/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Every Ruls of objective condition of chess game.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Current Rules Have not Attack Movements\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Ramin Edjlal\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Current Rules Have Not 'Check' And 'CheckMate' \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+\*)QC-OK.

\* Elephant Rules Hardly\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Horse Rules Hardly\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Minister Rules Hardly\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* King Rules Hardly\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Castles Rules Hardly\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Restricted has been solved\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*--\*\*(-)

\* No movements greater than one. Some Abnormal Movements\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(++)

\* Abnormal Movements Correction\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*--\*\*(-)

\* Clear Dirty Part\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*--\*\*(-)

\* Chess Rules Soldier Not Moved Jump From Enemy to 2\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Chess Rules Abnormally Minister Gray Elephant to Right\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Chess Rules Elephant Normally\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*--\*\*(-)

\* Abnormally Recursive Method\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Chess Rule Check CheckMate Doesn’t Work\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Clicking 'Table' Content Has been Abnormally\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* The Mechanism of Check Declared and Act 'Not' Logically\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* The Mechanism of Table Assignments and the Virtualization Misleading\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* The Movements of horse Brown 'Alice' Left Side Cause to Mislead\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* ExistInDestinationEnemy Thinking Misleading Operations\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Null Thinking Exception Handling Should be Configured\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Malfunction of Mouse 'Bob' Event Handling For Movements\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Non 'Check' Second Rules 'Alice' Move to 'Check' State\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* 'CheckMate' Not Recognized By 'Alice'.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* 'Check' Recognized From 'Hard' Game. CheckMate Have Not Been Identified.\*\*\*\*\*\*\*\*\*\*\*\*RSPB(+\*)

\* Chess Rules MalFunctional\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Unsatisfied CheckMate By 'Bob' With 'Alice'\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Removable 'Check' by 'Bob' Was Not done by 'Alice' \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Unknown 'CheckRemovable' and Unknown 'Check' Mechanism\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Table Content at 'Bob' 'Check' of 'Alice', Malfunction with 'horse'\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Can Hit 'King'\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Gone to 'Check' State Deterministic\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* King Killer. Gone to ObjectDanger State by 'Alice' and 'Bob'\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RSRS(++)

\* King Killer By 'Alice' and Gone to ObjectDanger Remaining\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Killer Check Solved by Changing Strategy. Check by 'Alice' Cannot Been Removed.RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(-+)

\* Castle King Mechanism Failed\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Arguments IgonoreTowEnemy Between King and Attaker in Check ObjectDanger Misleading\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* 'Check' Ignore. Un Rulement 'Bob Movements\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RSRS(++)

\* Unidentified 'Bob' Minister Movements in Check and Unrulements Movements\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Tow King Beside Them\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* King of 'Bob' Gone to ObjectDanger.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Gone to Check by 'Bob'\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Chess Order and Chess Check by Bob Malfunctioned\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* 'CheckMate' of 'Alice' Ended by Moving of 'Bob' King Unrulments\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Movements of 'Alice' Soldier to Backward.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* BrigdeKing Movements in Large Castle King Misleading\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(\_)

\* Syntax Statements Failed By Halting.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Check Of Bob Misleading no reason.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Syntax Error At Genetic Algorithm By Bob.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* 1394/12/20\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+:Sum(48)) (\_ :Sum(1)) (-:Sum(5)) (\*:Sum(2))

\* Chess Syntax MalFunction.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*[+]

\* Chess Rules Non Soldier Colud Not been Detected. For AStarGreedy Fist Algorithm.\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*{+}

\* 'Check' Released isolatly.'Check' of Brown (Alice) No Matched Realesed.\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*<+>

\* 'Check' Not Detected By Bob.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*<+>

\* Bob Cloud not Remove 'Check'.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*<+>

\* Bob Colud not Move.No Check asnd CheckMate.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*<+>

\* Kings Have been Realeased Attacked.By Alice and Bob.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*<+>

\* ObjectDanger kings Not work!\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*CU\*\*\*\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*<\*>QC-OK.

\* Chess Rules of Movments AStarGreedy First caused to Databse MalFunction.\*\*\*\*\*\*\*\*\*\*\*\*\*CU\*\*\*\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(\*)QC-OK.

\* Mal Function of Table.Table zero.!\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Timer of Bob and Alice do not works!\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*CU\*\*\*\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(\*)QC-OK.

\* Not Right of Penalty Regard Mechansim.Misleading of Operations.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*(+)

\* Reveal From CheckMate By Alice MalFunction.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*{+}

\* CheckMate Not Work On Statistic and More By Alice.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*{+}

\* CheckMate Operation By Alice is MalFunction.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*CU\*\*\*\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(\*)QC-OK.

\* 'Minister' Alice Have been Check unreasonably.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*CU\*\*\*\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(\*)QC-OK.

\* 'Alice' Supposed Wrongly CheckObjectDangour Means Check.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*CU\*\*\*\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(\*)QC-OK.

\* Illegal King Foundation in Rules Function No Reasonaly.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*<+>

\* Brown (Alice) King Dosn't exist.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*\*\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*<+>

\* Mal Function of Castles King User Determination.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*CU\*\*\*\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*{\*}QC-OK

\* Mal Function of Check int Detection at ObjectDangouring.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*CU\*\*\*\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*<\*>QC\_OK

\* Assignment of Check State at ObjectDangourCheckRemove Method Not Occured.\*\*\*\*\*\*\*\*\*\*\*\*\*\*CU\*\*\*\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*<\*>QC\_OK.

\* Table Incredible Content Leads to Undisirable Result in Check and ....\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*(+)

\* CheckedMate Complexity Over Numbers Of Cores Compexity.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*CU\*\*\*\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*{\*}QC-OK.

\* Alice Castles King Statistic is misleading.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*.CU\*\*\*\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*{\*}Qc\_OK

\* Sodiers of Alice get to go backward and non Existining Enemy of Killer sometimes.CU\*\*\*\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*{\*}Qc\_Bad.

\* Pat Mechanisam Dosn't act Misleading.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*.CU\*\*\*\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*{\*}QC\_OK.

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+):Sum(1)) 4:(+:Sum(5)) 5.(\*:Sum(1)) 6.(+:Sum(2)) (\*:Sum(2)) 7.(+:Sum(2)) 8.(\*:Sum(3)) 9.(QC-OK.:Sum(7))

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Drawing;

using System.IO;

namespace RefrigtzDLL

{

[Serializable]

public class ChessRules

{

public bool IgnoreSelfObject = false;

public static int ObjectHittedRow = -1;

public static bool SelfHomeStatCP = false;

public static int ObjectHittedColumn = -1;

//Inititae Global Variables.

public bool MovementsAStarGreedyHuristicFoundT = false;

public bool IgnoreSelfObjectsT = false;

public bool UsePenaltyRegardMechnisamT = true;

public bool BestMovmentsT = false;

public bool PredictHuristicT = true;

public bool OnlySelfT = false;

public bool AStarGreedyHuristicT = false;

static int NumbersofKingMovesToPatGray = 0;

static int NumbersofKingMovesToPatBrown = 0;

public static bool PatCheckedInKingRule = false;

public static bool CastleKingAllowedGray = true;

public static bool CastleKingAllowedBrown = true;

public static bool KingAttacker = false;

public static bool SmallKingCastleBrown = false;

public static bool KingCastleBrown = false;

public static bool SmallKingCastleGray = false;

public static bool KingCastleGray = false;

public static bool BigKingCastleBrown = false;

public static bool BigKingCastleGray = false;

public static bool CheckObjectDangourIgnoreSelfThingBetweenTowEnemyKingHaveSupporter = false;

public static int CheckObjectDangourIgnoreSelfThingBetweenTowEnemyKingHaveSupporterNumber = 0;

public static bool CheckObjectDangourIgnoreSelfThingBetweenTowEnemyKing = false;

public bool CheckGrayObjectDangour = false;

public bool CheckBrownObjectDangour = false;

public static bool CheckGrayObjectDangourFirstTimesOcured = false;

public static bool CheckBrownObjectDangourFirstTimesOcured = false;

public static bool CastleActGray = false;

public static bool CastleActBrown = false;

public static int CurrentOrder = 1;

public bool PatkGray = false;

public bool PatBrown = false;

public bool CheckGray = false;

public bool CheckBrown = false;

public bool CheckMateGray = false;

public bool CheckMateBrown = false;

public static bool CheckGrayRemovable = true;

public static bool CheckBrownRemovable = true;

public static int CheckGrayRemovableValueRowi = 0;

public static int CheckGrayRemovableValueColumni = 0;

public static int CheckGrayRemovableValueRowii = 0;

public static int CheckGrayRemovableValueColumnjj = 0;

public static int CheckBrownRemovableValueRowi = 0;

public static int CheckBrownRemovableValueColumnj = 0;

public static int CheckBrownRemovableValueRowii = 0;

public static int CheckBrownRemovableValueColumnjj = 0;

int Kind;

int KindNA;

int Row, Column;

int[,] Table = new int[8, 8];

static int Order = 0;

//public bool ExistInDestinationEnemy = false;

bool ArrangmentsBoard = false;

int CurrentAStarGredyMax = -1;

static void Log(Exception ex)

{

try

{

Object a = new Object();

lock (a)

{

string stackTrace = ex.ToString();

File.AppendAllText(AllDraw.Root + "\\ErrorProgramRun.txt", stackTrace + ": On" + DateTime.Now.ToString()); // path of file where stack trace will be stored.

}

}

catch (Exception t) { Log(t); }

}

public ChessRules(int CurrentAStarGredy, bool MovementsAStarGreedyHuristicTFou, bool IgnoreSelfObject, bool UsePenaltyRegardMechnisa, bool BestMovment, bool PredictHurist, bool OnlySel, bool AStarGreedyHuris, bool ArrangmentsChanged, int oRDER)

{

CurrentAStarGredyMax = CurrentAStarGredy;

MovementsAStarGreedyHuristicFoundT = MovementsAStarGreedyHuristicTFou;

IgnoreSelfObjectsT = IgnoreSelfObject;

UsePenaltyRegardMechnisamT = UsePenaltyRegardMechnisa;

BestMovmentsT = BestMovment;

PredictHuristicT = PredictHurist;

OnlySelfT = OnlySel;

AStarGreedyHuristicT = AStarGreedyHuris;

Order = oRDER;

ArrangmentsBoard = ArrangmentsChanged;

}

public ChessRules(int CurrentAStarGredy, int oRDER, bool MovementsAStarGreedyHuristicTFou, bool IgnoreSelfObject, bool UsePenaltyRegardMechnisa, bool BestMovment, bool PredictHurist, bool OnlySel, bool AStarGreedyHuris, bool ArrangmentsChanged)

{

CurrentAStarGredyMax = CurrentAStarGredy;

Order = oRDER;

MovementsAStarGreedyHuristicFoundT = MovementsAStarGreedyHuristicTFou;

IgnoreSelfObjectsT = IgnoreSelfObject;

UsePenaltyRegardMechnisamT = UsePenaltyRegardMechnisa;

BestMovmentsT = BestMovment;

PredictHuristicT = PredictHurist;

OnlySelfT = OnlySel;

AStarGreedyHuristicT = AStarGreedyHuris;

ArrangmentsBoard = ArrangmentsChanged;

}

//Constructor

public ChessRules(int CurrentAStarGredy, bool MovementsAStarGreedyHuristicTFou, bool IgnoreSelfObject, bool UsePenaltyRegardMechnisa, bool BestMovment, bool PredictHurist, bool OnlySel, bool AStarGreedyHuris, bool ArrangmentsChanged, int Ki, int[,] A, int Ord, int i, int j)

{

CurrentAStarGredyMax = CurrentAStarGredy;

MovementsAStarGreedyHuristicFoundT = MovementsAStarGreedyHuristicTFou;

IgnoreSelfObjectsT = IgnoreSelfObject;

UsePenaltyRegardMechnisamT = UsePenaltyRegardMechnisa;

BestMovmentsT = BestMovment;

PredictHuristicT = PredictHurist;

OnlySelfT = OnlySel;

AStarGreedyHuristicT = AStarGreedyHuris;

ArrangmentsBoard = ArrangmentsChanged;

Row = i;

Column = j;

//Initiate Global Variables By Local Parameters.

