Example 1:

```
public void checkAvailability(String phoneNumber, MyDate startDate, MyDate endDate)
{
  // Dominating Term Analysis:
  // Outer loop iterates over reservations: O(n) where n is the number of reservations.
  // Inside the loop, we perform constant-time operations like checking phone number and date comparisons: O(1).
  // Overall time complexity: O(n).


  for (Reservation reservation : reservations) // O(n)
  {
    if (reservation.getPet().getCustomer().getPhoneNumber().equals(phoneNumber)) // O(1)
    {
      // Checking startDate conflicts: O(1)
      if (reservation.getStartDate().isBefore(startDate) && reservation.getEndDate().isAfter(startDate))
      {
        throw new IllegalArgumentException("Pet is already reserved for that date.");
      }
      // Checking endDate conflicts: O(1)
      if (reservation.getStartDate().isBefore(endDate) && reservation.getEndDate().isAfter(endDate))
      {
        throw new IllegalArgumentException("Pet is already reserved for that date.");
      }
    }
  }
}
```

```
// Optimization Suggestion:
// 1. **Use a Map for Efficient Lookup**:
//    - Organize reservations by customer phone number in a HashMap<String, List<Reservation>>.
//    - Time Complexity for lookup by phone number reduces to O(1) for the map and O(m) for their reservations (m = reservations for a customer).
// 2. **Merge Date Conditions**:
//    - Simplify overlapping date logic using a single condition for clarity and efficiency.
```

Optimized Version:

```
public void checkAvailability(String phoneNumber, MyDate startDate, MyDate endDate)
{
  List<Reservation> customerReservations = reservationMap.get(phoneNumber); // O(1)
  if (customerReservations == null) return;

  for (Reservation reservation : customerReservations) // O(m)
  {
    if (reservation.getEndDate().isAfter(startDate) && reservation.getStartDate().isBefore(endDate)) // O(1)
    {
      throw new IllegalArgumentException("Pet is already reserved for that date.");
    }
  }
}
```

Optimized Complexity:

- Lookup by phone number: O(1)O(1)O(1).

- Check overlapping dates: O(m)O(m)O(m), where mmm is the number of reservations for that customer.
- **Overall: O(1+m)O(1 + m)O(1+m)**.

Example 2:

```
public void savePets(PetList pets)
{
  try
  {
    // Writes the PetList object to a binary file
    MyFileHandler.writeToBinaryFile(fileName, pets); // Time complexity depends on the size of PetList, O(n)
                                                      // where n is the number of pets in the list.
  }
  catch (FileNotFoundException e)
  {
    System.out.println("File not found or could not be opened pet"); // O(1)
  }
  catch (IOException e)
  {
    System.out.println("IO Error writing to file pet"); // O(1)
  }
}
```

- **Write Operation**: The dominant operation is MyFileHandler.writeToBinaryFile(). Its complexity depends on:
    - The number of pets in PetList (nnn).
    - Serialization cost of each pet (kkk), which may involve converting each pet's data into binary format.
    - Overall complexity of writeToBinaryFile: O(n·k)O(n.k)O(n·k).
- **Catch Blocks**: Printing error messages in the catch blocks is O(1)O(1)O(1) and insignificant compared to the file write operation.

Dominating Term Analysis

The dominating term is O(n·k)O(n.k)O(n·k), where nnn is the number of pets in the list, and kkk is the cost of serializing a single pet.

Optimization Suggestions

1. **Buffering the Write Operation**:
    - Ensure the writeToBinaryFile method uses a buffered output stream to optimize the write operation and reduce disk I/O overhead.

Optimized Code Example

```
public void savePets(PetList pets)
{
 try
 {
  // Optimized writing using buffering or delta saving
  MyFileHandler.writeToBinaryFile(fileName, pets); // Optimized O(m * k), where m <= n
 }
 catch (FileNotFoundException e)
 {
  System.out.println("File not found or could not be opened pet"); // O(1)
 }
 catch (IOException e)
 {
  System.out.println("IO Error writing to file pet"); // O(1)
 }
}
```

Ensures efficiency with larger datasets and frequent writes.