November 7 2023

JAVA CRASH COURSE

INFO113 Final Exam

PRESENTED BY NATHAN LAI

Agenda

1. **Interfaces**
2. **Anonymous Classes**
3. **Lambdas**
4. **Generics**
5. **Wildcards**
6. **Past Exam Questions**

Interfaces

- An interface is like a contract that defines a set of methods that a class must implement.
- Ensure that classes have certain behaviour.
- Promote code abstraction, support multiple inheritance, and enhance code reusability.

Interfaces

- An interface is like a contract that defines a set of methods that a class must implement.
- Ensure that classes have certain behaviour.
- Promote code abstraction, support multiple inheritance, and enhance code reusability.

- Classes that implement an interface must provide concrete implementations for the interface's methods.

```
public interface Shape {
        public double area();
}
```

Interfaces

- An interface is like a contract that defines a set of methods that a class must implement.
- Ensure that classes have certain behaviour.
- Promote code abstraction, support multiple inheritance, and enhance code reusability.

- Classes that implement an interface must provide concrete implementations for the interface's methods.

```
public interface Shape {
        public double area();
}
```

Any class that implements the Shape interface must provide a definition for the area() method.

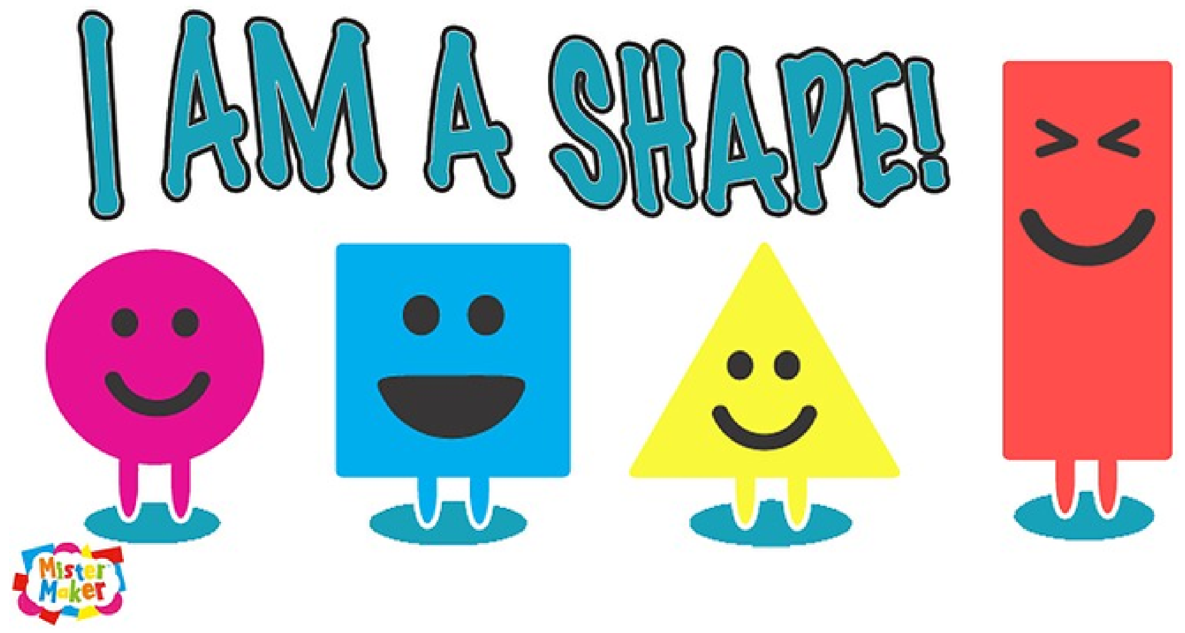
Compiler when you don't define Area()
Compiler when you define Area()

Interfaces - Default Methods

- Default methods were added to interfaces in Java 8.
- Allow interfaces to define behaviours of methods.

