Advanced Programming Techniques in Java

COSI 12B

Interfaces & ArrayLists

Lecture 15

Class Objectives

- Interfaces (section 9.5)
- `ArrayList` (section 10.1)

Review: Abstract Classes

- An abstract class is a placeholder in a class hierarchy that represents a generic concept
- An abstract class cannot be instantiated
- Why?
  - The use of abstract classes is a design decision; it helps us establish common elements in a class that are too general to instantiate

Review: Abstract Classes

- To declare a class as abstract we use the modifier `abstract` on the class header
- **Syntax**

```
public abstract class <name> {
   // contents
}
```

- **Example**

```
public abstract class Shape{
   // contents
}
```

- If the client code tries to create a `Shape` object, we get a compilation error

```
Cannot instantiate the type Shape
```

Review: Abstract Method

- An abstract method is a method that has just the signature but does not contain implementation

```
public abstract <type> <name>(<type> <name>, ..);
```

- A class declared as `abstract` **does not** need to contain `abstract` methods

Review: Abstract Classes & Constructors

- Can an abstract class have a constructor?
  - An abstract class can have a constructor. You can either explicitly provide a constructor to an abstract class or if you don't, the compiler will add a default constructor

- Why can an abstract class have a constructor?
  - When a class extends an abstract class, the constructor of subclass will invoke the constructor of super class either implicitly or explicitly

Review: Multiple Inheritance

- Java supports single inheritance, meaning that a derived class can have only one parent class
- Multiple inheritance allows a class to be derived from two or more classes, inheriting the members of all parents
- **Java does not support multiple inheritance** because of possible conflicts
  - Which class should `super` refer when child class has multiple parents?
- Alternative: **Interface**
  - Looks like a class
  - It describes what a class does
- You can have a class that **extends** one class and **implements** one or more interfaces

Review: Interface Definition

FORM:

```
public interface interfaceName {
    abstract method headings
    constant declarations
}
```

EXAMPLE:

```
public interface Payable {
    public abstract double calcSalary();
    public abstract boolean salaried();
    public static final
           double DEDUCTIONS = 25.5;
}
```

- The keyword `abstract` is implicit in each *abstract method* definition
- And keywords `static final` are implicit in each *constant* declaration
- As such, they may be omitted

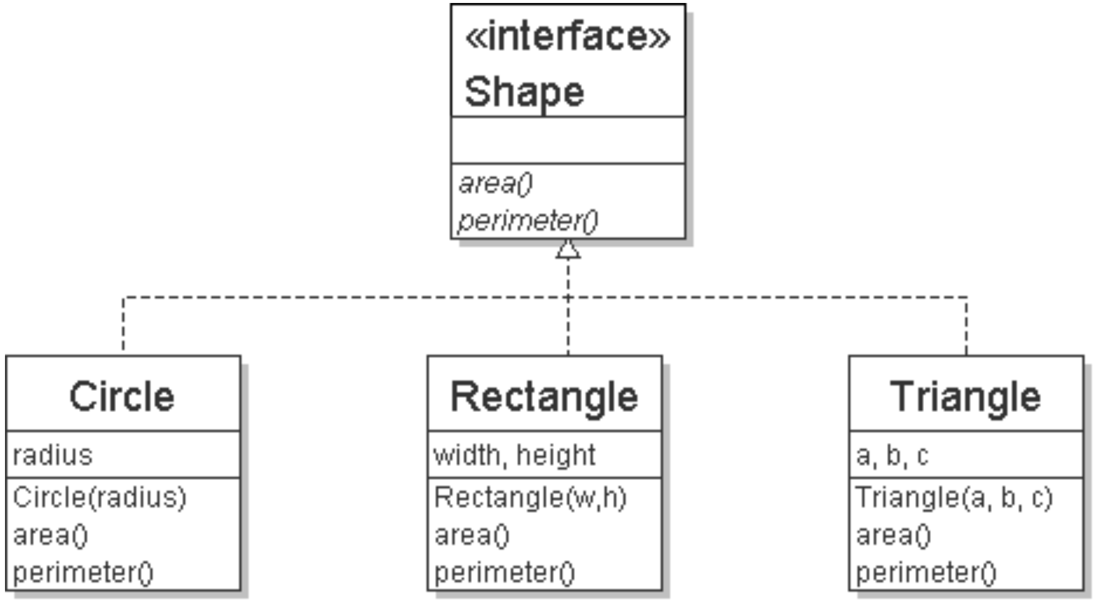
Shape Interface

- An interface for shapes:

```
// A general interface for shape classes
public interface Shape {
    public double area();
    public double perimeter();
}
```

