```
public class test1
    public static void main (String[]args)1
        Fun f = (int x) → System.out.println (2*x);
        f.~~RunM~~abstractfun(5);
    }
}
```

Output

10

- Exception Handling

Exception Handling is one of the effective means to handle runtime errors so that the regular flow of the application can be preserved.

TYPES OF EXCEPTIONS	
USER DEFINED EXCEPTION	BUILT IN EXCEPTION
CHECKED EXCEPTIONS (Checked at compile-time)	UNCHECKED EXCEPTIONS (Not checked)
• ClassNotFoundException	• ArithmeticException
• InterruptedException	• ClassCastException
• IOException	• NullPointerException
• InstantiationException	• ArrayIndexOutOfBoundsEx
• SqlException	• ArrayStoreException
• FileNotFoundException	• IllegalThreadStateExcepti

Example

```
public class test {
    public static void main(String[] args) {
        try {
            int data = 100/0;
        }
        catch (ArithmeticException e) {
            System.out.println(e);
        }
        finally {
            System.out.println("finally Block");
        }
    }
}
```

Output

java.lang.ArithmeticException : / by zero
finally Block.

```
public class Main {
    public static void main(String[] args) {
        int a = 5;
        int b = 0;
        try {
            divide(a, b);
        } catch (ArithmeticException e) {
            System.out.println(e.getMessage());
        }
    }
    static void divide(int a, int b) throws ArithmeticExcep-
    -tion {
        if (b == 0)
            throw new ArithmeticException("Wrong");
    }
}
```

Output

Wrong

```
class MyException extends exception {
    public Myexception (String message) {
        super(message);
    }
}
public class Main {
    public static void main(String[]args) {
        try {
            throw new MyException ("This is Custom Excep
        } catch (MyException e) {
            System.out.println("Caught Exception:" +
                e.getMessage());
        }
    }
}
```

Output

Caught Myexception: This is my custom exception.

- **Try-Catch Block**

  The try block contains code that might throw exception

  The catch block handles specific exceptions

- **Finally Block**

  The finally block contains code that will always execute, regardless of whether an exception is thrown or not

- **Throwing exception**

  The throw keyword is used to throw a custom exception.

. Object cloning

Object cloning is the process of creating an exact copy (clone) of an existing object.

Shallow cloning

It creates a new instance and copies all the fields of the object to that new instance where both are referencing to the same memory in heap memory.

```
class clone {
    public static void main(String[] args) throws CloneNotSuppo-
    -rted Exception {
        Human main = new Human(19, "Main");
        Human twin = (Human) main.clone();
        System.out.println(twin.age + " " + twin.name);
        System.out.println(Arrays.toString(twin.arr));
        twin.arr[0] = 100;
        System.out.println("Twin :" + Arrays.toString(twin.arr));
        System.out.println("Main :" + Arrays.toString(main.arr));
        twin.name = "Twin";
        System.out.println("Twin: " + twin.name);
        System.out.println("Main: " + main.name);
    }
}

class Human implements Cloneable {
    int age;
    String name;
    int[] arr;
    Human(int age, String name) {
        this.age = age;
        this.name = name;
    }
```

```
@override
public Object clone() throws CloneNotFoundException{
        return super.clone();
    }
}
```

Output:

19 Main

[1, 2, 3, 4]

Twin : [100, 2, 3, 4]

Main : [100, 2, 3, 4]

Twin : Twin

Main : Main

- **Deep cloning**

Deep cloning is the process of creating exactly the independent duplicate objects in the heap memory and manually assigning the values of the second object where values are supposed to be copied.

```
class clone {
    public static void main(String[] args) throws
        CloneNotSupportedException {
            Human main = new Human(19, "Main");
            Human twin = (Human) main.clone();
            System.out.println(Arrays.toString(arr));
            twin.arr[0]: 100;
            System.out.println(Arrays.toString(twin.arr));
            System.out.println(Arrays.toString(main.arr));
    }
}
```

```
class Human implements Cloneable {
    int age;
    String name;
    int[] arr;
    Human (int age, String name) {
        this.age = age;
        this.name = name;
    }
@Override
public Object clone() throws CloneNotSupportedException {
    Human twin = (Human) super.clone();
    twin.arr = new int[twin.arr.length];
    for(int i=0; i<a twin.arr.length; i++) {
        twin.arr[i] = this.arr[i];
    }
    return twin;
  }
}
```

Output

[1, 2, 3, 4]

[100, 2, 3, 4]

[1, 2, 3, 4]

Enums

An enum is a special class that represents a group of constants (unchangeable variables like final variable)

Note: The constants should be in uppercase letters

Example

```
enum Level {
    LOW,
    MEDIUM,
    HIGTH
}
```

Level variableName = Level.MEDIUM;

- Vector class

The Vector class is a part of Java Collection framework and is used to create dynamic arrays.

Thread-Safe

Unlike ArrayList, Vector is synchronized which means its thread-safe. This means that multiple thread can access a vector ~~but with~~ object concurrently without any thread interference.

Performance

Because of its synchronization, Vector can be slower than ArrayList in single threaded application. However in multi-threaded application, Vector can be more efficent because it ensures thread safety.

legacy class

Vector is a legacy class and it is recommended to use ArrayList or LinkedList instead as they provide better performance in most cases

Example

```
public class Main {
    public static void main(String[] args) {
        Vector<Integer> vc = new Vector<>();
        vc.addElement(1); //Add elements to vector.
        vc.addElement(2);
        //Access elements in the vector
        System.out.println("Vector: " + vc.elementAt(0));
        //Remove elements in the vector.
        vc.removeElement(2);
    }
}
```