Demo - Implementing the Shape interface

```java
public interface Shape {
        public double area();
        public default void Greet(){
                System.out.println("I am a shape!");
        }
}
```



Anonymous Classes

- An anonymous class is immediately constructed and an instance is returned to the caller.
- Typically used for creating a single instance of a class that extends another class or implements an interface.
- I.e. when you need to define a small, one-off class for a specific purpose without the need to create a separate named class.

Anonymous Classes

- An anonymous class is immediately constructed and an instance is returned to the caller.
- Typically used for creating a single instance of a class that extends another class or implements an interface.
- I.e. when you need to define a small, one-off class for a specific purpose without the need to create a separate named class.

Anonymous Classes

- An anonymous class is immediately constructed and an instance is returned to the caller.
- Typically used for creating a single instance of a class that extends another class or implements an interface.
- I.e. when you need to define a small, one-off class for a specific purpose without the need to create a separate named class.

```
Shape inkBlot = new Shape(){
    public double area(){
    return 14.5;
    }
};
```

Lambdas

- Lambda methods require an interface that declare only one abstract (non-default) method.

- Syntax:
  (parameters) -> expression

Lambdas

- Lambda methods require an interface that declare only one abstract (non-default) method.

- Syntax:
  (parameters) -> expression

```
Shape inkBlot = () -> {return 14.5;};
```

Lambdas

- Lambda methods require an interface that declare only one abstract (non-default) method.

- Syntax:
  (parameters) -> expression

Demo - Binary Operator Lambdas

AREA = 14.5

Generics

- Generics allow us to parameterise types in a class.
- Type parameters are defined by angular brackets containing an identifier.

Generics

- Generics allow us to parameterise types in a class.
- Type parameters are defined by angular brackets containing an identifier.

Demo - Generic Container

Generics

- Generics allow us to parameterise types in a class.
- Type parameters are defined by angular brackets containing an identifier.

- A generic class you may be familiar with:

```
public class ArrayList<E>
```

```
import java.uitil.ArrayList;

ArrayList<Integer> myNumbers = new Arraylist<Integer>();

ArrayList<String> myStrings = new Arraylist<String>();
```

Generics

- Generics allow us to parameterise types in a class.
- Type parameters are defined by angular brackets containing an identifier.

- A generic class you may be familiar with:

```
public class ArrayList<E>
```

```
import java.uitil.ArrayList;

ArrayList<Integer> myNumbers = new Arraylist<Integer>();

ArrayList<String> myStrings = new Arraylist<String>();
```

In this case, generics save us the trouble of needing to use a different ArrayList class for every type.

Wildcards

- Wildcards are used in the context of generics to represent an unknown type or a range of possible types.
- There are three main types of wildcards in Java:

Wildcards

- Wildcards are used in the context of generics to represent an unknown type or a range of possible types.
- There are three main types of wildcards in Java:

1. Unbounded Wildcard
   - Denoted by a question mark (?).
   - It represents an unknown type, meaning that you can use it to accept any type argument.

```java
public void printList(List<?> list) {
    for (Object element : list) {
        System.out.println(element);
    }
}
```

Wildcards

- Wildcards are used in the context of generics to represent an unknown type or a range of possible types.
- There are three main types of wildcards in Java:

1. Unbounded Wildcard

- Denoted by a question mark (?).
- It represents an unknown type, meaning that you can use it to accept any type argument.

```
public void printList(List<?> list) {
    for (Object element : list) {
        System.out.println(element);
    }
}
```

2. Upper bounded Wildcard

- Specifies constraint on the type parameter using the "extends" keyword.

```
public double sum(List<? extends Number> list) {
    double total = 0.0;
    for (Number number : list) {
        total += number.doubleValue();
    }
    return total;
}
```

Wildcards

- Wildcards are used in the context of generics to represent an unknown type or a range of possible types.
- There are three main types of wildcards in Java:

1. Unbounded Wildcard
   - Denoted by a question mark (?).
   - It represents an unknown type, meaning that you can use it to accept any type argument.

```
public void printList(List<?> list) {
    for (Object element : list) {
        System.out.println(element);
    }
}
```

2. Upper bounded Wildcard
   - Specifies constraint on the type parameter using the "extends" keyword.

```
public double sum(List<? extends Number> list) {
    double total = 0.0;
    for (Number number : list) {
        total += number.doubleValue();
    }
    return total;
}
```