KindNA = Ki;

Kind = System.Math.Abs(Ki);

Table = new int[8, 8];

for (int ik = 0; ik < 8; ik++)

for (int jk = 0; jk < 8; jk++)

Table[ik, jk] = A[ik, jk];

Order = Ord;

}

//Initiate of Rules of Chess Refregitz.

public bool Rules(int RowFirst, //The First Click Row

int ColumnFirst, //The First Click Column.

int RowSecond, //The Destination Click Row

int ColumnSecond, //The Destination Click Column

Color color,//int.

int Ki//Current Kind.

, bool SelfHomeStatCP = true

)

{

Object O = new Object();

lock (O)

{

if (!SelfHomeStatCP)

IgnoreSelfObject = true;

else

IgnoreSelfObject = false;

}

//Initaite Global Varibales.

Object O1 = new Object();

lock (O1)

{

CheckObjectDangourIgnoreSelfThingBetweenTowEnemyKingHaveSupporter = false;

CheckObjectDangourIgnoreSelfThingBetweenTowEnemyKingHaveSupporterNumber = 0;

}

//When Order is Non Detectable Continue Traversal Back.

//if (Order != CurrentOrder)

// return false;

//Found Location of Tow Gray and Brown Kings.

int RowB = 0, ColumnB = 0;

int RowG = 0, ColumnG = 0;

FindBrownKing(Table, ref RowB, ref ColumnB);

FindGrayKing(Table, ref RowG, ref ColumnG);

//Gray Order.

if ((Order == 1))

{

if (Table[RowFirst, ColumnFirst] == 6)

{

if (System.Math.Abs(RowB - RowSecond) <= 1 && System.Math.Abs(ColumnB - ColumnSecond) <= 1)

return false;

}

//Illegal King Foundation.

if (System.Math.Abs(RowB - RowG) <= 1 && System.Math.Abs(ColumnB - ColumnG) <= 1)

return false;

}//Brown Order.

else

{

if (Table[RowFirst, ColumnFirst] == -6)

{

if (System.Math.Abs(RowG - RowSecond) <= 1 && System.Math.Abs(ColumnG - ColumnSecond) <= 1)

return false;

}

//Ilegal Kings Foundation.

if (System.Math.Abs(RowB - RowG) <= 1 && System.Math.Abs(ColumnB - ColumnG) <= 1)

return false;

}

//Determination of Enemy in the Destionation Home.

bool ExistInDestinationEnemy = new bool();

if (((Table[RowFirst, ColumnFirst] > 0) && (Table[RowSecond, ColumnSecond] < 0) && (Order == 1)))

{

ExistInDestinationEnemy = true;

}

else

if (((Table[RowFirst, ColumnFirst] < 0) && (Table[RowSecond, ColumnSecond] > 0) && (Order == -1)))

{

ExistInDestinationEnemy = true;

}

//If There is A Source of Soldier.

if (System.Math.Abs(Kind) == 1)

{

if (!(ArrangmentsBoard))

{

//Solders of Gray at Begining.

if (ColumnFirst == 1 && (Order == 1))

return Rule(RowFirst, ColumnFirst, RowSecond, ColumnSecond, true, color, ExistInDestinationEnemy, Ki, SelfHomeStatCP);

else//Solder of Brown At Begining.

if (ColumnFirst == 6 && (Order == -1))

return Rule(RowFirst, ColumnFirst, RowSecond, ColumnSecond, true, color, ExistInDestinationEnemy, Ki, SelfHomeStatCP);

else//Another Solder Movments.

return Rule(RowFirst, ColumnFirst, RowSecond, ColumnSecond, false, color, ExistInDestinationEnemy, Ki, SelfHomeStatCP);

}

else

{

//Solders of Gray at Begining.

if (ColumnFirst == 6 && (Order == 1))

return Rule(RowFirst, ColumnFirst, RowSecond, ColumnSecond, true, color, ExistInDestinationEnemy, Ki, SelfHomeStatCP);

else//Solder of Brown At Begining.

if (ColumnFirst == 1 && (Order == -1))

return Rule(RowFirst, ColumnFirst, RowSecond, ColumnSecond, true, color, ExistInDestinationEnemy, Ki, SelfHomeStatCP);

else//Another Solder Movments.

return Rule(RowFirst, ColumnFirst, RowSecond, ColumnSecond, false, color, ExistInDestinationEnemy, Ki, SelfHomeStatCP);

}

}

else//For another Kind of Objects.

return Rule(RowFirst, ColumnFirst, RowSecond, ColumnSecond, false, color, ExistInDestinationEnemy, Ki, SelfHomeStatCP);

}

//Castle King Movment Consideration.

public bool CastleKing(int RowFirst, int ColumnFirst, int RowSecond, int ColumnSecond, bool NotMoved, Color color, int Ki)

{

if (!(ArrangmentsBoard))

{ //Gray Order.

if (Order == 1)

{

//When Gray Castles Not Act.

if (ChessRules.CastleKingAllowedGray)

{

//If Column is At First Location.

if (ColumnFirst == 0 && ColumnSecond == 0)

{

//When Kings Moves for Small Kings Castles Movments.

if (RowFirst == RowSecond - 2 && ((RowSecond - 2) >= 0))

{

//Consideration of Castles King of Gray King.

try

{

if (((RowSecond - 1) >= 0) && ((RowSecond + 1) < 8) && ((RowSecond - 2) >= 0) && Table[RowSecond - 2, ColumnSecond] == 6 && Table[RowSecond - 1, ColumnSecond] == 0 && Table[RowSecond, ColumnSecond] == 0 && Table[RowSecond + 1, ColumnSecond] == 4)

{

Object O = new Object();

lock (O)

{

CastleActGray = true;

SmallKingCastleGray = true;

}

return true;

}

}

catch (Exception t)

{

Log(t);

}

}

else//For Greates Castles King Movments.

if (RowFirst == RowSecond + 2 && ((RowSecond + 2) < 8))

{

//Consideration of Castles King M<ovments.

try

{

if (((RowSecond + 2) < 8) && ((RowSecond - 1) >= 0) && ((RowSecond + 1) < 8) && ((RowSecond - 2) >= 0) && Table[RowSecond + 2, ColumnSecond] == 6 && Table[RowSecond + 1, ColumnSecond] == 0 && Table[RowSecond, ColumnSecond] == 0 && Table[RowSecond - 1, ColumnSecond] == 0 && Table[RowSecond - 2, ColumnSecond] == 4)

{

Object O = new Object();

lock (O)

{

CastleActGray = true;

BigKingCastleGray = true;

}

return true;

}

}

catch (Exception t)

{

Log(t);

}

}

}

}

}

else//Order of Brown.

{

//When Brown Castles King Not Occured.

if (ChessRules.CastleKingAllowedBrown)

{

//Column Situation.

if (ColumnFirst == 7 && ColumnSecond == 7)

{

//Small Brown King Castles Consideration.

if (RowFirst == RowSecond - 2 && ((RowSecond - 2) < 8))

{

try

{

if (((RowSecond - 1) >= 0) && ((RowSecond + 1) < 8) && Table[RowSecond - 2, ColumnSecond] == -6 && Table[RowSecond - 1, ColumnSecond] == 0 && Table[RowSecond, ColumnSecond] == 0 && Table[RowSecond + 1, ColumnSecond] == -4)

{

//CastleActBrown = true;

Object O1 = new Object();

lock (O1)

{

SmallKingCastleBrown = true;

}

return true;

}

}

catch (Exception t)

{

Log(t);

}

}

else

if (RowFirst == RowSecond + 2 && ((RowSecond + 2) < 8))

//Brown Kings.Big King Castles Consideration.

{

try

{

if (((RowSecond + 2) < 8) && ((RowSecond - 1) >= 0) && ((RowSecond + 1) < 8) && ((RowSecond - 2) >= 0) && Table[RowSecond + 2, ColumnSecond] == -6 && Table[RowSecond + 1, ColumnSecond] == 0 && Table[RowSecond, ColumnSecond] == 0 && Table[RowSecond - 1, ColumnSecond] == 0 && Table[RowSecond - 2, ColumnSecond] == -4)

{

//CastleActBrown = true;

Object O = new Object();

lock (O)

{

BigKingCastleBrown = true;

}

return true;

}

}

catch (Exception t)

{

Log(t);

}

}

}

}

}

}

else

{

//Gray Order.

if (Order == 1)

{

//When Gray Castles Not Act.

if (ChessRules.CastleKingAllowedGray)

{

//If Column is At First Location.

if (ColumnFirst == 7 && ColumnSecond == 7)

{

//When Kings Moves for Small Kings Castles Movments.

if (RowFirst == RowSecond - 2 && ((RowSecond - 2) >= 0))

{

//Consideration of Castles King of Gray King.

try

{

if (((RowSecond - 2) >= 0) && ((RowSecond - 1) >= 0) && ((RowSecond + 1) < 8) && Table[RowSecond - 2, ColumnSecond] == 6 && Table[RowSecond - 1, ColumnSecond] == 0 && Table[RowSecond, ColumnSecond] == 0 && Table[RowSecond + 1, ColumnSecond] == 4)

{

//CastleActGray = true;

//SmallKingCastleGray = true;

return true;

}

}

catch (Exception t)

{

Log(t);

}

}

else//For Greates Castles King Movments.

if (RowFirst == RowSecond + 2 && ((RowSecond + 2) < 8))

{

//Consideration of Castles King M<ovments.

try

{

if (((RowSecond + 2) < 8) && ((RowSecond - 1) >= 0) && ((RowSecond + 1) < 8) && ((RowSecond - 2) >= 0) && Table[RowSecond + 2, ColumnSecond] == 6 && Table[RowSecond + 1, ColumnSecond] == 0 && Table[RowSecond, ColumnSecond] == 0 && Table[RowSecond - 1, ColumnSecond] == 0 && Table[RowSecond - 2, ColumnSecond] == 4)

{

//CastleActGray = true;

//BigKingCastleGray = true;

return true;

}

}

catch (Exception t)

{

Log(t);

}

}

}

}

}

else//Order of Brown.

{

//When Brown Castles King Not Occured.

if (ChessRules.CastleKingAllowedBrown)

{

//Column Situation.

if (ColumnFirst == 0 && ColumnSecond == 0)

{

//Small Brown King Castles Consideration.

if (RowFirst == RowSecond - 2 && ((RowSecond - 2) > 0))

{

try

{

if (((RowSecond - 2) >= 0) && ((RowSecond - 1) >= 0) && ((RowSecond + 1) < 8) && Table[RowSecond - 2, ColumnSecond] == -6 && Table[RowSecond - 1, ColumnSecond] == 0 && Table[RowSecond, ColumnSecond] == 0 && Table[RowSecond + 1, ColumnSecond] == -4)

{

//CastleActBrown = true;

//SmallKingCastleBrown = true;

return true;

}

}

catch (Exception t)

{

Log(t);

}

}

else

if (RowFirst == RowSecond + 2 && ((RowSecond + 2) < 8))

//Brown Kings.Big King Castles Consideration.

{

try

{

if (((RowSecond + 2) < 8) && ((RowSecond - 1) >= 0) && ((RowSecond + 1) < 8) && ((RowSecond - 2) >= 0) && Table[RowSecond + 2, ColumnSecond] == -6 && Table[RowSecond + 1, ColumnSecond] == 0 && Table[RowSecond, ColumnSecond] == 0 && Table[RowSecond - 1, ColumnSecond] == 0 && Table[RowSecond - 2, ColumnSecond] == -4)

{

// CastleActBrown = true;

//BigKingCastleBrown = true;

return true;

}

}

catch (Exception t)

{

Log(t);

}

}

}

}

}

}

return false;

}

//Simulation and Consdtruction of Check.

public bool CheckConstructor(Color color, int RowFirst, int ColumnFirst, int RowSecond, int ColumnSecond, int Ki, int Order)

{

//Initiate a Local Variable.

int[,] tab = new int[8, 8];

//Clone A Copy of Table.

