;

Abstract Data Type (ADT) -

- An opportunity for an acronym
- Mathematical description of an object and the set of operations on the object
- An **abstract data type (ADT)** is a programmer-defined data type that specifies a set of data values and a collection of well-defined operations that can be performed on those values.
- ADT are defined independently of their implementation.

What is Data Structure?

- A data structure is a (often *non-obvious*) way to **organize information** to enable *efficient* computation over that information
  - Objects - collection of data items of various types
  - arrays - collection of data items of the same type, stored contiguously
- This course will cover some more complicated data structures
  - how to implement them efficiently and what they are good for
- A data structure supports certain **operations**, each with
  - **Meaning:** what does the operation do/return
  - **Performance:** how efficient is the operation
- Examples:
  - ***List*** with operations **insert** and **delete**
  - ***Stack*** with operations **push** and **pop**

More Data Structure

- But there are unavoidable **trade-offs**:
  - Time vs. space
  - One operation more efficient if another less efficient
  - Generality vs. simplicity vs. performance

- We ask ourselves questions like:
  - Does this support the operations I need efficiently?
  - Will it be easy to use (and reuse), implement, and debug?

- What assumptions am I making about how my software will be used? (E.g., more lookups or more inserts?)

What this course will cover

- Introduction to Algorithm Analysis
- Lists
- Stacks
- Queues
- Trees
- Hashing
- Heaps
- Priority Queues
- Sorting
- Searching
- Graph

All in Java Programming Language

Object Oriented Programming
Classes and Objects
Inheritance
Polymorphism
Arrays
ArrayList
Programming Fundamentals

Why so many data structures?

Ideal data structure:

fast, elegant, memory efficient

Generates tensions:

- time *vs.* space
- performance *vs.* elegance
- generality *vs.* simplicity
- one operation's performance *vs.* another's

Dictionary ADT

- **list**
- **binary search tree**
- **AVL tree**
- **Splay tree**
- **Red-Black tree**
- **hash table**

ArrayList or LinkedList

```
import java.util.ArrayList;
import java.util.LinkedList;

public class ListDemo {

  public static void main(String[] args) {

    LinkedList<String> list = new LinkedList<String>();

    for (int i = 0; i < 200000; i++) {
      list.add(0, "A");
    }

    for (int i = 0; i < 200000; i++) {
      list.remove(0);
    }

  }
}
```

What is the result of changing the code snippet to use an ArrayList instead of a LinkedList?

```
for (int i = 0; i < 200000; i++) {
     list.add(0, "A");
}
for (int i = 0; i < 200000; i++) {
     list.remove(0);
}
```

A) No change in correctness or execution time. No one notices you made the change.

B) An ArrayList will not work correctly in this case. You've broken the code! You're Fired!

C) The new implementation is slightly slower than the original implementation.

D) The new implementation is slightly faster than the original implementation.

E) The new implementation is MUCH faster than the original version. Nice Job! Promoted on your first day!

F) The new implementation is MUCH slower than the original version. You're fired!

List, ArrayList, LinkedList

- Java interfaces can be thought of as ADT's*:
  - List
- Java collection classes are data structures:
  - ArrayList
  - LinkedList
- *Some authors would argue that an ADT is an abstract mathematical description that is not tied to any particular language construct.

Back to Linked List vs. ArrayList...

- LinkedList CRUSHED ArrayList in our earlier code snippet. Which, if any, of these will be much *faster for an ArrayList* (assuming a large number of entries)?

A) list.get(0);

B) list.get(list.size() - 1);

C) list.get(list.size() / 2);

D) None of the above.

Data types in Java

- Primitive type
- Reference type

Reference type

- A *reference type* is a data type that's based on a class rather than on one of the primitive types that are built in to the Java language.

- To declare a variable using a reference type, you simply list the class name as the data type.

- For example, this is how you have a String variable, which is reference type – `String str;`

- `We will discuss String in good detail`

Primitive Data types

- We can store different types of data

There are eight primitive data types in Java

- Four of them represent integers:
  - `byte`, `short`, `int`, `long`
- Two of them represent floating point numbers:
  - `float`, `double`
- One of them represents characters:
  - `char`
- And one of them represents boolean values:
  - `boolean`

Data Types - char

- Holds only a single character
- Legal Examples
  - char myMiddleInitial = ‘M’;
  - char myGradeInChemistry = ‘A’;
  - char aStar = ‘*’;
  - char aCharValue = ‘9’;
  - char aNewLine = ‘\n’;
  - char aTabChar = ‘\t’;

The char variables still hold a single character

The backslash gives a new meaning to the character that follows

The pair together represents a single nonprinting character

Escape sequence

Variables

- A *variable* is a name for a location in memory
- Variables allow a program to store data at one point and refer back to it later
- A variable must be *declared* by specifying the variable's name and the type of information that it will hold

data type

variable name

```
int total;
```

A variable holds only one value at a time

- Multiple variables can be created in one declaration:

```
int count, temp, result;
```