3. Lower bounded Wildcard
   - Specifies constraint on the type parameter using the "super" keyword.

```
public void addIntegers(List<? super Integer> list, int n) {
    list.add(n);
}
```

Practice Questions

What will be the output of the following program?

```java
import java.util.*;

class Liquid {
   String name;
   double price;

   Liquid(String name, double price) {
      this.name = name;
      this.price = price;
   }

   public String toString() {
      return name + " " + price;
   }
}

class Water extends Liquid {
   public Water(String name, double price) {
      super(name, price);
   }
}

class MineralWater extends Water {
   MineralWater(String name, double price) {
      super(name, price);
   }
}

class Milk extends Liquid {
   Milk(String name, double price) {
      super(name, price);
   }
}

public class Wildcard {

   public static void deleteLiquid(List<? extends Water> list, Water water) {
      list.remove(water);
   }

   public static void addLiquid(List<? super MineralWater> list) {
      list.add(new MineralWater("Mineral Water", 10));
   }

   public static void print(List<?> list) {
      for (Object item : list)
         System.out.println(item);
   }

   public static void main(String[] args) {

      List<Liquid> LiquidList= new ArrayList<Liquid>();
      List<MineralWater> MineralWaterList= new ArrayList<MineralWater>();

      addLiquid(LiquidList);

      addLiquid(MineralWaterList);
      addLiquid(MineralWaterList);

      Water Liquid = MineralWaterList.get(0);

      deleteLiquid(MineralWaterList, Liquid);

      print(MineralWaterList);
   }
}
```

Practice Questions

What will be the output of the following program?

```java
import java.util.*;

class Liquid {
   String name;
   double price;

   Liquid(String name, double price) {
      this.name = name;
      this.price = price;
   }

   public String toString() {
      return name + " " + price;
   }
}

class Water extends Liquid {
   public Water(String name, double price) {
      super(name, price);
   }
}

class MineralWater extends Water {
   MineralWater(String name, double price) {
      super(name, price);
   }
}

class Milk extends Liquid {
   Milk(String name, double price) {
      super(name, price);
   }
}

public class Wildcard {

   public static void deleteLiquid(List<? extends Water> list, Water water) {
      list.remove(water);
   }

   public static void addLiquid(List<? super MineralWater> list) {
      list.add(new MineralWater("Mineral Water", 10));
   }

   public static void print(List<?> list) {
      for (Object item : list)
         System.out.println(item);
   }

   public static void main(String[] args) {

      List<Liquid> LiquidList= new ArrayList<Liquid>();
      List<MineralWater> MineralWaterList= new ArrayList<MineralWater>();

      addLiquid(LiquidList);

      addLiquid(MineralWaterList);
      addLiquid(MineralWaterList);

      Water Liquid = MineralWaterList.get(0);

      deleteLiquid(MineralWaterList, Liquid);

      print(MineralWaterList);
   }
}
```

Create empty lists

Practice Questions

What will be the output of the following program?

```java
import java.util.*;

class Liquid {
   String name;
   double price;

   Liquid(String name, double price) {
      this.name = name;
      this.price = price;
   }

   public String toString() {
      return name + " " + price;
   }
}

class Water extends Liquid {
   public Water(String name, double price) {
      super(name, price);
   }
}

class MineralWater extends Water {
   MineralWater(String name, double price) {
      super(name, price);
   }
}

class Milk extends Liquid {
   Milk(String name, double price) {
      super(name, price);
   }
}

public class Wildcard {

   public static void deleteLiquid(List<? extends Water> list, Water water) {
      list.remove(water);
   }

   public static void addLiquid(List<? super MineralWater> list) {
      list.add(new MineralWater("Mineral Water", 10));
   }

   public static void print(List<?> list) {
      for (Object item : list)
         System.out.println(item);
   }

   public static void main(String[] args) {

      List<Liquid> LiquidList= new ArrayList<Liquid>();
      List<MineralWater> MineralWaterList= new ArrayList<MineralWater>();

      addLiquid(LiquidList);

      addLiquid(MineralWaterList);
      addLiquid(MineralWaterList);

      Water Liquid = MineralWaterList.get(0);

      deleteLiquid(MineralWaterList, Liquid);

      print(MineralWaterList);
   }
}
```

