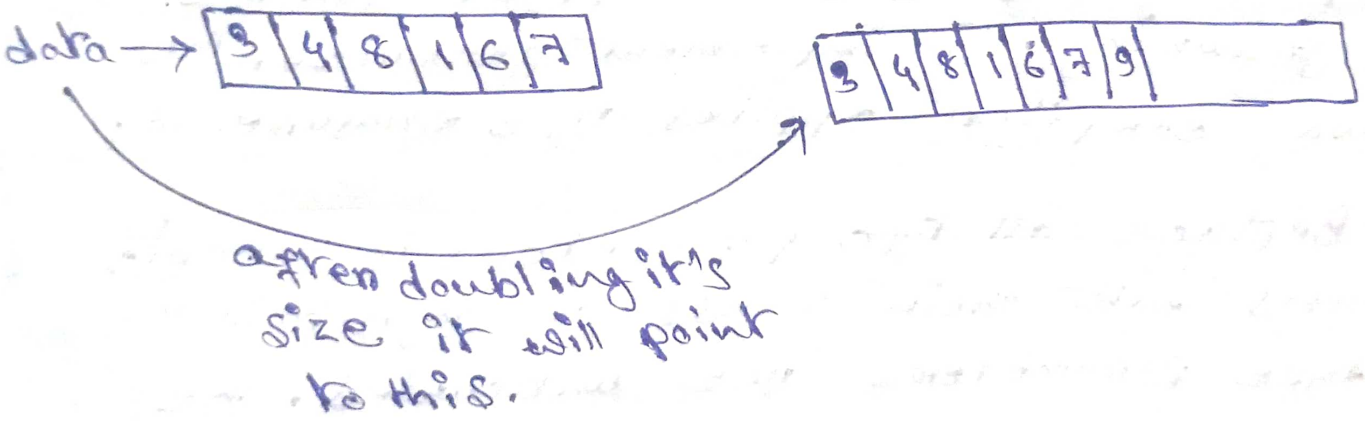
Object-Oriented-Programming #LEC-6

▶ Generics:

- Something about Arraylist: Suppose you have a Array full of data, basically the Array is full, Now if you want to add any data in that Array, the Array will double it's size and then it will copy the previous data into the new Array



- The problem with Arraylist By default it storing integer type values
- Generic helps us parameterize type of values providing
- Generics helps us ~~to put~~ what type of data to put in our custom classes. So, instead of making a custom Arraylist

of integers (which is by default), now we can make custom ArrayList of any type that we want

- Incase of ArrayList. we can only add

```
ArrayList<Integer> list2 = new ArrayList<>();
```

we can only add classes over here, and can't add primitives like: int or something like that

- Type Erasure:

Generics were introduced to Java language to provide tighter check at compile time. and to support generic programming. To implement generics, the Java compiler applies type erasure to:

- Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded. The produced bytecode, therefore. contains only ordinary classes, interfaces and methods.
- Insert type cast if necessary to preserve type safety.
- Generate bridge methods to preserve polymorphism in extended generic types

- **Can not declare static fields whose types are type parameters :**

A class static field is a class-level variable shared by all non-static objects of the class. Hence static fields of type parameters are not allowed. Consider the following class:

```
public class MobileDevice<T> {
    private static T os;
    // ...
}
```

if static fields of type parameters were allowed, then the following code would be confused:

```
MobileDevice<Smartphone> phone = new MobileDevice<>();
MobileDevice<Pager> pager = new MobileDevice<>();
MobileDevice<TabletPC> pc = new MobileDevice<>();
```

Because the static field os is shared by phone, pager and pc, what is the actual type of os? It cannot be Smartphone, Pager and TabletPC at the same time. You cannot therefore create static field of type parameters.