- A literal is a value that is written into the code of a program

Assignment and Equal

- An *assignment statement* changes the value of a variable
- The equals sign is also the assignment operator

```
total = 55;
```

- The expression on the right is evaluated and the result is stored as the value of the variable on the left
- The value previously stored in `total` is overwritten

=	Assignment
==	Equal Checking

Notes: Initializing Variables

- A variable can be initialized with a value, the value of another variable, or by evaluating an expression

```
// declare a char named letter initialized to ‘a’
char letter = ‘a’;

// declare a double named d1 initialized to 132.32
double d1 = 132.32;

// declare a double named d2 initialized to d1
double d2 = d1;

// declare a float name z initialized to x * y + 15
float z = x*y + 15.0f;
```

Variable Declaration, initialization, assignment

- Variable Defintion

```
int a;
```

- Variable Declaration/Initialization

```
int a = 20;
```

- Variable Assignment

```
int a;
a = 20;
```

Constants

- Constants are similar to variables except that they hold a fixed value. They are also called "READ" only variables.

- Constants are declared with the reserved word "final".

```
final int MAX_LENGTH = 420;
final double PI = 3.1428;
```

- By convention upper case letters are used for defining constants.

Escape Sequence

- An *escape sequence* is a series of characters that represents a special character
- Escape sequences begin with a backslash character (\)

```
System.out.println ("I said \"Hello\" to you.");
```

Don't confuse backslash (\) with a forward slash (/)

- Please look at Table 2-2 (Chapter 2) of textbook - "Common Escape Sequences"

identifier and keyword

- Identifier: Programmer defined names
  - class name, variable name
- Keyword: makes up the core of the java language (i.e. language defined)
  - If-else, constant, null, final, long, static, …etc

The println Method

- We use `println` method to print a character string
- The `System.out` object represents a destination (the monitor screen) to which we can send output

```
System.out.println ("Whatever you are, be a good one.");
```

object

method name

information provided to the method (parameters)

How do you print your name?

```
System.out.println ("Farzana Rahman");
```

Compatible Data Types

Any type in right column can be assigned to type in left column:

Data Type	**Compatible Data Types**
byte	*byte*
short	*byte, short*
int	*byte, short, int, char*
long	*byte, short, int, long, char*
float	*float, byte, short, int, long, char*
double	*float, double, byte, short, int, long, char*
boolean	*boolean*
char	*char*

Sample Assignments

- This is a **valid** assignment:

```
float saxXlesTax = .05f;
double taxRate = salesTax;
```

- The following is **invalid** because the *float* data type is lower in precision than the *double* data type:

```
double taxRate = .05;
float salesTax = taxRate;
```

Type casting

Forces a value of one data type to be used as a value of another type

- **Example**

```
Double d1= 189.66;
Float d2= (float) d1/ 4;
```

- Explicit Type Casting: Syntax:

```
(dataType)( expression )
```

- `X = (int) number`

If `number` is `double/float` its fraction part will be truncated and number will be converted into an `int`

Mixed-Type Arithmetic

When performing calculations with operands of different data types:

- Lower-precision operands are promoted to higher-precision data types, then the operation is performed
- Called "implicit type casting"

any expression involving a floating-point operand will have a floating-point result

Comparison operators

< less than

> greater than

== equal to

<= less than or equal to

>= greater than or equal to

!= not equal to

The result is boolean, always

Operator precedence

- In a complex expression with several operators, Java uses rules of *precedence* to decide the order in which to apply the operators.

- **Precedence**: **Order** in which **operations** are computed in an expression.

Operator	Order of evaluation	Operation
( )	left - right	parenthesis for explicit grouping
* / %	left - right	multiplication, division, modulus
+ -	left - right	addition, subtraction
=	right - left	assignment

Precedence examples

```
1 * 2 + 3 * 5 / 4
  \ _ /
   |
   2    + 3 * 5 / 4
            \ _ /
             |
   2    +   15    / 4
                \ ___ /
                  |
   2    +        3
     \ _____________ /
            |
            5
```

```
1 + 2 / 3 * 5 - 4
      \ _ /
       |
1 +    0    * 5 - 4
          \ ___ /
           |
1 +        0      - 4
  \ _________ /
     |
     1            - 4
       \ _____________ /
            |
           -3
```

Combined assignment operators

+= x+=2 → x =x+2

-= x-=2 → x =x-2

= x=2 → x =x*2

/= x/=2 → x =x/2

Shortcut Operators

`++` increment by 1 `--` decrement by 1

Example:

```
count++;     // count = count + 1;
count--;     // count = count - 1;
```

Postfix version (`var++, var--`): use value of *var* in expression, then increment or decrement

Prefix version (`++var, --var`): increment or decrement *var*, then use value in expression

Inc and Dec

```
x = 5;
y = ++x;
```

y = 6

```
x = 5;
y = x++;
```

y = 5