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

{

tab[i, j] = Table[i, j];

}

//Act a Move.

tab[RowSecond, ColumnSecond] = tab[RowFirst, ColumnFirst];

tab[RowFirst, ColumnFirst] = 0;

//If There is Check State.

if (Check(tab, Order))

{

//When int of Order is Gray Check return Check State.

if (Order == 1)

if (CheckGray)

return true;

//When int is Brown State there is Check State return Check State.

if (Order == -1)

if (CheckBrown)

return true;

}

//Return Non Check State.

return false;

}

//Method of Self Home int Objects Consideration.

private bool ExistSelfHome(int RowFirst, int ColumnFirst, int RowSecond, int ColumnSecond, bool NotMoved, Color color, int Ki)

{

//Initiate of Local Variable.

bool NotExistInDestinationSelfHome = false;

//When There is Not Source and Destination is the Same Home Location.

if (RowFirst != RowSecond || ColumnFirst != ColumnSecond)

{

//If the Same Gray int Return Self Home.

if (Table[RowSecond, ColumnSecond] > 0 && Table[RowFirst, ColumnFirst] > 0)

NotExistInDestinationSelfHome = true;

else//If The Same int Brown Return Self Home.

if (Table[RowSecond, ColumnSecond] < 0 && Table[RowFirst, ColumnFirst] < 0)

NotExistInDestinationSelfHome = true;

}

return NotExistInDestinationSelfHome;

}

//ObjectDanger Consideration.

public bool ObjectDangourKingMove(int Order, int[,] Table, bool DoIgnore)

{

int[,] Tab = new int[8, 8];

//Clone a Copy

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

Tab[i, j] = Table[i, j];

//Initiate Variables.

CheckGray = false;

CheckBrown = false;

CheckGrayObjectDangour = false;

CheckBrownObjectDangour = false;

int RowG = 0, ColumnG = 0;

int RowB = 0, ColumnB = 0;

Object O = new Object();

lock (O)

{

if (DoIgnore)

ChessRules.CheckObjectDangourIgnoreSelfThingBetweenTowEnemyKing = true;

}

//Check identification.

Check(Tab, Order);

bool CheckGrayDummy = CheckGray;

bool CheckBrownDummy = CheckBrown;

//If There is Check on Tow Side.

if (CheckBrown || CheckGray)

{

//Check meand achmaz.

if (CheckBrown)

CheckBrownObjectDangour = true;

if (CheckGray)

CheckGrayObjectDangour = true;

return true;

}

int CDummy = ChessRules.CurrentOrder;

int COrder = Order;

if (Order == 1)

{

//Location of King Gary

if (FindGrayKing(Tab, ref RowG, ref ColumnG))

{

//For Enemy Brown.

for (int ii = 0; ii < 8; ii++)

{

for (int jj = 0; jj < 8; jj++)

{

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

Tab[i, j] = Table[i, j];

//Ignore Gray.

if (Tab[ii, jj] >= 0)

continue;

//For Current Gray and Empty.

for (int iii = 0; iii < 8; iii++)

{

for (int jjj = 0; jjj < 8; jjj++)

{

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

Tab[i, j] = Table[i, j];

//Ignore Brown.

if (Tab[iii, jjj] < 0)

continue;

ThinkingChess AA = new ThinkingChess(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsBoard, ii, jj);

//When There is Attacked to Gray from Brown.

if (AA.Attack(Tab, ii, jj, iii, jjj, Color.Brown, Order \* -1))

{

/\*

//Move.

int a = Tab[iii, jjj];

Tab[iii, jjj] = Tab[ii, jj];

Tab[ii, jj] = 0;

int[,] Tabl = new int[8, 8];

for (int h = 0; h < 8; h++)

for (int g = 0; g < 8; g++)

Tabl[h, g] = Tab[h, g];

ChessRules AAA = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsBoard, Tabl[iii, jjj], Tabl, Order, iii, jjj);

//When there is checked or checkmate.

if (AAA.Check(Tabl, Order))

{

CheckGrayObjectDangour = true;

break;

}

\*/

CheckGrayObjectDangour = true;

}

if (CheckGrayObjectDangour)

break;

}

if (CheckGrayObjectDangour)

break;

}

if (CheckGrayObjectDangour)

break;

}

if (CheckGrayObjectDangour)

break;

}

}

}

else

{

//Location of King Brown

if (FindBrownKing(Tab, ref RowB, ref ColumnB))

{

//For Gray Enemy.

for (int ii = 0; ii < 8; ii++)

{

for (int jj = 0; jj < 8; jj++)

{

//Ignore Brown

if (Tab[ii, jj] <= 0)

continue;

//For Current BNrown.

for (int iii = 0; iii < 8; iii++)

{

for (int jjj = 0; jjj < 8; jjj++)

{

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

Tab[i, j] = Table[i, j];

//Ignore Gray.

if (Tab[iii, jjj] > 0)

continue;

ThinkingChess AA = new ThinkingChess(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsBoard, ii, jj);

//Wehn There is Attack to Brwon.

if (AA.Attack(Tab, ii, jj, iii, jjj, Color.Gray, Order \* -1))

{

//Move

/\*int a = Tab[iii, jjj];

Tab[iii, jjj] = Tab[ii, jj];

Tab[ii, jj] = 0;

int[,] Tabl = new int[8, 8];

for (int h = 0; h < 8; h++)

for (int g = 0; g < 8; g++)

Tabl[h, g] = Tab[h, g];

ChessRules AAA = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsBoard, Tabl[iii, jjj], Tabl, Order, iii, jjj);

//When There is Check or Checkedmate

if (AAA.Check(Tab, Order))

{

CheckBrownObjectDangour = true;

break;

}

\*/

CheckBrownObjectDangour = true;

}

if (CheckBrownObjectDangour)

break;

}

if (CheckBrownObjectDangour)

break;

}

if (CheckBrownObjectDangour)

break;

}

if (CheckBrownObjectDangour)

break;

}

}

}

//Iniaiate Global Variables.

Object O1 = new Object();

lock (O1)

{

ChessRules.CheckObjectDangourIgnoreSelfThingBetweenTowEnemyKing = false;

}

//If There is Brown ObjectDanger Or Gray ObjectDanger.

if (CheckBrownObjectDangour || CheckGrayObjectDangour)

{

//Iniaate Global Check Variable By Local Variables.

ChessRules.CurrentOrder = CDummy;

Order = COrder;

CheckGray = CheckGrayDummy;

CheckBrown = CheckBrownDummy;

//Achamz is Validity.

return true;

}

ChessRules.CurrentOrder = CDummy;

Order = COrder;

//Iniatiate Of Global Varibales By Local Variables.

CheckGray = CheckGrayDummy;

CheckBrown = CheckBrownDummy;

//Return Not Validiy.

return false;

}

bool AchmazCheckByMoveByRule(int[,] Tabl, int RowF, int ColumnF, int RowS, int ColumnS, int Order)

{

bool Achmaz = false;

int[,] Table = new int[8, 8];

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

Table[i, j] = Tabl[i, j];

Table[RowS, ColumnS] = Table[RowF, ColumnF];

Table[RowF, ColumnF] = 0;

if (Check(Table, Order))

{

if (Order == 1 && CheckGray)

Achmaz = true;

if (Order == -1 && CheckBrown)

Achmaz = true;

}

return Achmaz;

}

public bool ObjectDangourKingMove(int Order, int[,] Table, bool DoIgnore, int ii, int jj)

{

int[,] Tab = new int[8, 8];

//Clone a Copy

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

Tab[i, j] = Table[i, j];

//Initiate Variables.

CheckGray = false;

CheckBrown = false;

CheckGrayObjectDangour = false;

CheckBrownObjectDangour = false;

Object O = new Object();

lock (O)

{

if (DoIgnore)

ChessRules.CheckObjectDangourIgnoreSelfThingBetweenTowEnemyKing = true;

}

//Check identification.

Check(Tab, Order);

bool CheckGrayDummy = CheckGray;

bool CheckBrownDummy = CheckBrown;

//If There is Check on Tow Side.

if (CheckBrown || CheckGray)

{

//Check meand achmaz.

if (CheckBrown)

CheckBrownObjectDangour = true;

if (CheckGray)

CheckGrayObjectDangour = true;

return true;

}

int CDummy = ChessRules.CurrentOrder;

int COrder = Order;

//Location of King Gary

{

//Iniatite Global Varibales.

ChessRules.CurrentOrder = -1;

Order = -1;

//For Enemies.

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

//Ignore of current.

if (Order == 1 && Tab[i, i] >= 0)

continue;

if (Order == -1 && Tab[i, i] <= 0)

continue;

//For All Current

for (int iii = 0; iii < 8; iii++)

{

for (int jjj = 0; jjj < 8; jjj++)

{

//Ignore of enemies.

if (Order == 1 && Tab[iii, jjj] <= 0)

continue;

if (Order == -1 && Tab[iii, jjj] >= 0)

continue;

//Clone a Copy

for (int ik = 0; ik < 8; ik++)

for (int jk = 0; jk < 8; jk++)

Tab[ik, jk] = Table[ik, jk];

ChessRules A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsBoard, Tab[i, j], Tab, Order \* -1, i, j);

Color a = Color.Gray;

if (Order \* -1 == -1)

a = Color.Brown;

//When Enemies can gard King

if (A.Rules(i, j, iii, jjj, a, Tab[i, j]))

{

Tab[iii, jjj] = Tab[i, j];

Tab[i, j] = 0;

if (A.CheckMate(Tab, Order))

{

if (Order == 1 && A.CheckMateGray)

{

//For Current.

for (int iiii = 0; iiii < 8; iiii++)

{

for (int jjjj = 0; jjjj < 8; jjjj++)

{

//Ignore of enemies.

if (Order == 1 && Tab[iiii, jjjj] <= 0)

continue;

if (Order == -1 && Tab[iiii, jjjj] >= 0)

continue;

//For Enemies and Emety.

for (int iiiii = 0; iiiii < 8; iiiii++)

{

for (int jjjjj = 0; jjjjj < 8; jjjjj++)

{

//Ignore of Current.

if (Order == 1 && Tab[iiiii, jjjjj] > 0)

continue;

if (Order == -1 && Tab[iiiii, jjjjj] < 0)

continue;

for (int ik = 0; ik < 8; ik++)

for (int jk = 0; jk < 8; jk++)

Tab[ik, jk] = Table[ik, jk];

Tab[iii, jjj] = Tab[i, j];

Tab[i, j] = 0;

A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsBoard, Tab[iiii, jjjj], Tab, Order, iiii, jjjj);

if (A.Rules(iiii, jjjj, iiiii, jjjjj, a, Tab[i, j]))

{

Tab[iiiii, jjjjj] = Tab[iiii, jjjj];

Tab[iiii, jjjj] = 0;

if (A.CheckMate(Tab, Order))

{

CheckBrown = A.CheckBrown;

CheckGray = A.CheckGray;

CheckMateGray = A.CheckMateGray;

CheckMateBrown = A.CheckMateBrown;

CheckGrayObjectDangour = A.CheckGrayObjectDangour;

CheckBrownObjectDangour = A.CheckBrownObjectDangour;

ChessRules.CurrentOrder = CDummy;

Order = COrder;

return true;

}

}

}

}

}

}

}

else

if (Order == -1 && A.CheckMateBrown)

{

//For Current.

for (int iiii = 0; iiii < 8; iiii++)

{

for (int jjjj = 0; jjjj < 8; jjjj++)

{

//Ignore of enemies.

if (Order == 1 && Tab[iiii, jjjj] <= 0)

continue;

if (Order == -1 && Tab[iiii, jjjj] >= 0)

continue;

//For Enemies and Emety.

for (int iiiii = 0; iiiii < 8; iiiii++)

{

for (int jjjjj = 0; jjjjj < 8; jjjjj++)

{

//Ignore of Current.

if (Order == 1 && Tab[iiiii, jjjjj] > 0)

continue;

if (Order == -1 && Tab[iiiii, jjjjj] < 0)

continue;

for (int ik = 0; ik < 8; ik++)

for (int jk = 0; jk < 8; jk++)

Tab[ik, jk] = Table[ik, jk];

Tab[iii, jjj] = Tab[i, j];

Tab[i, j] = 0;

A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsBoard, Tab[iiii, jjjj], Tab, Order, iiii, jjjj);

if (A.Rules(iiii, jjjj, iiiii, jjjjj, a, Tab[i, j]))

{

Tab[iiiii, jjjjj] = Tab[iiii, jjjj];

Tab[iiii, jjjj] = 0;

if (A.CheckMate(Tab, Order))

{

CheckBrown = A.CheckBrown;

CheckGray = A.CheckGray;

CheckMateGray = A.CheckMateGray;

CheckMateBrown = A.CheckMateBrown;

CheckGrayObjectDangour = A.CheckGrayObjectDangour;

CheckBrownObjectDangour = A.CheckBrownObjectDangour;

ChessRules.CurrentOrder = CDummy;

Order = COrder;

return true;

}

}

}

}

}

}

}

}

}

}

}

}

}

}

ChessRules.CurrentOrder = CDummy;

Order = COrder;

//Iniatiate Of Global Varibales By Local Variables.