- This interface describes the features common to all shapes (every shape has an area and perimeter)

Diagrams of Interfaces



- We draw arrows upward from the classes to the interface(s) they implement
  - There is a supertype-subtype relationship here: e.g., all Circles are Shapes, but not all Shapes are Circles

Complete Circle class

```
// Represents circle shape
public class Circle implements Shape {
    private double radius;

    // Constructs a new circle with the given radius.
    public Circle(double radius) {
        this.radius = radius;
    }

    // Returns the area of this circle.
    public double area() {
        return Math.PI * radius * radius;
    }

    // Returns the perimeter of this circle.
    public double perimeter() {
        return 2.0 * Math.PI * radius;
    }
}
```

Complete Rectangle class

```
// Represents rectangle shapes.
public class Rectangle implements Shape {
    private double width;
    private double height;

    // Constructs a new rectangle with the given dimensions.
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    // Returns the area of this rectangle.
    public double area() {
        return width * height;
    }

    // Returns the perimeter of this rectangle.
    public double perimeter() {
        return 2.0 * (width + height);
    }
}
```

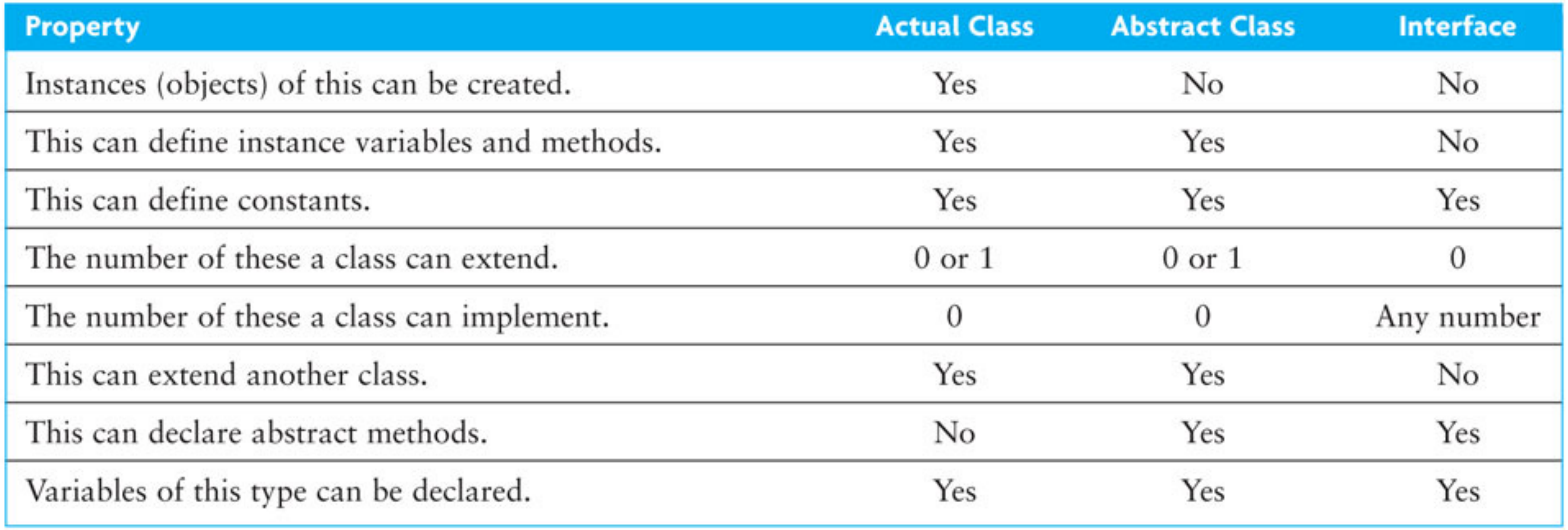
Complete Triangle class

```java
// Represents triangle shapes.
public class Triangle implements Shape {
    private double a;
    private double b;
    private double c;

    // Constructs a new Triangle given side lengths.
    public Triangle(double a, double b, double c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    // Returns this triangle's area using Heron's formula.
    public double area() {
        double s = (a + b + c) / 2.0;
        return Math.sqrt(s * (s - a) * (s - b) * (s - c));
    }

    // Returns the perimeter of this triangle.
    public double perimeter() {
        return a + b + c;
    }
}
```