- **Cannot use casts or instanceof with Parameterized Types:**

Because the Java compiler erases all type parameters in generic code, you cannot verify which parameterized type for a generic type is being used at runtime:

```
public static <E> void rtti(List<E> list) {
if (list instanceof ArrayList<Integer>) {
    // compile-time error.
    // ...
}
}
```

- **Cannot create Arrays of Parameterized Types:**

we can not create arrays of parameterized types. For example, the following code does not compile:

```
List<Integer>[] arrayOfLists = new List<Integer>[2];
```

- **Cannot overload a method where the Formal Parameter Types of Each Overload Erase to the Same Raw Type:**

A class can not have two overloaded methods that will have the same signature after type erasure.

```
public class Example {
    public void print(Set<String> strSet) { }
    public void print(Set<Integer> intSet) { }
}
```

The overloads would all share the same classfile representation and will generate a compile-time error

- **Upper Bounded Wildcards in Java :**

The Upper Bounded Wildcards section shows that an upper bounded wildcard restricts the unknown type to be a specific type or a subtype of that type and is represented using the 'extends' keyword.

In a similar way, a lower bounded wildcard restricts the unknown type to be a specific type or a super type of that type.

example :

```
public class WildCardExample<T extends Number> {
    ... some code
}
```

here, we extends T to Number class, as, we want the type to be Number only like: float, String and all.

So, we can add the all the classes and subclasses of Number. means here T

should be Number of it's subclasses

- **Comparison Objects:** When we compare two objects Java gets confused which values of object to compare, As an object can contain different type of multiple values. For example if we make objects like this ↓

```
public class Student {
  int rollno;
  float marks;

public Student(int rollno, float marks){
    this.rollno = rollno;
    this.marks = marks;          }

public static void main(String[] arg){
Student Bigoy = new Student(85, 78.95f);
Student Manash = new Student(73, 88.35f);
// Now if we do
 if(Bigoy < Manash){
System.out.println("Manash has more marks");
                  }              }
                 }
```

As, we have compare two objects with variable name, Bigoy & Manash Java will get confused which value of the Objects to compare.

In order to solve this problem we have to implement comparable like this ↓

```
public class Student Comparable<> {
    int roll no;
    float marks;
    ... some code
}
```

```
public class Student implements Comparable<Student>{
    int roll no;
    float marks;
    ... some code
}
```

The conventional way to compare two object is, implements Comparable and used compareTo method.

```
public class Student implements Comparable<Student> {
    int rollno;
    float marks;
    public Student(int rollno, float marks) {
        this.rollno = rollno;
        this.marks = marks;
    }
    public int compareTo(Student o) {
        int diff = (int)(this.marks - o.marks);
        return diff;
    }
}
```

Main function

```
public class Main{
  public static void main(String[] args){
Student Bijoy = new Student(85, 78.35f);
Student Manash = new Student(73, 88.75f);
  if(Bijoy.compareTo(Manash)<0){
System.out.println("Manash has more marks");
                                  }  }
                                }
```

- Whey we do

```
A.compareTo(B)
```

If this thing gives result

i) = 0 mean, A and B equal

ii) > 0 means, A is Bigger than B

iii) < 0 means, B is Bigger than A or A is less than B

- Lambda functions: A lambda in Java essentially consists of three parts: a parenthesized set of parameters, an arrow and then a body, which can either be a single expression or a block of Java code.