//Return Not Validiy.

return false;

}

//Gray King Founder.

public bool FindGrayKing(int[,] Table, ref int Row, ref int Column)

{

//For All Home Table.

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

{

//If Current is Gray Home

if (Table[i, j] == 6)

{

//Initiate Refreable Parameters.

Row = i;

Column = j;

return true;

}

}

//Not Found.

return false;

}

//Alpahber Object Consideration.

static String ThingsAlphabet(int i)

{

//Initiate a Local Varibale.

String A = "";

//Determinbe Gray Or Brown Movment.

if (i < 0)

A = "Brown:";

if (i > 0)

A = "Gray:";

//Determine Object Alhpabet.

if (System.Math.Abs(i) == 1)

A += "(S)";

if (System.Math.Abs(i) == 2)

A += "(E)";

if (System.Math.Abs(i) == 3)

A += "(H)";

if (System.Math.Abs(i) == 4)

A += "(B)";

if (System.Math.Abs(i) == 5)

A += "(M)";

if (System.Math.Abs(i) == 6)

A += "(K)";

//Retrun Alphabet.

return A;

}

//Row Alphabet Consideration.

static String RowAlphabet(int i)

{

//Initiate Local Variable.

String A = "";

//Row Alphabet Consideration.

if (i == 0)

A = "a";

if (i == 1)

A = "b";

if (i == 2)

A = "c";

if (i == 3)

A = "d";

if (i == 4)

A = "e";

if (i == 5)

A = "f";

if (i == 6)

A = "g";

if (i == 7)

A = "h";

//Return Row Alphabet.

return A;

}

//Create Syntax of Movments.

public String CreateStatistic(bool Arrange, int[,] Tab, int Movments, int SourceThings, int Column, int Row, bool Hit, int HitThings, bool CastleKing, bool SodierConvert//, ref AllDraw. THIS

)

{

if (Movments == 46)

Movments = 46;

ArrangmentsBoard = Arrange;

bool ms = false;

int bn = Movments;

if (bn % 2 == 1)

ms = true;

//Movments String Number Creation in String.

bn = bn / 2 + 1;

String SN = "";

String S = "";

if (ms)

SN = bn.ToString() + ".";

//Consider CheckMate Condition of Table.

ChessRules A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, Arrange, 1, Tab, 1, Row, Column);

ChessRules AA = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, Arrange, 1, Tab, 1, Row, Column);

ChessRules AAA = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, Arrange, 1, Tab, 1, Row, Column);

A.CheckMate(Tab, Order);

AA.ObjectDangourKingMove(Order, Tab, false);

Color a = Color.Gray;

if (Order == -1)

a = Color.Brown;

AAA.Pat(Tab, Order, a);

if (A.CheckGray)

{

Object O2 = new Object();

lock (O2)

{

ChessRules.CastleKingAllowedGray = false;

ChessRules.CastleActGray = true;

ThinkingChess.KingMaovableGray = true;

}

}

else if (A.CheckBrown)

{

Object O2 = new Object();

lock (O2)

{

ChessRules.CastleActBrown = true;

ChessRules.CastleKingAllowedBrown = false;

ThinkingChess.KingMaovableBrown = true;

}

}

bool Castles = false;

if (Order == 1)

if (ChessRules.SmallKingCastleGray || ChessRules.BigKingCastleGray)

Castles = true;

if (Order == -1)

if (ChessRules.SmallKingCastleBrown || ChessRules.BigKingCastleBrown)

Castles = true;

//When Solder Converted or Castles King Acts.

if (SodierConvert || (CastleKing && Castles))

{

//When Castles Acts.

if (CastleKing)

{

//Castles Brown King.

if (ChessRules.SmallKingCastleGray)

{

Object O2 = new Object();

lock (O2)

{

ThinkingChess.KingMaovableGray = true;

S += "Gray-BK-S";

if (!AllDraw.Stockfish)

{

Object O = new Object();

lock (O)

{

ChessRules.SmallKingCastleGray = false;

ChessRules.CastleKingAllowedGray = false;

}

}

}

}

else

if (ChessRules.BigKingCastleGray)

//Castles Brown King.

{

Object O2 = new Object();

lock (O2)

{

S += "Gray-BK-B";

ThinkingChess.KingMaovableGray = true;

if (!AllDraw.Stockfish)

{

Object O = new Object();

lock (O)

{

ChessRules.BigKingCastleGray = false;

ChessRules.CastleKingAllowedGray = false;

}

}

}

}

else

if (ChessRules.SmallKingCastleBrown)

//Castles Brown King.

{

Object O2 = new Object();

lock (O2)

{

S += "Brown-BK-S";

ThinkingChess.KingMaovableBrown = true;

if (!AllDraw.Stockfish)

{

Object O = new Object();

lock (O)

{

ChessRules.SmallKingCastleBrown = false;

ChessRules.CastleKingAllowedBrown = false;

}

}

}

}

else

if (ChessRules.BigKingCastleBrown)

//Castles Brown King.

{

Object O2 = new Object();

lock (O2)

{

S += "Brown-BK-B";

ThinkingChess.KingMaovableBrown = true;

if (!AllDraw.Stockfish)

{

Object O = new Object();

lock (O)

{

ChessRules.BigKingCastleBrown = false;

ChessRules.CastleKingAllowedBrown = false;

}

}

}

}

//Castles Brown King.

//Great Castles Gray King.

}

//Soldier Converted.

if (SodierConvert)

{

//Object Kind String Addition.

S += ThingsAlphabet(SourceThings);

//If Hit Acts.

if (Hit)

{

Object O = new Object();

lock (O)

{

ObjectHittedRow = Row;

ObjectHittedColumn = Column;

}

//THIS.SetObjectInPictureBox(Row, Column);

S += "x";

}

S += Column.ToString();

//CheckMate of Gray Or Brown

if (AAA.PatkGray || AAA.PatBrown)

{

S += "-O-";

}

else

if (A.CheckMateGray || A.CheckMateBrown)

{

S += "++";

}

//Check Of Gray Or Brown.

else if (A.CheckBrown || A.CheckGray)

{

S += "+";

if (A.CheckBrown && Order == -1)

{

Object O2 = new Object();

lock (O2)

{

ThinkingChess.KingMaovableBrown = true;

ChessRules.BigKingCastleBrown = false;

ChessRules.CastleKingAllowedBrown = false;

} }

if (A.CheckGray && Order == 1)

{

Object O2 = new Object();

lock (O2)

{

ThinkingChess.KingMaovableGray = true;

ChessRules.BigKingCastleGray = false;

ChessRules.CastleKingAllowedGray = false;

} }

}

else if (AA.CheckGrayObjectDangour || AA.CheckBrownObjectDangour)

{

if (AA.CheckGrayObjectDangour && Order == -1)

{

Object O2 = new Object();

lock (O2)

{

ThinkingChess.KingMaovableBrown = true;

} }

if (AA.CheckBrownObjectDangour && Order == 1)

{

Object O2 = new Object();

lock (O2)

{

ThinkingChess.KingMaovableGray = true;

} }

}

}

}

else//Brown Order.

{

//Object of Kind.

S += ThingsAlphabet(SourceThings);

//Hit Consideration.

if (Hit)

{

Object O = new Object();

lock (O)

{

ObjectHittedRow = Row;

ObjectHittedColumn = Column;

}

//THIS.SetObjectInPictureBox(Row, Column);

S += "x";

}

//Row Column Consideration.

S += RowAlphabet(Row);

S += Column.ToString();

//CheckMate Consideration.

if (AAA.PatkGray || AAA.PatBrown)

{

S += "-O-";

}

else

if (A.CheckMateGray || A.CheckMateBrown)

{

S += "++";

}

//Gray Consideration.

else if (A.CheckBrown || A.CheckGray)

{

S += "+";

if (A.CheckBrown && Order == -1)

{

Object O2 = new Object();

lock (O2)

{

ChessRules.BigKingCastleBrown = false;

ChessRules.CastleKingAllowedBrown = false;

ThinkingChess.KingMaovableGray = true;

}

}

if (A.CheckGray && Order == 1)

{

Object O2 = new Object();

lock (O2)

{

ChessRules.BigKingCastleGray = false;

ChessRules.CastleKingAllowedGray = false;

ThinkingChess.KingMaovableGray = true;

} }

}

else if (AA.CheckGrayObjectDangour || AA.CheckBrownObjectDangour)

{

if (AA.CheckGrayObjectDangour && Order == -1)

{

Object O2 = new Object();

lock (O2)

{

ThinkingChess.KingMaovableBrown = true;

} }

if (AA.CheckBrownObjectDangour && Order == 1)

{

Object O2 = new Object();

lock (O2)

{

ThinkingChess.KingMaovableGray = true;

} }

}

}

//Separate.

if (AllDraw.Less != Double.MinValue)

S += " With Huristic (" + AllDraw.Less.ToString() + ")--";

else

S += " --";

//Return String Sysntax.

return SN + S;

}

//Consideration of Existing Table in List.

bool ArrayInList(List<int[]> List, int[] A)

{

//Initiate Local Variables.

bool Is = false;

//For each Items of a Tow Part List.

for (int i = 0; i < List.Count; i++)

{

//If Listis Equal Setting of Local Variable Equality.

if (A[0] == List[i][0] && A[1] == List[i][1])

Is = true;

}

//Retrun Condition.

return Is;

}

//Find a Specific Objects.

public bool FindAThing(int[,] Table, ref int Row, ref int Column, int Thing, bool BeMovable, List<int[]> List)

{

//For All Items In Table Home.

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

{

//Initiate Local Variables.

int[] AA = new int[2];

AA[0] = i;

AA[1] = j;

//If Table Home is Eqaul Tow Things Object.

if (Table[i, j] == Thing)

{

//If Set A Global Variable Low Logical.

if (!BeMovable)

{

//If Array Exist In List Continue Traversal Back.

if (ArrayInList(List, AA))

continue;

//Iniatiate Local Varibales.

Row = i;

Column = j;

//Found State.

return true;

}

else//Else of Condition.

{

//Iniatiate Local Variables.

Color A = Color.Gray;

if (Order == -1)

A = Color.Brown;

//For All Second Home.

for (int ii = 0; ii < 8; ii++)

for (int jj = 0; jj < 8; jj++)

{

//If First Home is Movable to Second Home.

if ((new ThinkingChess(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsBoard, i, j)).Movable(Table, i, j, ii, jj, A, Order))

{

//If Array Exist in Home.

if (ArrayInList(List, AA))

continue;

//Initaite Local Variables.

Row = i;

Column = j;

//Found of State

return true;

}

}

}

}

}

//Not Found State.

return false;

}

//Brown King Found Consideration.

public bool FindBrownKing(int[,] Table, ref int Row, ref int Column)

{

//For All Home Table.