Inheriting from Interfaces vs. Classes

- A class can *extend* 0 or 1 superclass
- An interface cannot extend a class
- A class can *implement* one or more interfaces
- An interface can extend one or more interfaces

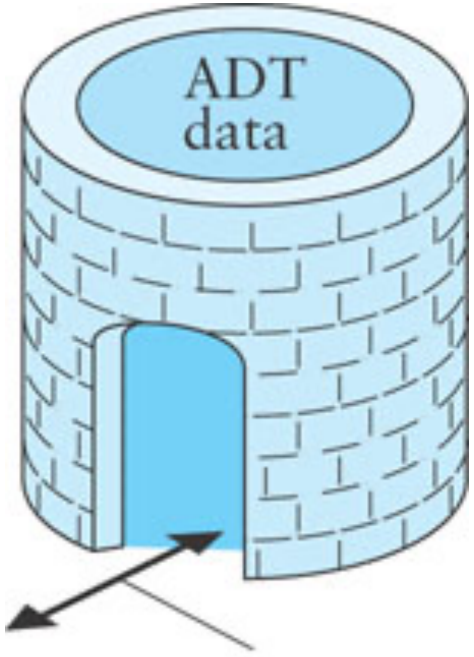
Summary of Features of Actual Classes, Abstract Classes, and Interfaces



Property	Actual Class	Abstract Class	Interface
Instances (objects) of this can be created.	Yes	No	No
This can define instance variables and methods.	Yes	Yes	No
This can define constants.	Yes	Yes	Yes
The number of these a class can extend.	0 or 1	0 or 1	0
The number of these a class can implement.	0	0	Any number
This can extend another class.	Yes	Yes	No
This can declare abstract methods.	No	Yes	Yes
Variables of this type can be declared.	Yes	Yes	Yes

ADTs

- Abstract Data Type (ADT)
- An encapsulation of data and methods
- Allows for reusable code
- The user need not know about the implementation of the ADT
- A user interacts with the ADT using only public methods



ADT operations

Problem

- Write a program that reads a file and displays:
    - First display all words
    - Then display them with all plurals capitalized
    - Then display them in reverse order
    - Then display them with all plural words removed

Naive solution

```
String[] allWords = new String[1000];
int wordCount = 0;

Scanner input = new Scanner(new File("data.txt"));
while (input.hasNext()) {
    String word = input.next();
    allWords[wordCount] = word;
    wordCount++;
}
```

- You don't know how many words the file will have
  - Hard to create an array of the appropriate size
- Luckily, there are other ways to store data besides in an array

Array Limitations

- You need to know in advance the maximum number of elements! What if these changes later?
- What if you want to remove something?
  - You end up with empty elements in the middle of the array
  - You would have to shift the rest of the elements over and be left with empty slots at the end
- We need more flexibility!!
  - “Add something here to the list”
  - “Remove this value from the list”
  - List should grow and shrink and move elements around automatically!