LiquidList

MineralWaterList

Create empty lists

Practice Questions

What will be the output of the following program?

```java
import java.util.*;

class Liquid {
   String name;
   double price;

   Liquid(String name, double price) {
      this.name = name;
      this.price = price;
   }

   public String toString() {
      return name + " " + price;
   }
}

class Water extends Liquid {
   public Water(String name, double price) {
      super(name, price);
   }
}

class MineralWater extends Water {
   MineralWater(String name, double price) {
      super(name, price);
   }
}

class Milk extends Liquid {
   Milk(String name, double price) {
      super(name, price);
   }
}
```

```java

public class Wildcard {

   public static void deleteLiquid(List<? extends Water> list, Water water) {
      list.remove(water);
   }

   public static void addLiquid(List<? super MineralWater> list) {
      list.add(new MineralWater("Mineral Water", 10));
   }
```

Add mineral water with cost 10 to list

```java

   public static void print(List<?> list) {
      for (Object item : list)
         System.out.println(item);
   }

   public static void main(String[] args) {

      List<Liquid> LiquidList= new ArrayList<Liquid>();
      List<MineralWater> MineralWaterList= new ArrayList<MineralWater>();

      addLiquid(LiquidList);
```

Called once on LiquidList

```java

      addLiquid(MineralWaterList);
      addLiquid(MineralWaterList);
```

Called twice on MineralWaterList

```java

      Water Liquid = MineralWaterList.get(0);

      deleteLiquid(MineralWaterList, Liquid);

      print(MineralWaterList);
   }
}
```

LiquidList

0. Mineral Water, 10

MineralWaterList

0. Mineral Water, 10

1. Mineral Water, 10

Practice Questions I

What will be the output of the following program?

```java
import java.util.*;

class Liquid {
   String name;
   double price;

   Liquid(String name, double price) {
      this.name = name;
      this.price = price;
   }

   public String toString() {
      return name + " " + price;
   }
}

class Water extends Liquid {
   public Water(String name, double price) {
      super(name, price);
   }
}

class MineralWater extends Water {
   MineralWater(String name, double price) {
      super(name, price);
   }
}

class Milk extends Liquid {
   Milk(String name, double price) {
      super(name, price);
   }
}

public class Wildcard {

   public static void deleteLiquid(List<? extends Water> list, Water water) {
      list.remove(water);
   }

   public static void addLiquid(List<? super MineralWater> list) {
      list.add(new MineralWater("Mineral Water", 10));
   }

   public static void print(List<?> list) {
      for (Object item : list)
         System.out.println(item);
   }

   public static void main(String[] args) {

      List<Liquid> LiquidList= new ArrayList<Liquid>();
      List<MineralWater> MineralWaterList= new ArrayList<MineralWater>();

      addLiquid(LiquidList);

      addLiquid(MineralWaterList);
      addLiquid(MineralWaterList);

      Water Liquid = MineralWaterList.get(0);

      deleteLiquid(MineralWaterList, Liquid);

      print(MineralWaterList);
   }
}
```