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

{

//If Current Home is Brown King.

if (Table[i, j] == -6)

{

//Initiate Refrencable Parameter.

Row = i;

Column = j;

//Found of Brown King.

return true;

}

}

//Not Found.

return false;

}

//A Constraint Check Removed Unused Method.

public bool CheckRemovableByAttack(int[,] Table, int Order)

{

//Initiate Local Variables.

int[,] Tabl = new int[8, 8];

//Clone a Copy.

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

Tabl[i, j] = Table[i, j];

//Initiate Global Variables.

Object O = new Object();

lock (O)

{

CheckGrayRemovable = true;

CheckBrownRemovable = true;

}

Check(Tabl, Order);

//if (Order == -1)

{

//For All Home Tables in Fourth Second Traversal.

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

for (int ii = 0; ii < 8; ii++)

for (int jj = 0; jj < 8; jj++)

{

//If Tow How is the Same Continue Traversal Back.

if (i == ii && j == jj)

continue;

//If is Brown Order.

if (Table[i, j] < 0)

{

//If Is Gray Order.

if (Table[ii, jj] > 0)

{

//Initiate Local Variables.

int[,] Tab = new int[8, 8];

//Clone a Copy.

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

Tab[iii, jjj] = Table[iii, jjj];

}

//If Is Movable.

if ((new ThinkingChess(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsBoard, i, j)).Movable(Tab, i, j, ii, jj, Color.Brown, -1))

{

//Clone a Copy.

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

Tab[iii, jjj] = Table[iii, jjj];

}

//If Brown Check.

if (CheckBrown)

{

//Initiate Local Variables.

Tab[ii, jj] = Tab[i, j];

Tab[i, j] = 0;

//If There is Not Check.

if (!Check(Tab, Order))

{

//If Is Not Brown Check.

if (!CheckBrown)

{

//Initiate and Move.

Tab[i, j] = Table[ii, jj];

Tab[ii, jj] = 0;

Object O1 = new Object();

lock (O1)

{

CheckBrownRemovableValueRowi = i;

CheckGrayRemovableValueColumni = j;

CheckGrayRemovableValueRowii = ii;

CheckGrayRemovableValueColumnjj = jj;

CheckGrayRemovable = true;

}

}

}

//Move Back.

Tab[i, j] = Table[ii, jj];

Tab[ii, jj] = 0;

}

}

}

}

}

}

{

//For All Second Traversal Homes.

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

for (int ii = 0; ii < 8; ii++)

for (int jj = 0; jj < 8; jj++)

{

//if The Tow Traversal are the ame Continue Traversal Back.

if (i == ii && j == jj)

continue;

//If the Gray.

if (Table[i, j] > 0)

{

//If the Brown.

if (Table[ii, jj] < 0)

{

//Inaitate Local Variables.

int[,] Tab = new int[8, 8];

//Clone a Copy.

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

Tab[iii, jjj] = Table[iii, jjj];

}

//Moveable Movemnts in the Tow Traversal Kind.

if ((new ThinkingChess(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsBoard, i, j)).Movable(Tab, i, j, ii, jj, Color.Gray, 1))

{

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

Tab[iii, jjj] = Table[iii, jjj];

}

//If the Gray Check.

if (CheckGray)

{

//Move

Tab[ii, jj] = Tab[i, j];

Tab[i, j] = 0;

//If ther is Not Check.

if (!Check(Tab, Order))

{

//If there is Not Gray Check.

if (!CheckGray)

{

//Move and Initaite Local and Global Variables.

Tab[i, j] = Table[ii, jj];

Tab[ii, jj] = 0;

Object O1 = new Object();

lock (O1)

{

CheckBrownRemovableValueRowi = i;

CheckBrownRemovableValueColumnj = j;

CheckBrownRemovableValueRowii = ii;

CheckBrownRemovableValueColumnjj = jj;

CheckBrownRemovable = true;

}

}

}

//Move Back.

Tab[i, j] = Table[ii, jj];

Tab[ii, jj] = 0;

}

}

}

}

}

}

//If Check Remoavbe Brown Or Gray Return Removable.

if (CheckBrownRemovable || CheckGrayRemovable)

return true;

//Return Not Removable.

return false;

}

bool[,] VeryFye(int[,] Table, int Order, Color a, int ii, int jj)

{

int Cdummy = ChessRules.CurrentOrder;

if (Order == 1)

ChessRules.CurrentOrder = 1;

else

ChessRules.CurrentOrder = -1;

bool[,] Tab = new bool[8, 8];

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

{

if (i == ii && j == jj)

continue;

if ((new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsBoard, Table[ii, jj], Table, Order, ii, jj)).Rules(ii, jj, i, j, a, Table[ii, jj]))

{

Tab[i, j] = true;

}

if ((new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsBoard, Table[ii, jj], Table, Order, ii, jj)).Rules(ii, jj, i, j, a, Table[ii, jj]))

{

Tab[i, j] = true;

}

}

ChessRules.CurrentOrder = Cdummy;

return Tab;

}

public bool OnlyKingMovable(int[,] Tab, bool[,] TabB, int Order)

{

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

{

if (TabB[i, j])

{

if (Order == 1)

{

if (Tab[i, j] != 6)

return false;

}

else

if (Tab[i, j] != -6)

return false;

}

}

return true;

}

public bool Pat(int[,] Tab, int Order, Color a)

{

int[,] Table = new int[8, 8];

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

Table[i, j] = Tab[i, j];

bool Pat = false;

Object O = new Object();

lock (O)

{

PatCheckedInKingRule = true;

}

if (!Check(Table, Order))

{

bool[,] TableS = new bool[8, 8];

// if (Order == -1)

for (int ii = 0; ii < 8; ii++)

for (int jj = 0; jj < 8; jj++)

{

if (Table[ii, jj] > 0)

{

bool[,] TableSS = VeryFye(Table, 1, Color.Gray, ii, jj);

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

TableS[iii, jjj] |= TableSS[iii, jjj];

}

}

}

if (OnlyKingMovable(Table, TableS, 1))

{

NumbersofKingMovesToPatGray++;

}

Pat = false;

for (int ii = 0; ii < 8; ii++)

for (int jj = 0; jj < 8; jj++)

{

Pat |= TableS[ii, jj];

}

Pat = !Pat;

if (Pat || NumbersofKingMovesToPatGray > 16)

{

AllDraw.EndOfGame = true;

PatkGray = true;

}

TableS = new bool[8, 8];

for (int ii = 0; ii < 8; ii++)

for (int jj = 0; jj < 8; jj++)

{

if (Table[ii, jj] < 0)

{

bool[,] TableSS = VeryFye(Table, -1, Color.Brown, ii, jj);

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

TableS[iii, jjj] |= TableSS[iii, jjj];

}

}

}

if (OnlyKingMovable(Table, TableS, -1))

{

NumbersofKingMovesToPatBrown++;

}

Pat = false;

for (int ii = 0; ii < 8; ii++)

for (int jj = 0; jj < 8; jj++)

{

Pat |= TableS[ii, jj];

}

Pat = !Pat;

if (Pat || NumbersofKingMovesToPatBrown >= 16)

{

AllDraw.EndOfGame = true;

PatBrown = true;

}

if (PatkGray || PatBrown)

Pat = true;

}

else

{

if (CheckGray)

NumbersofKingMovesToPatGray = 0;

else

if (CheckBrown)

NumbersofKingMovesToPatBrown = 0;

}

Object O1 = new Object();

lock (O1)

{

PatCheckedInKingRule = false;

}

return Pat;

}

void CheckKing(int[,] Table, int Ord, int RowK, int ColumnK)

{

int[,] Tab = new int[8, 8];

//Clone a Copy.

for (int ii = 0; ii < 8; ii++)

for (int jj = 0; jj < 8; jj++)

Tab[ii, jj] = Table[ii, jj];

bool BREAK = false;

//For All Home Table.

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

//If The Current Home is the Gray King Continue Traversal Back.

if (i == RowK && j == ColumnK)

continue;

if (Ord == 1 & Tab[i, j] <= 0)

continue;

if (Ord == -1 & Tab[i, j] >= 0)

continue;

//Initiate Global Variables.

int Dummt = ChessRules.CurrentOrder;

ChessRules.CurrentOrder = -1;

//Clone a Copy.

for (int ii = 0; ii < 8; ii++)

for (int jj = 0; jj < 8; jj++)

Tab[ii, jj] = Table[ii, jj];

Color a = Color.Gray;

if (Ord == -1)

a = Color.Brown;

ChessRules A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsBoard, Table[i, j], Table, Ord, i, j);

if (Ord == -1)

{

//Menen Parameter is Moveble to Second Parameters Location returm Movable.

if (A.Rules(i, j, RowK, ColumnK, a, Ord))

{

BREAK = true;

//Initiate Local Is Check Variables.

CheckGray = true;

break;

}

}

else

{ //Menen Parameter is Moveble to Second Parameters Location returm Movable.

if (A.Rules(i, j, RowK, ColumnK, a, Ord))

{

BREAK = true;

CheckBrown = true;

break;

}

}

//Initiate Global Variables.

ChessRules.CurrentOrder = Dummt;

}

if (BREAK)

break;

}

}

//Check Consideration Method.

public bool Check(int[,] Table, int Ord)

{

int DummyOrder = Ord;

//Initiate Local and Global Briables.

bool Store = ChessRules.CheckObjectDangourIgnoreSelfThingBetweenTowEnemyKing;

Object O = new Object();

lock (O)

{

ChessRules.CheckObjectDangourIgnoreSelfThingBetweenTowEnemyKing = false;

}

CheckGray = false;

CheckBrown = false;

//Initiate Local Variables.

int RowG = 0, ColumnG = 0;

int RowB = 0, ColumnB = 0;

//if (Ord == 1)

//Foud of Gray King.

if (FindGrayKing(Table, ref RowG, ref ColumnG))

CheckKing(Table, -1, RowG, ColumnG);

//Found of Brown King.

if (FindBrownKing(Table, ref RowB, ref ColumnB))

CheckKing(Table, 1, RowB, ColumnB);

Ord = DummyOrder;

//If Gray Check Or brwon Check return Check..

if (CheckBrown || CheckGray)

return true;

//Return Non Check.

return false;

}

void CheckMateKing(int[,] Tab, int Ord, bool CheckGrayDummy, bool CheckBrownDummy, int RowK, int ColumnK, ref bool ActMove, bool Checked)

{

int DummyOrder = Order;

//For All Home Table.

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

if (Ord == 1 && Tab[i, j] > 0)

continue;

if (Ord == -1 && Tab[i, j] < 0)

continue;

//Clone a Copy.

CheckGray = CheckGrayDummy;

CheckBrown = CheckBrownDummy;

//If There is Gray Check.

if (Checked)

{

//Initiate Global Variables.

ChessRules.CurrentOrder = 1;

//Ig Gray King is Movable to First Home Table.

Color a = Color.Gray;

if (Ord == -1)

a = Color.Brown;

ChessRules A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsBoard, Table[RowK, ColumnK], Table, Ord, RowK, ColumnK);

Order = DummyOrder;

///Table[ii, jj] = 0;

//Menen Parameter is Moveble to Second Parameters Location returm Movable.

for (int k = 0; k < 8; k++)

for (int p = 0; p < 8; p++)

Table[k, p] = Tab[k, p];

if (A.Rules(RowK, ColumnK, i, j, a, Ord))

{

Order = DummyOrder;

//Initaite Loval and Move.

//ActMove = false;

int Store = Table[i, j];

//For Another Methods

Table[i, j] = Table[RowK, ColumnK];

Table[RowK, ColumnK] = 0;

//If Is Check.

if (A.Check(Table, Ord))

{

//Move Back.

//If Gray Check.

if (Ord == 1)

{

if (A.CheckGray)

{

//Move Mack.

ActMove = true;

continue;

}

else//If There is Not Gray Check.

{

//Move Back.