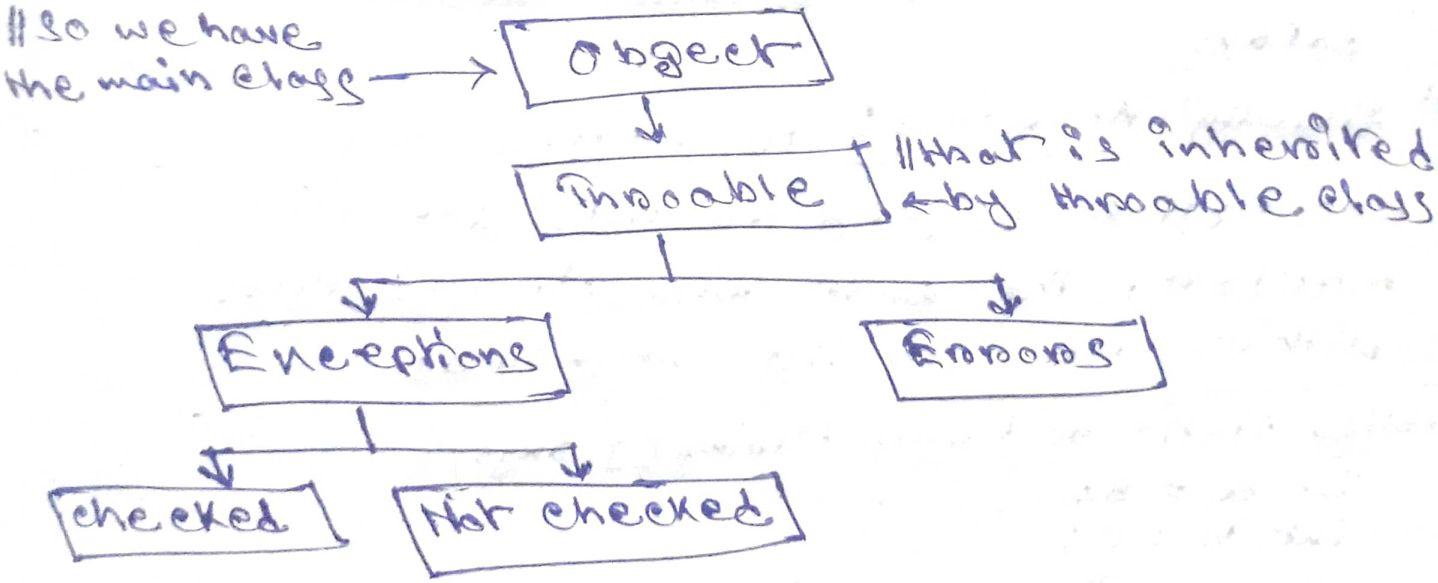
Exception Handling :-

- Difference between error and exception

> error is basically non recoverable, you can not recover from error and program can not handle it, like you can not write code inorder to handle it

exception on the other hand something that prevents normal flow of the program, something like dividing something by zero. that's arithmetic exception you can't divide by zero

In Java there is class that handles all the exception and all, that is known as the throwable class.



> checked exceptions are the ones that being checked during the compile time, if at compile time you are getting

some errors, an exception it will be a checked exception, while the code is compiling

- Unchecked exception on the otherhand is something that the compiler will not be able to detect, that will only be detected while program is running

Exception handling keywords:

- 'try', 'catch', 'finally' keyword:
  - 'try' keyword: It says try an execute the code within the block
  - 'catch' keyword: catch keyword says that if catch something some exception or something like that run the block of code within the catch
  - 'finally' keyword: No matter any exception happen or not, doesn't matters, the code will always run within the finally keyword.

```
public static void main (String [] args) {
    int a = 5;
    int b = 0;
    try { divide (a, b)';}
    Catch (ArithmeticException e){
    System.out.println(e.getMessage());}
    finally {System.out.println("This will always execute");}
                                                    }
```

- 'throw' and 'throws' keyword:

'throw' → throw means we are throwing exception

'throws' → used to declare exceptions

```
static int divide(int a, int b) throws ArithmeticException
{ if (b == 0) {
throw new ArithmeticException("please donot
                    divide by zero");
                    }
    return a/b;
}
```

- These are the five Exception handling keywords:

try, catch, finally, throw, throws.

- you can create your own custom Exception too
- 'finally' is option you can keep it or not, and also there could be only one 'finally' keyword.

- Object cloning: clone is actually a method in the Object class that can be used to make this copies we must implement that whos clone we want to create
In Object cloning we ~~are~~ create exact

copy of object

- **Shallow and Deep copy:**
  - **Understanding shallow copy:** Suppose you have object with name ..

Bigoy = [age = 56 ;
name = "Bigoy",
ann = [3, 4, 5, 6, 9, 1]]

Now if we have create a new object out of Bigoy using using cloning method or something

twin = [age = 56 ;
name = it will point to the string in the actual or parent object
ann = it will point to ann in parent object]

So, this doesn't define actual copy of the object, so this is shallow copy, where premitives will be copy of it is and in case of non premitive value, like the reference variable will point to the same object. or non-premitive that is present in the main object

change through the copy is resulting change in the actual object. This is known as shallow copy.

In order to solve this problem, we can do something called deep copy.

- **Deep copy:** If object itself contain some other object, while creating a copy don't pointed to that only, actually create a copy of that object itself.

Bijoy = [age = 56;
name = "Bijoy",
arr. = [3, 4, 5, 6, 9, 1]]

while cloning the above method, it will not point to the same object, but will create a copy of it.

twin = [age = 56;
name = "Bijoy",
arr = [3, 4, 5, 6, 9, 1]]