# Animal Kingdom Project Report

## Abstract

This report details the development of the Animal Kingdom simulation project. The project required implementing behaviors for specific animal types within a simulated world. The behaviors were defined through subclassing a given superclass, `Critter`, to adhere to object-oriented principles such as inheritance and polymorphism. This report outlines the project goals, design approach, implementation, and results observed during testing.

## Introduction

The Animal Kingdom project is an exploration of class inheritance and object-oriented programming principles. A `Critter` superclass was provided, with a requirement to create five subclasses: `Bear`, `Tiger`, `WhiteTiger`, `Giant`, and `NinjaCat`. Each subclass implements distinct behaviors, colors, and movement patterns, adhering to specific rules.  
  
The primary goal was to model these behaviors programmatically, ensuring modularity, reusability, and alignment with provided specifications. This project demonstrates practical coding skills in Java and the ability to integrate multiple components within a pre-built simulation framework.

## Project Description

The project operates within a simulation environment where various `Critter` subclasses interact. The primary functionalities of each critter subclass are determined by overriding three methods:  
1. `getMove(CritterInfo info)`: Defines the action taken by the critter during its turn.  
2. `getColor()`: Specifies the color of the critter.  
3. `toString()`: Provides a string representation of the critter for visualization.  
  
The following five subclasses were implemented:  
- \*\*`Bear`\*\*: Alternates between `/` and `\` symbols and follows simple movement logic.  
- \*\*`Tiger`\*\*: Cycles through random colors and behaves dynamically in response to its surroundings.  
- \*\*`WhiteTiger`\*\*: A specialized Tiger with additional infection-related behavior.  
- \*\*`Giant`\*\*: Displays a cycling sequence of strings and prioritizes infection or wall-hugging movement.  
- \*\*`NinjaCat`\*\*: Implements unique movement and infection strategies with distinct visuals.  
  
The supporting framework files (`CritterMain`, `CritterModel`, `CritterPanel`) manage simulation setup, rendering, and interactions between critters.

## Project Design

#### Class Descriptions

- \*\*`Critter` (Superclass)\*\*: Defines default behavior for critters and provides a base for subclasses to override.  
- \*\*`Bear`\*\*: Alternates display between `/` and `\` every move. Moves by infecting enemies, hopping forward, or turning left. Returns `Color.WHITE` for polar bears and `Color.BLACK` otherwise.  
- \*\*`Tiger`\*\*: Cycles colors (`RED`, `GREEN`, `BLUE`) every three moves. Displays as `TGR`. Prioritizes infecting enemies, avoids walls, and moves dynamically.  
- \*\*`WhiteTiger`\*\*: Behaves like a Tiger but is always white (`Color.WHITE`). Changes its string from `tgr` to `TGR` upon infecting another critter.  
- \*\*`Giant`\*\*: Displays a sequence of `fee`, `fie`, `foe`, `fum` in six-move intervals. Infects enemies in front, hops forward if possible, and turns right otherwise.  
- \*\*`NinjaCat`\*\*: Displays `z` initially and `Z` after infecting another critter. Dynamically changes color based on infection status. Implements custom movement logic sensitive to its environment.

## Program Workflow and Logic

1. \*\*Initialization\*\*:  
 - `CritterMain` creates instances of each critter subclass and initializes the simulation world.  
 - Each critter is added to the simulation grid, which is enclosed by walls.  
  
2. \*\*Simulation Cycle\*\*:  
 - The framework calls the `getMove` method for each critter to determine its action (hop, turn, infect).  
 - Critters interact based on their position and the surrounding environment.  
  
3. \*\*Rendering\*\*:  
 - `CritterPanel` visualizes critters using their string representation and color attributes.  
 - Debug mode displays directional arrows for critters.

## Development Details

#### Key Implementation Points

- \*\*Bear\*\*: Tracks moves using an internal counter to alternate its appearance. Maintains a boolean to differentiate polar bears.  
- \*\*Tiger\*\*: Uses a randomization mechanism for color selection, resetting every three moves. Handles movement adaptively to avoid clustering and walls.  
- \*\*WhiteTiger\*\*: Tracks infection state internally to determine when to change its string representation.  
- \*\*Giant\*\*: Implements a modular counter to cycle through text representations at fixed intervals.  
- \*\*NinjaCat\*\*: Uses custom logic for infection and movement, dynamically adapting its appearance.

## Results

The simulation was tested with each critter type individually and collectively. Key observations:  
- \*\*Bears\*\*: Alternated between `/` and `\` correctly. Displayed proper wall-following behavior.  
- \*\*Tigers\*\*: Changed colors correctly every three moves and reacted dynamically to walls and other critters.  
- \*\*White Tigers\*\*: Transitioned from `tgr` to `TGR` upon infection. Behaved identically to Tigers otherwise.  
- \*\*Giants\*\*: Cycled through `fee`, `fie`, `foe`, and `fum` correctly. Displayed clockwise wall-hugging behavior.  
- \*\*NinjaCats\*\*: Adapted their color and string dynamically. Exhibited unique and varied movement patterns.

## Discussion

#### Strengths

- Leveraged inheritance to simplify the addition of new critters.  
- Modular class design ensures maintainability and extensibility.  
- The simulation provided real-time feedback on critter behaviors.

#### Limitations

- Behavior logic was tightly coupled to specific requirements, limiting flexibility for new features.  
- Randomization introduced minor inconsistencies in Tiger color patterns during short runs.

#### Future Improvements

- Allow user-defined critter classes for enhanced customization.  
- Optimize the framework for larger grid sizes and higher critter counts.

## Conclusion

This project highlights the power of object-oriented design in creating a dynamic, interactive system. The use of inheritance and polymorphism enabled the seamless addition of critters with distinct behaviors. The simulation effectively demonstrated the intended functionalities, showcasing modularity, randomness, and reusability in the codebase.

## References

- Project specification document.  
- Java API documentation for `Color` and `Random` classes.  
- Course materials on object-oriented programming concepts.