ActMove = false;

break;

}

}

else

{

if (A.CheckBrown)

{

//Move Mack.

ActMove = true;

continue;

}

else//If There is Not Gray Check.

{

//Move Back.

ActMove = false;

break;

}

}

}

else

{

//Comon Move Back.

ActMove = false;

break;

}

}

}

}

//If One of The Not Movable.

if (!ActMove)

break;

}

Order = DummyOrder;

}

void CheckMateNotKing(int[,] Tab, int Ord, bool CheckGrayDummy, bool CheckBrownDummy, ref bool ActMove)

{

int DummyOrder = Ord;

//For All Home Table.

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

if (Ord == 1 && Tab[i, j] <= 0)

continue;

if (Ord == -1 && Tab[i, j] >= 0)

continue;

//Initiate Global varibales.

CheckGray = CheckGrayDummy;

CheckBrown = CheckBrownDummy;

//Clone a Copy.

CheckGray = CheckGrayDummy;

CheckBrown = CheckBrownDummy;

//If There is Gray Check.

//Initiate Local Varibale.

ActMove = true;

//For All Second Home Table.

for (int ii = 0; ii < 8; ii++)

{

for (int jj = 0; jj < 8; jj++)

{

if (Ord == 1 && Tab[ii, jj] > 0)

continue;

if (Ord == -1 && Tab[ii, jj] < 0)

continue;

//Clone a Copy.

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

Table[iii, jjj] = Tab[iii, jjj];

Color a = Color.Gray;

if (Ord == -1)

a = Color.Brown;

ChessRules A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsBoard, Table[i, j], Table, Ord, i, j);

///Table[ii, jj] = 0;

//Menen Parameter is Moveble to Second Parameters Location returm Movable.

if (A.Rules(i, j, ii, jj, a, Ord))

{

Order = DummyOrder;

//Initiate Local Varibales and Move.

//ActMove = false;

//For Another Methods

int Store = Table[i, j];

Table[ii, jj] = Table[i, j];

Table[i, j] = 0;

//If Check.

A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsBoard, Table[ii, jj], Table, Ord, ii, jj);

if (A.Check(Table, Ord))

{

Order = DummyOrder;

//Move Back.

Table[i, j] = Table[ii, jj];

Table[ii, jj] = Store;

//If Gray Check.

if (Ord == 1)

{

if (A.CheckGray)

{

//Initiate and Move Back.

ActMove = true;

Table[i, j] = Table[ii, jj];

Table[ii, jj] = Store;

continue;

}

//If There is Not Gray Check.

else

{

//Initiate Varaible and Move Back.

ActMove = false;

Table[i, j] = Table[ii, jj];

Table[ii, jj] = Store;

break;

}

}

else

{

if (A.CheckBrown)

{

//Initiate and Move Back.

ActMove = true;

Table[i, j] = Table[ii, jj];

Table[ii, jj] = Store;

continue;

}

//If There is Not Gray Check.

else

{

//Initiate Varaible and Move Back.

ActMove = false;

Table[i, j] = Table[ii, jj];

Table[ii, jj] = Store;

break;

}

}

}

else

{

//Move Back and Initiate.

Table[i, j] = Table[ii, jj];

Table[ii, jj] = Store;

ActMove = false;

break;

}

}

}

//If Not Movable Break.

if (!ActMove)

break;

}

if (!ActMove)

break;

}

//If Not Movable Break.

if (!ActMove)

break;

}

Order = DummyOrder;

}

//CheckMate Consideration.QC-OK

public bool CheckMate(int[,] Tab, int Ord)

{

//Initiate Local and Global Varibales.

int[,] Table = new int[8, 8];

try

{

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

Table[i, j] = Tab[i, j];

}

catch (Exception t)

{

Log(t);

return false;

}

CheckGray = false;

CheckBrown = false;

CheckMateBrown = false;

CheckMateGray = false;

bool ActMoveG = true;

bool ActMoveGF = true;

bool ActMoveB = true;

bool ActMoveBF = true;

int RowG = 0, ColumnG = 0;

int RowB = 0, ColumnB = 0;

int DumnyOrder = Ord;

//Check Consideration.

Check(Table, Ord);

//Initiate Local Varibales.

bool CheckGrayDummy = CheckGray;

bool CheckBrownDummy = CheckBrown;

ActMoveG = true;

ActMoveGF = true;

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

Table[i, j] = Tab[i, j];

ChessRules A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsBoard, Table[RowG, ColumnG], Table, Ord, RowG, ColumnG);

//Found of Gray King.

if (FindGrayKing(Table, ref RowG, ref ColumnG))

A.CheckMateKing(Table, 1, CheckGrayDummy, CheckBrownDummy, RowG, ColumnG, ref ActMoveG, CheckGray);

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

Table[i, j] = Tab[i, j];

//Found of Gray King.

if (FindGrayKing(Table, ref RowG, ref ColumnG))

A.CheckMateNotKing(Table, 1, CheckGrayDummy, CheckBrownDummy, ref ActMoveGF);

//Intiate Global Variables.

CheckGray = CheckGrayDummy;

CheckBrown = CheckBrownDummy;

//Condition of CheckMate Gray King.

if (CheckGray && (ActMoveG && ActMoveGF))

CheckMateGray = true;

ActMoveB = true;

ActMoveBF = true;

ChessRules AA = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsBoard, Table[RowB, ColumnB], Table, Ord, RowB, ColumnB);

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

Table[i, j] = Tab[i, j];

//Found of Brown King.

if (FindBrownKing(Table, ref RowB, ref ColumnB))

AA.CheckMateKing(Table, -1, CheckGrayDummy, CheckBrownDummy, RowB, ColumnB, ref ActMoveB, CheckBrown);

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

Table[i, j] = Tab[i, j];

//Found of Brown King.

if (FindBrownKing(Table, ref RowB, ref ColumnB))

AA.CheckMateNotKing(Table, -1, CheckGrayDummy, CheckBrownDummy, ref ActMoveBF);

//Initiate Global Varibales.

CheckGray = CheckGrayDummy;

CheckBrown = CheckBrownDummy;

//Condition of Brown CheckMate.

if (CheckBrown && (ActMoveB && ActMoveBF))

CheckMateBrown = true;

//Initiate Global Variables.

Ord = DumnyOrder;

//If Brown CheckMate and Gray.

if (CheckMateGray || CheckMateBrown)

{

//Initiate Global Variable and Return CheckMate.

CheckGray = CheckGrayDummy;

CheckBrown = CheckBrownDummy;

AllDraw.EndOfGame = true;

return true;

}

//Initiate Global Variables.

CheckGray = CheckGrayDummy;

CheckBrown = CheckBrownDummy;

//Return Not CheckMate.

return false;

}

//Internal Rule of Chess Method.

private bool Rule(int RowFirst, int ColumnFirst, int RowSecond, int ColumnSecond, bool NotMoved, Color color, bool ExistInDestinationEnemy, int Ki, bool SelfHomeStatCP)

{

//When is Not Castles King State.

if (Kind != 7)

{

//Determination of Enemy Existing.

if (ExistSelfHome(RowFirst, ColumnFirst, RowSecond, ColumnSecond, NotMoved, color, Ki) && SelfHomeStatCP)

return false;

}

//Determination of King Enemy at Destination Home.

/\*if (!KingAttacker)

{

//Coluld not hit King In Destination Enemy.

if (Order == 1 && Table[RowSecond, ColumnSecond] == -6)

return false;

if (Order == -1 && Table[RowSecond, ColumnSecond] == 6)

return false;

}\*/

//If Source and The Destination are The Same.

if (RowFirst == RowSecond && ColumnFirst == ColumnSecond)

return false;

//Initiate Global Variable.

Object O = new Object();

lock (O)

{

KingAttacker = false;

}

//Rule of Soldeir.

if (Kind == 1)

return SoldierRules(RowFirst, ColumnFirst, RowSecond, ColumnSecond, NotMoved, color, ExistInDestinationEnemy);

else//Rule of Castles.

if (Kind == 4)

return CastleRules(RowFirst, ColumnFirst, RowSecond, ColumnSecond, NotMoved, color, ExistInDestinationEnemy, Ki);

else//Rule of Hourses.

if (Kind == 3)

return HourseRules(RowFirst, ColumnFirst, RowSecond, ColumnSecond, NotMoved, color, ExistInDestinationEnemy);

else//Rule of Elephant.

if (Kind == 2)

return ElefantRules(RowFirst, ColumnFirst, RowSecond, ColumnSecond, NotMoved, color, ExistInDestinationEnemy, Ki);

else

if (Kind == 5)//Rule of Ministers.

return MinisterRules(RowFirst, ColumnFirst, RowSecond, ColumnSecond, NotMoved, color, ExistInDestinationEnemy, Ki);

else

if (Kind == 6)//Rule of Kings.

return KingRules(RowFirst, ColumnFirst, RowSecond, ColumnSecond, NotMoved, color, ExistInDestinationEnemy, Ki);

else

if (Kind == 7)//Rule of Castles King.

return CastleKing(RowFirst, ColumnFirst, RowSecond, ColumnSecond, NotMoved, color, Ki);

//Non Rulements.

return false;

}

//King Rule Method.

public bool KingRules(int RowFirst, int ColumnFirst, int RowSecond, int ColumnSecond, bool NotMoved, Color color, bool ExistInDestinationEnemy, int Ki)

{

bool Move = false;

//When Miniaster Rule is Valid.

if (MinisterRules(RowFirst, ColumnFirst, RowSecond, ColumnSecond, NotMoved, color, ExistInDestinationEnemy, Ki) && (System.Math.Abs(RowFirst - RowSecond) <= 1) && (System.Math.Abs(ColumnFirst - ColumnSecond) <= 1))

{

//Initiate Local Variable.

/\* int[,] Tab = new int[8, 8];

//Clone A Copy.,

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

{

Tab[i, j] = Table[i, j];

}

//Initiate Local Varibale and Move.

int Store = Tab[RowSecond, ColumnSecond];

Tab[RowSecond, ColumnSecond] = Tab[RowFirst, ColumnFirst];

Tab[RowFirst, ColumnFirst] = 0;

//When There is Check State.

if (Check(Tab, Order))

{

if (!PatCheckedInKingRule)

{

//Check Gray State return Non Rule.

if (Order == 1 && CheckGray)

return false;

else//Brown Check State return Non Rule.

if (Order == -1 && CheckBrown)

return false;

}

else

{

//Check Gray State return Non Rule.

if (Order == -1 && CheckGray)

return false;

else//Brown Check State return Non Rule.

if (Order == 1 && CheckBrown)

return false;

}

}

//Determination of Gray Enemy State Check at Enemy King at Around Existing Return Not Validity.

if (Order == 1 && Table[RowFirst, ColumnFirst] == 6)

{

try

{

if ((RowSecond + 1) < 8)

{

if (Table[RowSecond + 1, ColumnSecond] == -6)

return false;

}

}

catch (Exception t) { Log(t); }

try

{

if ((ColumnSecond + 1) < 8)

{

if (Table[RowSecond, ColumnSecond + 1] == -6)

return false;

}

}

catch (Exception t) { Log(t); }

try

{

if (((RowSecond + 1) < 8) && ((ColumnSecond + 1) < 8))

{

if (Table[RowSecond + 1, ColumnSecond + 1] == -6)

return false;

}

}

catch (Exception t) { Log(t); }

try

{

if (((RowSecond - 1) >= 0))

{

if (Table[RowSecond - 1, ColumnSecond] == -6)

return false;

}

}

catch (Exception t) { Log(t); }

try

{

if (ColumnSecond - 1 >= 0)

{

if (Table[RowSecond, ColumnSecond - 1] == -6)

return false;

}

}

catch (Exception t) { Log(t); }

try

{

if (((RowSecond - 1) >= 0) && ((ColumnSecond - 1) >= 0))

{

if (Table[RowSecond - 1, ColumnSecond - 1] == -6)

return false;

}

}

catch (Exception t) { Log(t); }

try

{

if (((RowSecond + 1) < 8) && ((ColumnSecond - 1) >= 0))

