Table of contents

Week 4

Char is used to represent any character.

Java uses unicode for encoding here is how to work with it:

```
char c = 'A';
int i = (int)c;
```

The above block stores the hexadecimal value of c in i.

Similarly, if we tried character arithmetic with int's the compiler ends up working with the encoded value of char and the assigned value of the int.

And here is how we read a char as a user input:

```
Scanner scan = new Scanner(System.in);
String s = scan.next();

char c = s.charAt(0);
```

Difference between = and ==

```
// this assigns s as a reference to hello world.
String s = "hello world"
// this compares the adress of v1 with that of v2
v1 == v2
```

Arraylist and Array data types

In general we choose an arraylist over an array whenever either the table size is initially not known or we would like to extend the structure later on in the code.

Array syntax:

```
\\declaration
int[] scores;
scores = new int[2];
\\ or a shorther way is
```

```
int[] scores = new int[2];
\\ or more simply
int[] scores = {1600, 1350, 990};
```

When declaring arrays, if each variable is unassigned here are the default values:

```
int 0
double 0.0
boolean false
char ’\u0000’
(any other object) (null)
```

And some array methods now:

```
scores.length; //returns length of n-1
```

Generally, if we want to make a copy of an array, we can not use the = operator as that simply creates an extra reference towards the same array. We would instead loop through:

```
for (int i = 0; i<b.length; ++i){
    a[i] = b[i];
}
```

Similarly, to test if two arrays are equal we use the Arrays class:

```
Arrays.equals(arr1, arr2);
```

And now we come to the interesting bit, that it tables of multiple dimensions:

```
// representing the 3x3 identity matrix
int [][] = {
    {1,0,0},
    {0,1,0},
    {0,0,1}
}
```

And the best way to iterate through a multidimensional array is nested for loop:

```
for(int i = 0; i < y.length; i++){
    for(int j = 0; j < y[i].length; j++){
        System.out.println(y[i]);
    }
}
```

And finally we list some ArrayList syntax and its methods :

```
ArrayList<int> t1 = new ArrayList<int>();

.size() // instead of .length
.get(index);
.set(index);
.remove(index);
.add(value);
.equals(content);
.clear();
.isEmpty();
```

Parametrized types

It may have appeared why we declare an arraylist as

```
ArrayList<Int> test = new ArrayList<Int>();
```

The reason for the passage of argument Int is that ArrayList is an example of a parametrized type, meaning it is able to accept a type as an argument **yet does not accept primitive data types such as int**. This is useful because we can create new object that store methods specific to the type argument.

Week 5

We start with an important reminder:

```
ArrayList<Integer> test = new ArrayList<Integer>();

test.add(200);
test.add(200);

System.out.println(tab.get(0) == tab.get(1)); //prints false
System.out.println(tab.get(0).equals(tab.get(1))); // prints true
```

Extension point of the week (Unicode)

Firstly, unicode is not simply a 16-bit code system where each character takes 16 bits.

Unicode has this notion of code points.Now every platonic letter in every alphabet is assigned a unique id such as U+0639(where the extension is in hex). For instance the string "hello" is represented as:

U+0048 U+0065 U+006C U+006C U+006F

This brings us to UTF-8. In UTF-8, every code point from 0-127 is stored as a single byte. Other code points above 127 are given 2,3 up to 6 bytes.

Methods and defaults

To prevent unneccesary coding we use the concept of method overloading.

```
public static void display(char aChar, int count = 4// thus sets default value){
    for(int i = 0; i< count; i++){
        System.out.println(aChar);
    }
}

public .... main(){
    display("*") // invokes default
    display("*", 10) //overloading
}
```

In addition, Java has the useful ellipsis feature wherein the number of arguments to a method is left variable.

```
int m(int ... a){
    return int b;

// or even better

int sum(int ... a){
    int sum = 0;
    for(int a: a){
        sum += a;
    }
    return sum;
}
}
```

Why use methods?

The point in using a method is simply to eliminate redundant code. Hence, we define a method as being a reusable piece of code. And here is some method jargon:

> *method name* serves as a reference to the method object

> *method field* the reusable part of the code aka. set of instructions

> *method variables* scope is only within method and get destroyed as soon as method quits

We come to the question of where method calls may be made. So long as we use the static keyword, we are able to call a method in the main function whether the method is called for variable assignment or passed as an argument to another method. For instance:

```
static int[] readScore(int n){
    Scanner scan = new Scanner(System.in);
    int[] scores = new int[n]; //will store n many scores
    for(int i = 0; i < n; i++){
        System.out.println("enter score" + i);
        int[i] = scan.nextInt();
    }
    return scores; //if we dont declare a return, scores simply gets destroyed as
soon as method is done.
}

public static void main(String Args[]){
    int[] epflScores = readScore(100);
    // or
    printScores(readScore(50));
}
```

We note that when a return type is not declared, a method defaults to void. In addition, **return must be the last invocation in a method** and similarly, the provocation of **return cannot be conditional**.

Method evaluation

For some method like

```
m(int x1, int x2,...)
// when we pass
m(2, 3, ...);
// we have that x1 stores a reference to 2 in memory and so on.
```

Yet another important concept is **passage by value vs. passage by reference** We highlight that all data is passed by **value** in java. Consider this:

```


int x = 60;
static void change(int val){
    val = val*2;
}

public static void main(String Args[]){
    change(x);
    System.out.println(x); // counter to what you think, this prints 60, not 120!
}
```

The above behaviour happens because when we call change(x), we are essentially making val = x meaning that val is a copy of the object x. Yet, for primitive data types, this is not the case. We have that both the copy val and x point to the same value hence when val changes, so does x!