1. Contents

- Syntax of throws
- Assertions

1.1. 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();
```

1.1.0.1. 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.

1.2. 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
```

1.2.0.2. 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.

1.2.0.3. 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;
}
}
```

1.2.0.4. 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**.

1.2.0.4.1. 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 v.

1.3. Week 7

1.3.0.5. Concepts in OOP

The main notion of OOP is encapsulation. The idea is to capture related attributes and methods under one common hood.

Now there are two sides to encapsulation:

> We encapsulate similar attributes and methods under one object. We may also define an *interface* which is a set of methods that an object *implements* and it is up to the programmer to use these.

And most importantly, our design must be such that we are able to change the implementation without causing any change in the interface.

We note that any class attribute has a global scope within the class and hence do not need to be parametrized inside methods. ie.:

```
class Test{
    double height;
    double getHeight(){
        return height;
    }
}
```

Whenever an attribute must not appear outside the instance of an object, we limit its scope using the *private* keyword.

```
class Rectangle{
    private int height;
}

class Test{
    System.out.println(rect.height); // gives error
}
```

In general, it is good practice to only make methods and attributes that are used by the client-side public. An example to this is the famous getters and setters.

1.3.0.6. Copy constructor

Suppose we want to define a constructor so that we can create a separate copy instance(this is because if we set two objects equal to other than they both point to the same object in memory.)

```
class test{
    private int val1;
    private int val2;

    public test(test instance){
        val1 = instance.val1;
        // and so on.
    }
}
```

1.4. Week 8

1.4.0.7. Introducing inheritance

The core idea is to define a tree of parent and children class where one child class can extend only one parent class at a time. Now the child class will inherit **variables, methods, type** of the parent class. A critical point is that if the parent class has private variables or methods, they are invisible in the child class. Yet a solution to overcome this is using the *protected* modifier. Yet note that this does not respect the notion of encapsulation since protected keyword gives access to variables in all classes found in the same package. Hence a workaround is to use getters and setters defined in the parent class for access in subclasses.

When instantiating a subclass, we use a separate constructor for instantiating attributes of the superclass. Here's the syntax":

```
Class Car{
    Car(){
        System.out.println("im a car");
    }
}

Class Ferrari extends Car{
    Ferrari{
        super();
        System.out.println("im a ferrari");
    }
}
```

The above is always good practice because it avoids a compilation error since the *super* keyword invokes the parent class constructor assuring access to private variables. And some rules for parent class instantiation:

1. Each constructor of a subclass must call *super*
2. Arguments provided to super must match constructor of parent class.

3. call to super must be first instruction.
4. no other method must call super

> as an aside, note that a default constructor(this is either no constructor or a constructor without any arguments) is provided only if no constructor has been defined. Consider these examples:

```
class Rectangle {
    private double largeur;
    private double hauteur;
    // il y a un constructeur par defaut !
        public Rectangle()
        { largeur = 0; hauteur = 0;}
        // le reste de la classe...
        };
class Rectangle3D extends Rectangle {
    private double profondeur;
    public Rectangle3D(double p)
    {profondeur=p;}
}
```

And yet another way to call the parent constructor is through *this(...)*

1.4.0.8. Dynamic method dispatch aka. runtime polymorphism

Suppose we have:

```
class first{
    void display(){
        print("first")
    }

    ... main(...){
        first ref;
        ref = new second();
        ref.display();
    }
}

class second extends first{
    void display2(){
        print("second")
    }
}
```

Now according to runtime polymorphism we have that only the type of object is checked and not the type of its reference. Hence the display method of second class will be called.

Note that for runtime polymorphism, it is not the compiler that decides which method to call but the JVM.

1.4.0.9. Java packages:extra of the week

The idea of a package is to group classes with similar behaviour under the same package. For instance suppose we are building an RPG game, then we may create a package for all player classes, a package for GUI classes and so on. Packages also prevent naming conflicts.

Suppose we have a project relating to cartography then surely we will have a class called Map. Now the standard Java library also has a Map class hence we are sure to have a conflict. The offered solution is to place our Map class into the package Carto and use the statement:

```
import Carto.Map

Map.show();
```

And whenever we are in the same package, we may simply omit the import statement of course not forgetting the package statement. To import totality of names in a class we use an asterisk(*):

```
import Carto.*

Map.show();
```

1.5. Week 9

1.5.0.10. Introducing polymorphism

There are two types of polymorphism:

1. polymorphism of methods
2. polymorphism of variables

1.5.0.11. Polymorphism syntax

Abstract methods are used when numerous subclasses share some common method with a different implementation for instance:

```
abstract class Lift{
    public abstract void performLift();