{

if (Table[RowSecond + 1, ColumnSecond - 1] == -6)

return false;

}

}

catch (Exception t) { Log(t); }

try

{

if (((RowSecond - 1) >= 0) && ((ColumnSecond + 1) < 8))

{

if (Table[RowSecond - 1, ColumnSecond + 1] == -6)

return false;

}

}

catch (Exception t) { Log(t); }

}//Determination of Brown Enemy State Check at Enemy King at Around Existing Return Not Validity.

else if (Order == -1 && Table[RowFirst, ColumnFirst] == -6)

{

try

{

if ((RowSecond + 1) < 8)

{

if (Table[RowSecond + 1, ColumnSecond] == 6)

return false;

}

}

catch (Exception t) { Log(t); }

try

{

if ((ColumnSecond + 1) < 8)

{

if (Table[RowSecond, ColumnSecond + 1] == 6)

return false;

}

}

catch (Exception t) { Log(t); }

try

{

if (((RowSecond + 1) < 8) && ((ColumnSecond + 1) < 8))

{

if (Table[RowSecond + 1, ColumnSecond + 1] == 6)

return false;

}

}

catch (Exception t) { Log(t); }

try

{

if (((RowSecond - 1) >= 0))

{

if (Table[RowSecond - 1, ColumnSecond] == 6)

return false;

}

}

catch (Exception t) { Log(t); }

try

{

if (ColumnSecond - 1 >= 0)

{

if (Table[RowSecond, ColumnSecond - 1] == 6)

return false;

}

}

catch (Exception t) { Log(t); }

try

{

if (((RowSecond - 1) >= 0) && ((ColumnSecond - 1) >= 0))

{

if (Table[RowSecond - 1, ColumnSecond - 1] == 6)

return false;

}

}

catch (Exception t) { Log(t); }

try

{

if (((RowSecond + 1) < 8) && ((ColumnSecond - 1) >= 0))

{

if (Table[RowSecond + 1, ColumnSecond - 1] == 6)

return false;

}

}

catch (Exception t) { Log(t); }

try

{

if (((RowSecond - 1) >= 0) && ((ColumnSecond + 1) < 8))

{

if (Table[RowSecond - 1, ColumnSecond + 1] == 6)

return false;

}

}

catch (Exception t) { Log(t); }

}

\*/

Move = true;

}

return Move;

}

//Rules of Minister Method.

public bool MinisterRules(int RowFirst, int ColumnFirst, int RowSecond, int ColumnSecond, bool NotMoved, Color color, bool ExistInDestinationEnemy, int Ki)

{

bool Move = false;

//When is Castles Rule.

if (CastleRules(RowFirst, ColumnFirst, RowSecond, ColumnSecond, NotMoved, color, ExistInDestinationEnemy, Ki))

//Return Validity.,

Move = true;

else

//When is Elephant Rule.

if (ElefantRules(RowFirst, ColumnFirst, RowSecond, ColumnSecond, NotMoved, color, ExistInDestinationEnemy, Ki))

//Return Validity.,

Move = true;

//Return Not Valididty.

return Move;

}

//Castles Rule Method.

public bool CastleRules(int RowFirst, int ColumnFirst, int RowSecond, int ColumnSecond, bool NotMoved, Color color, bool ExistInDestinationEnemy, int Ki)

{

bool Move = false;

bool Act = false;

//If Variation is Only in Row.

if (System.Math.Abs(ColumnFirst - ColumnSecond) == 0 && System.Math.Abs(RowFirst - RowSecond) != 0)

{

//Initiate Local Variables.

int RowU = RowSecond, RowD = RowFirst;

int ColD = ColumnFirst, ColU = ColumnSecond;

int Rowf = 1, Colf = 1;

if (RowU < RowD)

Rowf = -1;

if (ColU < ColD)

Colf = -1;

int incf = 0, incR = 0;

if (Rowf < 0)

incf = -1;

if (Colf < 0)

incR = -1;

int F = 0, G = 0;

int A = 0, B = 0;

if (incf < 0)

{

F = RowU;

G = RowD;

}

else

{

F = RowD;

G = RowU;

}

if (incR < 0)

{

A = ColU;

B = ColD;

}

else

{

A = ColD;

B = ColU;

}

{

//For Variation of Row Home.

for (int i = F; i <= G; i++)

{

if (IgnoreSelfObject && i == RowSecond)

continue;

//When is Not Current Source Home.

if (i != RowFirst)

{

//When There is Self Home at Home of Gray Return Not Validity.

if (Table[i, ColumnFirst] > 0 && Table[RowFirst, ColumnFirst] > 0)

{

Move = false;

Act = true;

}

//When There is Self Home of Brown Objects Return Not Validity.

if (Table[i, ColumnFirst] < 0 && Table[RowFirst, ColumnFirst] < 0)

{

Act = true;

Move = false;

}

//If Situation is Occured.

if (i != RowSecond)

{

//When There is Slef Home at Root Return Not Valididty.

if ((Table[i, ColumnFirst] < 0 || Table[i, ColumnFirst] > 0) && Table[RowFirst, ColumnFirst] > 0)

{

Act = true;

Move = false;

}

//When There is Slef Home at Root Return Not Valididty.

if ((Table[i, ColumnFirst] > 0 || Table[i, ColumnFirst] < 0) && Table[RowFirst, ColumnFirst] < 0)

{

Act = true;

Move = false;

}

}

}

}

}

if (!Act)

Move = true;

}

//When There is Only Column Variation Home Changes.

if (System.Math.Abs(ColumnFirst - ColumnSecond) != 0 && System.Math.Abs(RowFirst - RowSecond) == 0)

{

//Initiate Local Variables.

int RowU = RowSecond, RowD = RowFirst;

int ColD = ColumnFirst, ColU = ColumnSecond;

int Rowf = 1, Colf = 1;

if (RowU < RowD)

Rowf = -1;

if (ColU < ColD)

Colf = -1;

int incf = 0, incR = 0;

if (Rowf < 0)

incf = -1;

if (Colf < 0)

incR = -1;

int F = 0, G = 0;

int A = 0, B = 0;

if (incf < 0)

{

F = RowU;

G = RowD;

}

else

{

F = RowD;

G = RowU;

}

if (incR < 0)

{

A = ColU;

B = ColD;

}

else

{

A = ColD;

B = ColU;

}

//For All Column Home Variation.

for (int j = A; j <= B; j++)

{

if (IgnoreSelfObject && j == ColumnSecond)

continue;

//When The Source is Not The Current.

if (j != ColumnFirst)

{

//For All Self Home at Root Return Not Validity

if (Table[RowFirst, j] > 0 && Table[RowFirst, ColumnFirst] > 0)

{

Act = true;

Move = false;

}

//For All Self Home at Root Return Not Validity.

if (Table[RowFirst, j] < 0 && Table[RowFirst, ColumnFirst] < 0)

{

Act = true;

Move = false;

}

//Condition Determination.

if (j != ColumnSecond)

{

//Existing of Self Home At Root Cuased to Not validity.

if ((Table[RowFirst, j] < 0 || Table[RowFirst, j] > 0) && Table[RowFirst, ColumnFirst] > 0)

{

Act = true;

Move = false;

}

//Existing of Self Home At Root Cuased to Not validity.

if ((Table[RowFirst, j] > 0 || Table[RowFirst, j] < 0) && Table[RowFirst, ColumnFirst] < 0)

{

Act = true;

Move = false;

}

}

}

}

//Return Validity.

if (!Act)

Move = true;

}

//Return Not Validity.

/\*if (Move && System.Math.Abs(Ki) != 6)

{

if (AchmazCheckByMoveByRule(Table, RowFirst, ColumnFirst, RowSecond, ColumnSecond, Order))

Move = false;

}

\*/

//Return not Vailidity.

return Move;

}

//Elephant Rule Method.

public bool ElefantRules(int RowFirst, int ColumnFirst, int RowSecond, int ColumnSecond, bool NotMoved, Color color, bool ExistInDestinationEnemy, int Ki)

{

bool Move = false;

bool Act = false;

//Orthogonal Movments of One Abs Derivation.

if (System.Math.Abs(ColumnFirst - ColumnSecond) == System.Math.Abs(RowFirst - RowSecond))

{

//Initaiet Of Local Variables.

int RowU = RowSecond, RowD = RowFirst;

int ColD = ColumnFirst, ColU = ColumnSecond;

int Rowf = 1, Colf = 1;

if (RowU < RowD)

Rowf = -1;

if (ColU < ColD)

Colf = -1;

int incf = 0, incR = 0;

if (Rowf < 0)

incf = -1;

if (Colf < 0)

incR = -1;

int F = 0, G = 0;

int A = 0, B = 0;

if (incf < 0)

{

F = RowU;

G = RowD;

}

else

{

F = RowD;

G = RowU;

}

if (incR < 0)

{

A = ColU;

B = ColD;

}

else

{

A = ColD;

B = ColU;

}

//For All Root Source to Destination.

for (int i = F; i <= G; i++)

for (int j = A; j <= B; j++)

{

if (IgnoreSelfObject && i == RowSecond && j == ColumnSecond)

continue;

//If Abs Derivation is Not One Continue.

if (System.Math.Abs(i - RowFirst) != System.Math.Abs(j - ColumnFirst))

continue;

//If the Current is Not Source Home.

if (i != RowFirst && j != ColumnFirst)

{

{

//If the Root Contains Self Home Return Not Validity.

if (Table[i, j] > 0 && Table[RowFirst, ColumnFirst] > 0)

{

Act = true;

Move = false;

}

//If The Root Contains Self Home Return Not vALIDITY.

if (Table[i, j] < 0 && Table[RowFirst, ColumnFirst] < 0)

{

Act = true;

Move = false;

}

//When the Current is Not The Source Home.

if (i != RowSecond && j != ColumnSecond)

{

//When the Self ObjectExisting at the Root .

if ((Table[i, j] > 0 || Table[i, j] < 0) && Table[RowFirst, ColumnFirst] > 0)

{

Act = true;

Move = false;

}

//When the Self ObjectExisting at the Root .

if ((Table[i, j] < 0 || Table[i, j] > 0) && Table[RowFirst, ColumnFirst] < 0)

{

Act = true;

Move = false;

}

}

}

}

}

//Return Validity.

if (!Act)

Move = true;

}

/\*if (Move && System.Math.Abs(Ki) != 6)

{

if (AchmazCheckByMoveByRule(Table, RowFirst, ColumnFirst, RowSecond, ColumnSecond, Order))

Move = false;

}

\*/

//Return Not Validity.

return Move;

}

//Hource Rule Method.

public bool HourseRules(int RowFirst, int ColumnFirst, int RowSecond, int ColumnSecond, bool NotMoved, Color color, bool ExistInDestinationEnemy)

{

bool Move = false;

//When L Movament is Occured.

if (System.Math.Abs(ColumnFirst - ColumnSecond) == 2 && System.Math.Abs(RowFirst - RowSecond) == 1)

{

//Retrun Validity.

Move = true;

}

//When Second L Movments Occured.

if (System.Math.Abs(ColumnFirst - ColumnSecond) == 1 && System.Math.Abs(RowFirst - RowSecond) == 2)

{

//Return Validity.

Move = true;

}

//Return Not Validity.