LiquidList

0. Mineral Water, 10

MineralWaterList

0. Mineral Water, 10

1. Mineral Water, 10

Liquid = Mineral Water, 10

Remove first instance of Mineral Water, 10 from MineralWaterList

JAVA CRASH COURSE

Practice Questions I

What will be the output of the following program?

```java
import java.util.*;

class Liquid {
    String name;
    double price;

    Liquid(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public String toString() {
        return name + " " + price;
    }
}

class Water extends Liquid {
    public Water(String name, double price) {
        super(name, price);
    }
}

class MineralWater extends Water {
    MineralWater(String name, double price) {
        super(name, price);
    }
}

class Milk extends Liquid {
    Milk(String name, double price) {
        super(name, price);
    }
}

public class Wildcard {

    public static void deleteLiquid(List<? extends Water> list, Water water) {
        list.remove(water);
    }

    public static void addLiquid(List<? super MineralWater> list) {
        list.add(new MineralWater("Mineral Water", 10));
    }

    public static void print(List<?> list) {
        for (Object item : list)
            System.out.println(item);
    }

    public static void main(String[] args) {

        List<Liquid> LiquidList= new ArrayList<Liquid>();
        List<MineralWater> MineralWaterList= new ArrayList<MineralWater>();

        addLiquid(LiquidList);

        addLiquid(MineralWaterList);
        addLiquid(MineralWaterList);

        Water Liquid = MineralWaterList.get(0);

        deleteLiquid(MineralWaterList, Liquid);

        print(MineralWaterList);
    }
}
```

LiquidList

0. Mineral Water, 10

MineralWaterList

~~0. Mineral Water, 10~~

0. Mineral Water, 10

Liquid = Mineral Water, 10

Remove first instance of Mineral Water, 10 from MineralWaterList

Practice Questions I

What will be the output of the following program?

```java
import java.util.*;

class Liquid {
   String name;
   double price;

   Liquid(String name, double price) {
      this.name = name;
      this.price = price;
   }

   public String toString() {
      return name + " " + price;
   }
}

class Water extends Liquid {
   public Water(String name, double price) {
      super(name, price);
   }
}

class MineralWater extends Water {
   MineralWater(String name, double price) {
      super(name, price);
   }
}

class Milk extends Liquid {
   Milk(String name, double price) {
      super(name, price);
   }
}

public class Wildcard {

   public static void deleteLiquid(List<? extends Water> list, Water water) {
      list.remove(water);
   }

   public static void addLiquid(List<? super MineralWater> list) {
      list.add(new MineralWater("Mineral Water", 10));
   }

   public static void print(List<?> list) {
      for (Object item : list)
         System.out.println(item);
   }

   public static void main(String[] args) {

      List<Liquid> LiquidList= new ArrayList<Liquid>();
      List<MineralWater> MineralWaterList= new ArrayList<MineralWater>();

      addLiquid(LiquidList);

      addLiquid(MineralWaterList);
      addLiquid(MineralWaterList);

      Water Liquid = MineralWaterList.get(0);

      deleteLiquid(MineralWaterList, Liquid);

      print(MineralWaterList);
   }
}
```