}

class Lifter1{
    public performLift(){
        ...out("I lifted 200kg");
```

```
    }
}

class Lifter1{
    public performLift(){
        ...out("I lifted 100kg");
    }
}
```

And bindings of *abstract* are:

1. must exist(be defined) in all instantiable subclasses
2. abstract classes can not be instantiated

1.5.0.12. Static vs. dynamic method dispatch

The most simple difference between the two is that the former corresponds to *method overriding(aka. same method name with different number of arguments)* and the former corresponds to *method overloading*. The latter as mentioned before is where polymorphism occurs at compile-time. That is the compiler checks the object type ignoring the reference type and executes method found in the object. We also note that only methods in java have dynamic dispatch, same doesn't apply to variables. We also add that java doesn't allow multiple dispatch that is polymorphism relative to method argument. Hence

1.5.0.13. Bad OOP practice

Something to do and then slap yourself is using instanceof. That is, our child classes are always an instance of their parent classes hence will always validate the instanceof test.

1.5.0.14. Constructors and polymorphism

```


abstract class a{
    public abstract void m();
    public A{
        m();
    }
}

class B extends A{
    private int b;
    public B(){
        b = 1;
    }

    public void m(){
        ...print("b is" + b);
    }
}

B b = new B();
```

The above code prints 0 because the constructor of B calls the default constructor of A which through method overriding calls m and b is uninitialized hence 0.

> As a sidenote, although unrelated, the *override* annotation lets the compiler give an error in case the method to be overridden is mispelled.

1.5.0.15. Polymorphism jargon

> **Upcasting** Casting an object of type super-class to its subclass. This is always safe because a subclass can do everything that the parent class can do.

> **Downcasting** Casting an object of type sub-class to superclass. This is done as:

```
class A{
    ...
}

class B extends A{
    ...
}

A obj1 = new B();
B obj2 = (A)obj1;
```

We note that casting of primitives in java is quite different than casting of reference types. Since primitives hold the actual value in memory, casting one primitive to another irreversibly changes the value. When casting reference types however, only the reference address is changed which has the effect of narrowing or extending methods but not destroying the previously referenced object. The crucial part is, whenever we upcast, we limit the number of callable methods however out of the any two common methods to the subclass and parent class, it is the subclass method that gets called because our object is still an instance of subclass.

1.5.0.16. The global superclass *Object*

Every class in java inherits the class object. Some useful methods of the object class are(noting that all these methods are static, hence can only call inside class):

1. toString()
2. equals()
3. clone()

Similarly every Java class also extends the Exception class for exception handling.

> As an aside for object comparison, it is advised to use the getClass method instead of the instanceof operator to prevent collisions caused by parent classes.

1.5.0.17. Is-a and Has-a in Java

The Is-a relationship is essentially inheritance. Every subclass is also an instance of the parent class. The Has-a relationship is when one instance of an object is stored inside another object.

1.6. Week 10

1.6.0.18. Static modifier

Declaring some variable as static means its value is the same amongst all instances. This also means that the variable is some reference to the same zone in memory amongst all instances.

Declaring a method as static, the method may be called without instantiating an object.

> And here are properties of static variables:
>
> 1. must be declared outside of method scope
> 2. visible everywhere inside a class
> 3. inherited by subclasses

And so why might one use static?

> Good way of having a common value represented amongst objects.

And the best way to declare a constant in java is:

```
final static double PI = 3.14;
```

1.6.0.19. Accessing static methods

Here is what is right and what is wrong to do:

```
class A{
    void m1(){
        ...
    }

    static void m2(){
        ...
    }
}

class B{
    main(){
        A v = new A();
        v.m1(); // OK
        v.m2(); // OK
        A.m1(); // not OK
        A.m2(); // OK
    }
}
```

1.6.0.20. Restrictions on static methods

In general, a static method should never have to call an instance variable or instance method. This is because the compiler is never sure that the *this* object exists upon execution.

Similarly, whenever a class has only static methods and classes, it is completely unnecessary to instantiate an object.

1.6.0.21. Making sense of System.out.println

System is a predefined class in java. *Out* is a static variable defined inside System and println is a method of out.

1.6.0.22. Boxing, auto-boxing and unboxing

Consider the below:

```
main(){
    int i = 5;
    Integer ii = new Integer(5); // Boxing
    Integer jj = i; // Auto-boxing
    int j = jj.intValue(); // Unboxing
}
```

1.6.0.23. Interfaces

An interface is used as a 'contract' where every object that *implements* an interface must declare the methods found in an interface.

Every object of type interface must be a class implementing the interface.

We have that a class may extend only one class yet implement multiple interfaces.

> Contents of interface:
>
> 1. final static variable
> 2. abstract method

A novelty of Java 8 is that interfaces may also have default methods. Default methods simply also provide the implementation of the method. The need for this is that it removes the burden of having to define a method in a class implementing the interface. It is also possible to override a default method inside an interface extending another interface.

It is likely that a conflict will occur when some parent class implements one interface and the child another containing two different implementations of a default method. To ensure that this code is compiled, the class must resolve this conflict.

Finally, we note that an interface models a 'has-a' relationship.

1.6.0.24. Enumerations

An enum is simply a list of constants. We may for instance use an enum to model keyboard movements in a game:

```
enum Keyboard{
    LEFT,RIGHT,UP,DOWN;
}
```

And a more elaborate usage of enum:

```
enum Mobile{
    APPLE(90), SAMSUNG, HTC; //enum variables are by default final static

    int price;

    Mobile(){
        this.price = 80;
        print("i was called"); // will get called 3 times.
    }

    Mobile(int price){
        this.price = price; // constructs apple with 90.
    }
}
```

We note that for an enum, the *.values* method returns an array with each variable at the ordered index.

1.6.0.25. Why declare a reference variable of type interface

This is very common practice. For instance it is far better to say *List l1 = new ArrayList();* than to say *ArrayList l1 = new ArrayList();*

1.7. Week 11

Nested classes

The general way about declaring a nested class is:

```
public class A{
    private class B{
        ...
        // noting that B has access to all in A and A has access to all in B.
    }
```

```
}
```

Similarly access to method follows as:

```
public class A{

    void display(){
        ...
    }
    private class B{
        void display(){
            ...
        }
    }

    void main(){
        A.this.display(); //will call A's display
        this.display(); //will call B's display
    }
}
```

We note that if an inner class is declared as static, it will have access to only static members of the outer class.

Flaws of encapsulation

An object is said to be mutable if it has any public method that let it modify its content. To prevent such flaws, one might define getters that return a copy of the object. When returning a copy of an object via a getter, one must be very careful to create new object, that is:

```
class A{
    public getTime(){
        public int timecopy;
        return timecopy = this.actualTime; // this is wrong!!!
        return timecopy = new time(actualTime) // this is right!!!
    }
}
```

Here is how we define a copy constructor as used above:

```
class A {

    public A(A obj){
        this.p1 = obj.p1;
        this.p2 = obj.p2;
        ...
```

```
    }
}
```

> However we add the subtle note that the copy constructor is not polymorphic. That is suppose we have a parent class *A* and its child *B* with both copy constructors of their own. Now we say something like:

```
A copyB = new B(B);
```

That is we try to create a copy of some upcasted object and have copyB refer to it. This will produce an error as copyB is declared of type A hence will not have access to the copy constructor B. Similarly we may use the copy constructor of A but then it will not be a copy of B we create.

Week 11

Exception handling

The goal of exception handling is to prevent runtime errors that may surpass the compilation stage. Here is the syntax:

```
throw // indicates error
try // indicates that a block may possibly fail at runtime
catch // intercept errors
finally // indicates what to do after error handling
```

The idea is that throw will return an exception object for instance 'new Exception("there was an error")'

Throwable class

Throwable is the elementary error class, its constructors being:

```
public Throwable()
public Throwable(String message)
```

Child classes of Throwable

> Error Exception RuntimeException

The exception handling mechanism

The most general form is this:

```
try{
    k = i/j
```

```
}
catch(Exception e){
    ...print(e)
}
```

The above works because the try block creates an error which is caught by the most general exception type 'Exception'.

Using 'finally'

We may choose to use 'finally' which will run the piece of code inside a 'finally' block after the exception launched by try has ended. This code runs regardless of the outcome of the try/catch block. Here's an example:

```
    public static void main (String[] args) {
        try {
            //this is called the
        Integer a = Integer.valueOf(args[0]);
        int b = a.intValue();
        int c = 100/b;
        System.out.println("Inverse * 100 = " + c);
        }
        catch (NumberFormatException e1) {
        System.out.println("Il faut un nombre entier!");
        }
        catch (ArithmeticException e2) {
        System.out.println ("Parti vers l'infini!");
        }
        finally {
        System.out.println("on passe par le bloc final");
        }
}
```

Checked and unchecked errors

Checked errors are those that are exceptional cases a program may run into. We mark these as suspicious and as best practice put them inside a try/catch block. For instance the following is an unchecked exception:

```
public class DivideByZero {
    int dividerOfZeros(int notZero) {
        return notZero / 0;
    }
    public static void main(String[] args) {
        DivideByZero dbz = new DivideByZero();
        dbz.dividerOfZeros(10);
    }
}
```

Self defined exceptions

We may define our own exception in which case it must extend the Exception class. In addition, if we want our try block to include self-defined exceptions as below, we use *throw*.

```
try{
    int i = 1
    int j = 2;
    if(i+j < 5){
        throw(new OutOfBounds);
    }
}
catch(Exception e){
    ...print(e);
}
```

Syntax of throws

We must use *throws* whenever a method throws an exception without treating it, but wherever this method is called, it must be treated.

Assertions

Only use this for debugging purposes.