/\* if (Move)

{

if (AchmazCheckByMoveByRule(Table, RowFirst, ColumnFirst, RowSecond, ColumnSecond, Order))

Move = false;

}

\*/

return Move;

}

public bool SoldierRulesaArrangmentsBoardOne(int RowFirst, int ColumnFirst, int RowSecond, int ColumnSecond, bool NotMoved, Color color, bool ExistInDestinationEnemy)

{

bool Move = false;

//When int is Gray.

if (Order == 1)

{

//If Not Forward Return Not Validity.

if (ColumnFirst < ColumnSecond)

Move = false;

}

else//int of Brown.

if (Order == -1)

{

//If Not Back Wrad Return Not Vlaidity.

if (ColumnFirst > ColumnSecond)

Move = false;

}

//When Soldier Not Moved in Original Location do

if (NotMoved)

{

if (Order == -1 && Table[RowFirst, ColumnFirst] < 0)

{

//Depend on First Move do For Land Of Islam

try

{

if ((ColumnFirst + 2 < 8) && (ColumnFirst + 1 < 8) &&

(RowFirst == RowSecond) && (ColumnSecond == ColumnFirst + 2) && (Table[RowSecond, ColumnSecond - 1] == 0)

)

{

//When Destination is The Empty Return Validity Else Return Not Validity.

if (Table[RowSecond, ColumnSecond] == 0)

Move = true;

else

Move = false;

}

else

if ((ColumnFirst + 1 < 8) &&

(RowFirst == RowSecond) && (ColumnSecond == ColumnFirst + 1) && (Table[RowSecond, ColumnSecond] == 0))

{

//When Destination is The Empty Return Validity Else Return Not Validity.

if (Table[RowSecond, ColumnSecond] == 0)

Move = true;

else

Move = false;

}

else//Hit Brown Soldier Rulments.

if ((ColumnFirst + 1 < 8) && ColumnSecond == ColumnFirst + 1)

{

if ((RowSecond - 1 < 8) &&

(RowFirst == RowSecond - 1) && (ExistInDestinationEnemy || IgnoreSelfObject))

{

Move = true;

}

if ((RowSecond + 1 < 8) &&

(RowFirst == RowSecond + 1) && (ExistInDestinationEnemy || IgnoreSelfObject))

{

Move = true;

}

}

}

catch (Exception t)

{

Log(t);

}

}

else//Gray int.

if (Order == 1 && Table[RowFirst, ColumnFirst] > 0)

{

//Depend Of First Move do For Positivism

try

{

if ((ColumnSecond + 2 < 8) && (ColumnSecond + 1 < 8) &&

(RowFirst == RowSecond) && (ColumnFirst == ColumnSecond + 2) && (Table[RowSecond, ColumnSecond + 1] == 0)

)

{

//When Destination is The Empty Return Validity Else Return Not Validity.

if (Table[RowSecond, ColumnSecond] == 0)

Move = true;

else

Move = false;

}

else

if ((ColumnSecond + 1 < 8) &&

(RowFirst == RowSecond) && (ColumnFirst == ColumnSecond + 1) && (Table[RowSecond, ColumnSecond] == 0))

{

//When Destination is The Empty Return Validity Else Return Not Validity.

if (Table[RowSecond, ColumnSecond] == 0)

Move = true;

else

Move = false;

}

else//Hit Condition Enemy Movments.

if ((ColumnSecond + 1 < 8) && ColumnFirst == ColumnSecond + 1)

{

if ((RowSecond + 1 < 8) &&

(RowFirst == RowSecond + 1) && (ExistInDestinationEnemy || IgnoreSelfObject))

//Return Validity.

Move = true;

if ((RowSecond - 1 >= 0) &&

(RowFirst == RowSecond - 1) && (ExistInDestinationEnemy || IgnoreSelfObject))

//Return Validity.

Move = true;

}

}

catch (Exception t)

{

Log(t);

}

}

}

else//If Soldeior Moved Previously.

{

//For Brown int.

if (Order == -1 && Table[RowFirst, ColumnFirst] < 0)

{

//Depend on Second Move do For Land Of Islam

try

{

if ((ColumnFirst + 1 < 8) &&

(RowFirst == RowSecond) && (ColumnSecond == ColumnFirst + 1) && (Table[RowSecond, ColumnSecond] == 0))

{

//When Destination is The Empty Return Validity Else Return Not Validity.

if (Table[RowSecond, ColumnSecond] == 0)

Move = true;

else

Move = false;

}

else//Hit Brown Soldier Rulments.

if ((ColumnFirst + 1 < 8) && ColumnSecond == ColumnFirst + 1)

{

if ((RowSecond - 1 < 8) &&

(RowFirst == RowSecond - 1) && (ExistInDestinationEnemy || IgnoreSelfObject))

{

Move = true;

}

if ((RowSecond + 1 < 8) &&

(RowFirst == RowSecond + 1) && (ExistInDestinationEnemy || IgnoreSelfObject))

{

Move = true;

}

}

}

catch (Exception t)

{

Log(t);

}

}

else//Gray int.

if (Order == 1 && Table[RowFirst, ColumnFirst] > 0)

{

//Depend Of Second Move do For Positivism Land

try

{

if ((ColumnSecond + 1 < 8) &&

(RowFirst == RowSecond) && (ColumnFirst == ColumnSecond + 1) && (Table[RowSecond, ColumnSecond] == 0))

{

//When Destination is The Empty Return Validity Else Return Not Validity.

if (Table[RowSecond, ColumnSecond] == 0)

Move = true;

else

Move = false;

}

else//Hit Condition Enemy Movments.

if ((ColumnSecond + 1 < 8) && ColumnFirst == ColumnSecond + 1)

{

if ((RowSecond + 1 < 8) &&

(RowFirst == RowSecond + 1) && (ExistInDestinationEnemy || IgnoreSelfObject))

//Return Validity.

Move = true;

if ((RowSecond - 1 >= 0) &&

(RowFirst == RowSecond - 1) && (ExistInDestinationEnemy || IgnoreSelfObject))

//Return Validity.

Move = true;

}

}

catch (Exception t)

{

Log(t);

}

}

}

return Move;

}

public bool SoldierRulesaArrangmentsBoardZero(int RowFirst, int ColumnFirst, int RowSecond, int ColumnSecond, bool NotMoved, Color color, bool ExistInDestinationEnemy)

{

bool Move = false;

//When int is Gray.

if (Order == 1)

{

//If Not Forward Return Not Validity.

if (ColumnFirst > ColumnSecond)

Move = false;

}

else//int of Brown.

if (Order == -1)

{

//If Not Back Wrad Return Not Vlaidity.

if (ColumnFirst < ColumnSecond)

Move = false;

}

//When Soldier Not Moved in Original Location do

if (NotMoved)

{

if (Order == 1 && Table[RowFirst, ColumnFirst] > 0)

{

//Depend on First Move do For Land Of Islam

try

{

if ((ColumnFirst + 2 < 8) && (ColumnFirst + 1 < 8) &&

(RowFirst == RowSecond) && (ColumnSecond == ColumnFirst + 2) && (Table[RowSecond, ColumnSecond - 1] == 0)

)

{

//When Destination is The Empty Return Validity Else Return Not Validity.

if (Table[RowSecond, ColumnSecond] == 0)

Move = true;

else

Move = false;

}

else

if ((ColumnFirst + 1 < 8) &&

(RowFirst == RowSecond) && (ColumnSecond == ColumnFirst + 1) && (Table[RowSecond, ColumnSecond] == 0))

{

//When Destination is The Empty Return Validity Else Return Not Validity.

if (Table[RowSecond, ColumnSecond] == 0)

Move = true;

else

Move = false;

}

else//Hit Gray Soldier Rulments.

if ((ColumnFirst + 1 < 8) && ColumnSecond == ColumnFirst + 1)

{

if ((RowSecond - 1 < 8) &&

(RowFirst == RowSecond - 1) && (ExistInDestinationEnemy || IgnoreSelfObject))

{

Move = true;

}

if ((RowSecond + 1 < 8) &&

(RowFirst == RowSecond + 1) && (ExistInDestinationEnemy || IgnoreSelfObject))

{

Move = true;

}

}

}

catch (Exception t)

{

Log(t);

}

}

else//Brown int.

if (Order == -1 && Table[RowFirst, ColumnFirst] < 0)

{

//Depend Of First Move do For Positivism

try

{

if ((ColumnSecond + 2 < 8) && (ColumnSecond + 1 < 8) &&

(RowFirst == RowSecond) && (ColumnFirst == ColumnSecond + 2) && (Table[RowSecond, ColumnSecond + 1] == 0)

)

{

//When Destination is The Empty Return Validity Else Return Not Validity.

if (Table[RowSecond, ColumnSecond] == 0)

Move = true;

else

Move = false;

}

else

if ((ColumnSecond + 1 < 8) &&

(RowFirst == RowSecond) && (ColumnFirst == ColumnSecond + 1) && (Table[RowSecond, ColumnSecond] == 0))

{

//When Destination is The Empty Return Validity Else Return Not Validity.

if (Table[RowSecond, ColumnSecond] == 0)

Move = true;

else

Move = false;

}

else//Hit Condition Enemy Movments.

if ((ColumnSecond + 1 < 8) && ColumnFirst == ColumnSecond + 1)

{

if ((RowSecond + 1 < 8) &&

(RowFirst == RowSecond + 1) && (ExistInDestinationEnemy || IgnoreSelfObject))

//Return Validity.

Move = true;

if ((RowSecond - 1 >= 0) &&

(RowFirst == RowSecond - 1) && (ExistInDestinationEnemy || IgnoreSelfObject))

//Return Validity.

Move = true;

}

}

catch (Exception t)

{

Log(t);

}

}

}

else//If Soldeior Moved Previously.

{

//For Gray int.

if (Order == 1 && Table[RowFirst, ColumnFirst] > 0)

{

//Depend on Second Move do For Land Of Islam

try

{

if ((ColumnFirst + 1 < 8) &&

(RowFirst == RowSecond) && (ColumnSecond == ColumnFirst + 1) && (Table[RowSecond, ColumnSecond] == 0))

{

//When Destination is The Empty Return Validity Else Return Not Validity.

if (Table[RowSecond, ColumnSecond] == 0)

Move = true;

else

Move = false;

}

else//Hit Gray Soldier Rulments.

if ((ColumnFirst + 1 < 8) && ColumnSecond == ColumnFirst + 1)

{

if ((RowSecond - 1 < 8) &&

(RowFirst == RowSecond - 1) && (ExistInDestinationEnemy || IgnoreSelfObject))

{

Move = true;

}

if ((RowSecond + 1 < 8) &&

(RowFirst == RowSecond + 1) && (ExistInDestinationEnemy || IgnoreSelfObject))

{

Move = true;

}

}

}

catch (Exception t)

{

Log(t);

}

}

else//Brown int.

if (Order == -1 && Table[RowFirst, ColumnFirst] < 0)

{

//Depend Of Second Move do For Positivism Land

try

{

if ((ColumnSecond + 1 < 8) &&

(RowFirst == RowSecond) && (ColumnFirst == ColumnSecond + 1) && (Table[RowSecond, ColumnSecond] == 0))

{

//When Destination is The Empty Return Validity Else Return Not Validity.

if (Table[RowSecond, ColumnSecond] == 0)

Move = true;

else

Move = false;

}

else//Hit Condition Enemy Movments.

if ((ColumnSecond + 1 < 8) && ColumnFirst == ColumnSecond + 1)

{

if ((RowSecond + 1 < 8) &&

(RowFirst == RowSecond + 1) && (ExistInDestinationEnemy || IgnoreSelfObject))

//Return Validity.

Move = true;

if ((RowSecond - 1 >= 0) &&

(RowFirst == RowSecond - 1) && (ExistInDestinationEnemy || IgnoreSelfObject))

//Return Validity.

Move = true;

}

}

catch (Exception t)

{

Log(t);

}

}

}

return Move;

}

//Solder Rule Method.

public bool SoldierRules(int RowFirst, int ColumnFirst, int RowSecond, int ColumnSecond, bool NotMoved, Color color, bool ExistInDestinationEnemy)

{

if (!(ArrangmentsBoard))

{

return SoldierRulesaArrangmentsBoardZero(RowFirst, ColumnFirst, RowSecond, ColumnSecond, NotMoved, color, ExistInDestinationEnemy);

}

else

{

return SoldierRulesaArrangmentsBoardOne(RowFirst, ColumnFirst, RowSecond, ColumnSecond, NotMoved, color, ExistInDestinationEnemy);

}

/\*if (Move)

{

if (AchmazCheckByMoveByRule(Table, RowFirst, ColumnFirst, RowSecond, ColumnSecond, Order))

Move = false;

}

\*/

///Return Not Validity.

}

}

}

//End of Documentation.