LiquidList

0. Mineral Water, 10

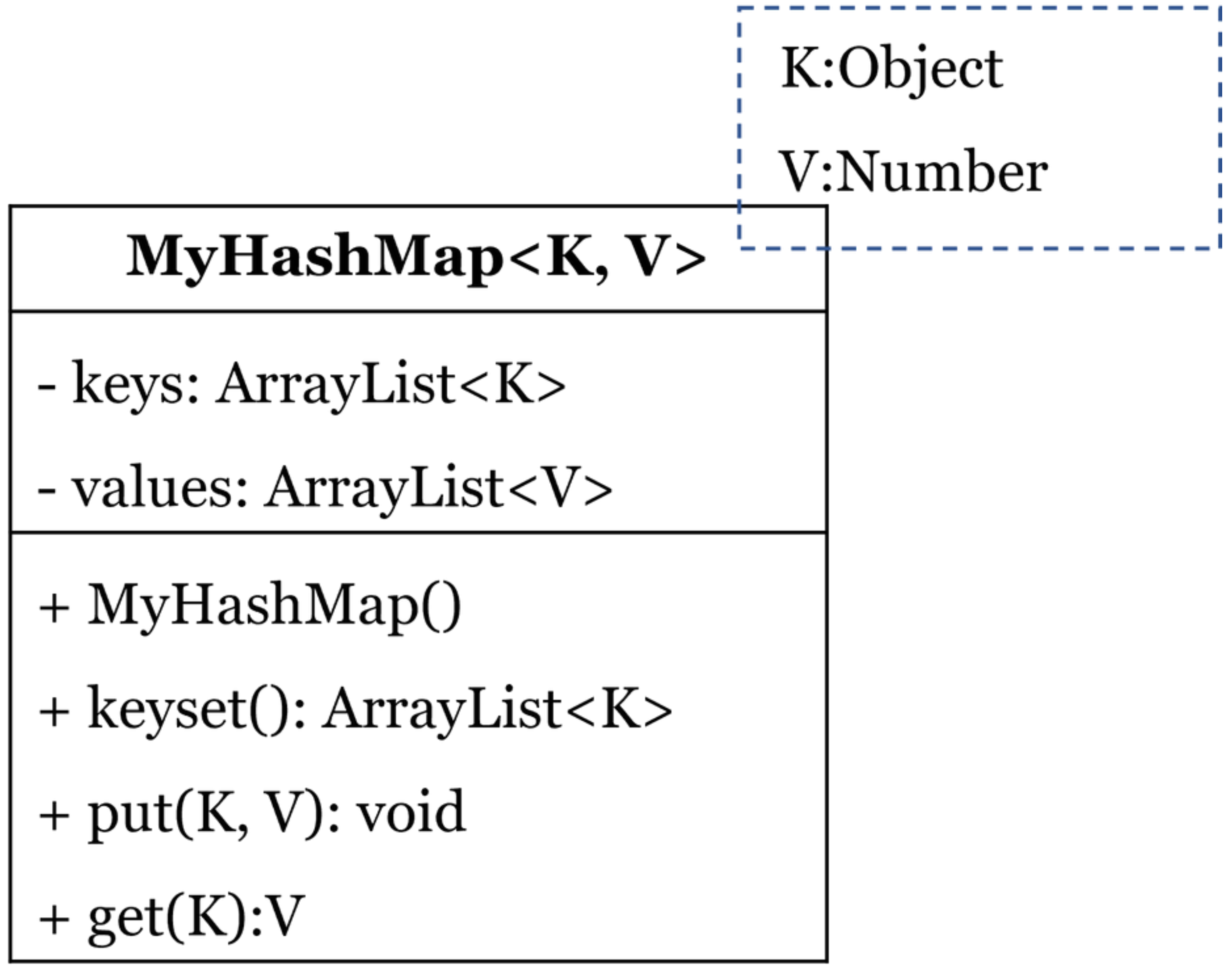
MineralWaterList

$\rightarrow$ 0. Mineral Water, 10

OUTPUT: Mineral Water 10

Practice Questions II

Create a generic HashMap class called MyHashMap.



MyHashMap<K, V> has 2 type parameters

- K for keys of the map. Note that K extends from Object
- V for the value of keys. Note that V extends from Number

MyHashMap<K, V> has following 2 attributes and a constructor to initialize these attributes:

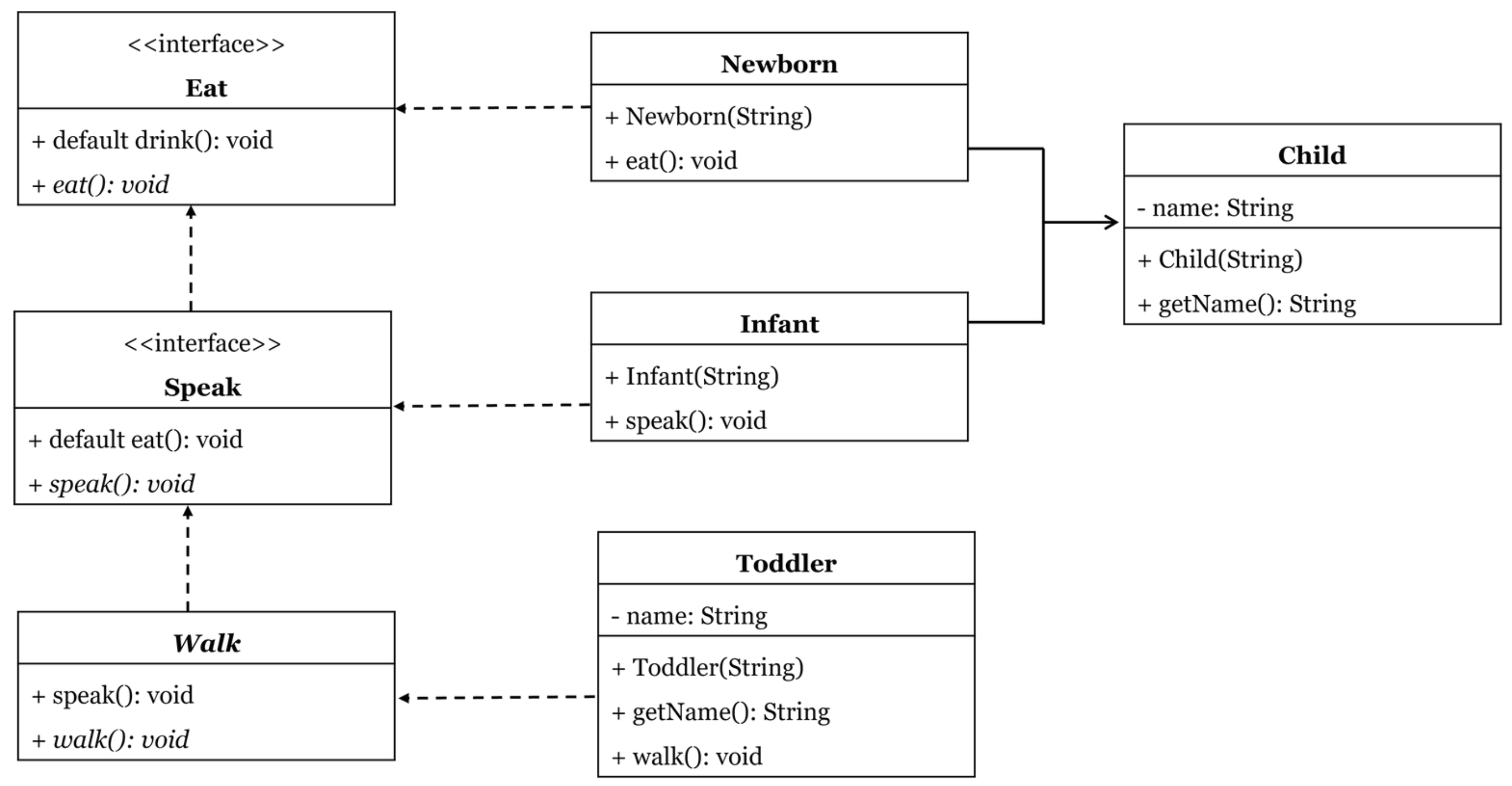
- ArrayList<K> to store the keys
- ArrayList<V> to store the values

MyHashMap<K, V> has 3 methods:

- keyset()
  - It returns the arraylist of keys
- put(K key, V value)
  - It adds the key and its corresponding values in the arraylist. However, if the key already exists in the list, then the old value of this key will be replaced by the new value
- get(K key)
  - It returns the corresponding value of a key. If the key does not exist, return null.

Practice Questions III

Consider the UML:



Create the following:

i) 2 interfaces:

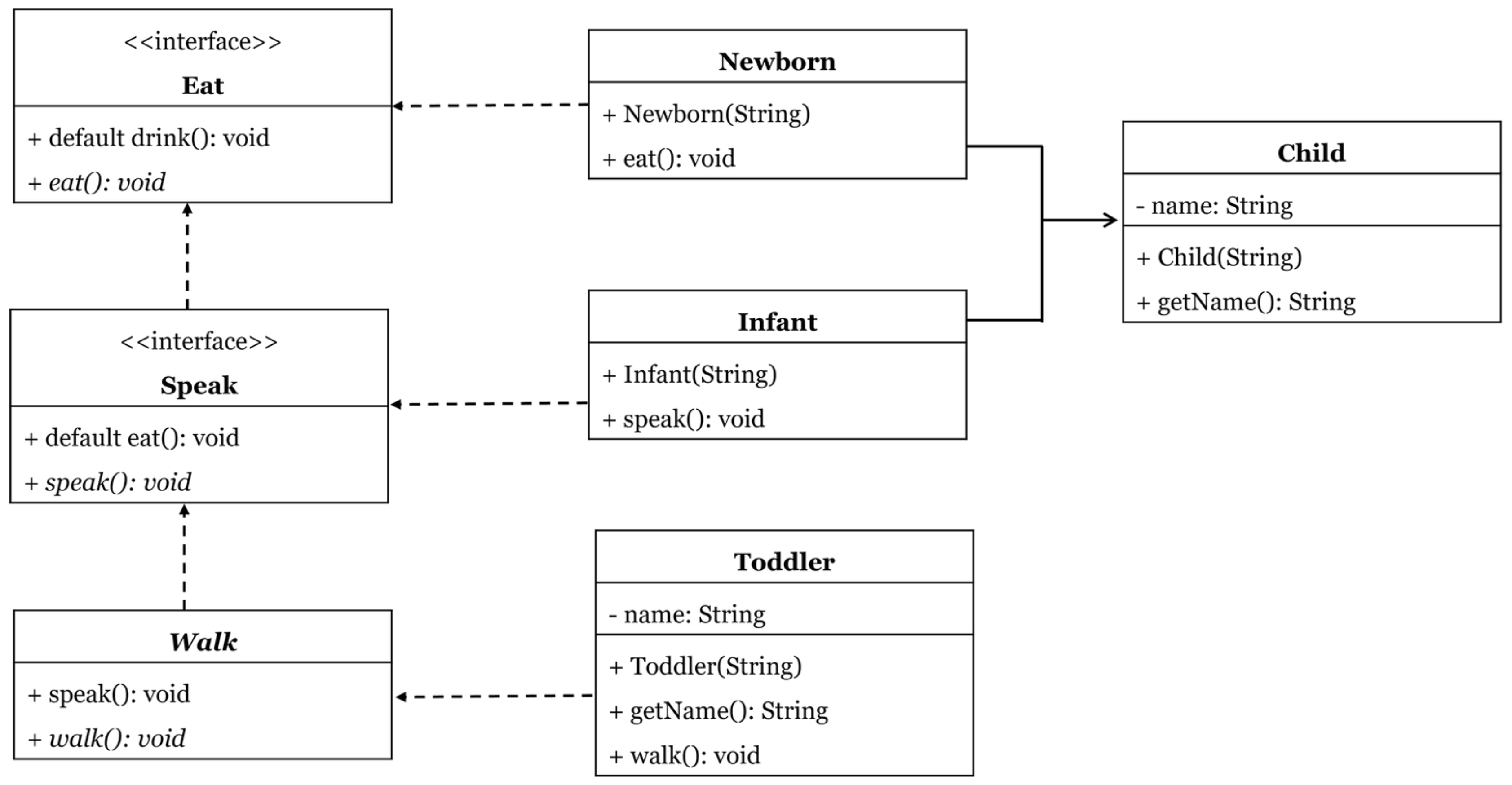
- Eat
  - It contains 1 default method called drink() which simply prints "I drink milk"
  - It contains 1 abstract method called eat()
- Speak
  - It is an extension of Eat interface
  - It contains 1 default method called eat() which simply prints "I eat solid food now"
  - It contains 1 abstract method called speak()

ii) Abstract class which implements Speak:

- Walk
  - It defines the speak() method to print "I can say my name now"
  - It contains 1 abstract method called walk()

Practice Questions III

Consider the UML:



Create the following:

iii) 4 classes:

- Child class
  - It has 1 private attribute called name. It contains a constructor to initialize the name and a getter method to return the name of the child.
- Newborn class
  - It is a subclass of Child and implements Eat interface
  - It defines the eat() method to print “I can’t eat solid food yet”
- Infant class
  - It is a subclass of Child and implements Speak interface
  - It defines the speak() method to print “I can say bu bu bu”
- Toddler class
  - It is a subclass of Walk
  - It has 1 private attribute called name. It contains a constructor to initialize the name and a getter method to return the name of the child.
  - It defines the walk() method to print “I can now walk”