ArrayList

The `ArrayList` class

- An `ArrayList` object uses an array to store its values
- Think of it as an auto-resizing array that can hold any type of object, with many convenient methods
- It maintains most of the benefits of arrays, such as fast random access
- It frees us from some tedious operations on arrays, such as sliding elements and resizing
- To use `ArrayList` remember to import `java.util.*;`
- We can declare arrays of different types e.g., `int[], String[], …` the `ArrayList` class has similar flexibility

The `ArrayList` class

- `ArrayList<E>` is a **generic class**
- The `<E>` is a placeholder in which you write the type of elements you want to store in the `ArrayList`

Java Generics

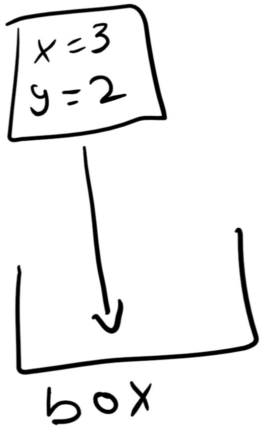
- Used to make an object usable for any types, while still preserving the type checking that Java allows

- Normally we must be specific about the type we’re passing into an object, but Java allows us to make this variable

- Useful for making data structures, which we want to be applicable for any data we want to insert into them

Java Generics

- We want to create a “box” that allows you to put something in it, and then look at what’s in there
- We could make a box for `Points`, and then copy that code for other objects



```
public class PointBox{
   private Point p;
   public void put(Point p){
      this.p = p;
   }
   public Point get( ){
      return this.p;
   }
}
```

```
public class Main{
 public static void main (String[ ] args){
       PointBox pb1 = new PointBox();
       pb1.put(new Point(3,2));
       System.out.println(pb1.get().getY());
 }
```

Java Generics

- We can make this code “generic”

```java
public class PointBox{
   private Point p;
   public void put(Point p){
      this.p = p;
   }
   public Point get( ){
      return this.p;
   }
}
```

```java
public class Box<T>{
   private T object;
   public void put(T object){
      this.object = object;
   }
   public T get( ){
      return this.object;
   }
}
```

- Now we can put an object of any type “T” into the box

How to use this “Generic” Type

- In the `main` method, you can initialize a `Box` of any type by doing the following: `Box<TYPE> name = new Box<TYPE>( );`
  - e.g: `Box<String> stringBox = new Box<String>( );`
  - or: `Box<Point> pointBox = new Box<Point>( );`
- Now our code can be used for any type*!

Example Code

```
public class Main{
  public static void main(String[ ] args){
    Point p2 = new Point(0,5);
    System.out.println("Making a box for points:");
    Box<Point> b1 = new Box<Point>( );
    b1.put(p2);
    System.out.println(b1.get().getY());

  }
}
```

Makes a specific version of the `Box` object for points

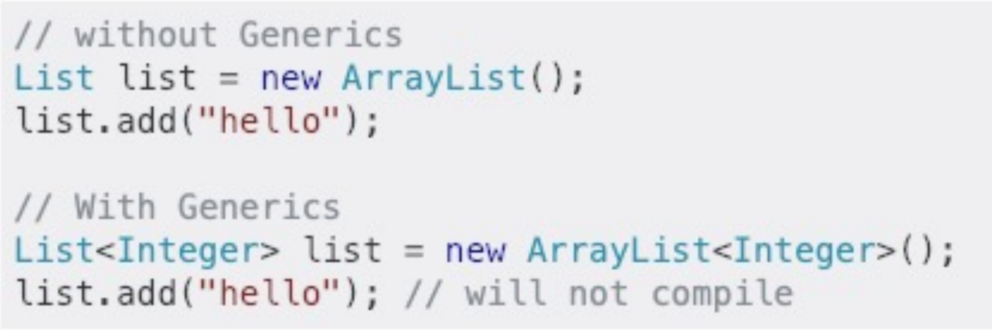
Java doesn’t complain that we do `.getY()` on the object coming out of the box, since we told it that the object was going to be a `Point`

In summary …

- **Generic class** is a type in Java that is written to accept another type as part of itself
    - Generic (or "parameterized") classes were added to Java (after version 5) to improve the type safety of Java's collections
    - A parameterized type has one or more other types' names written between < and >

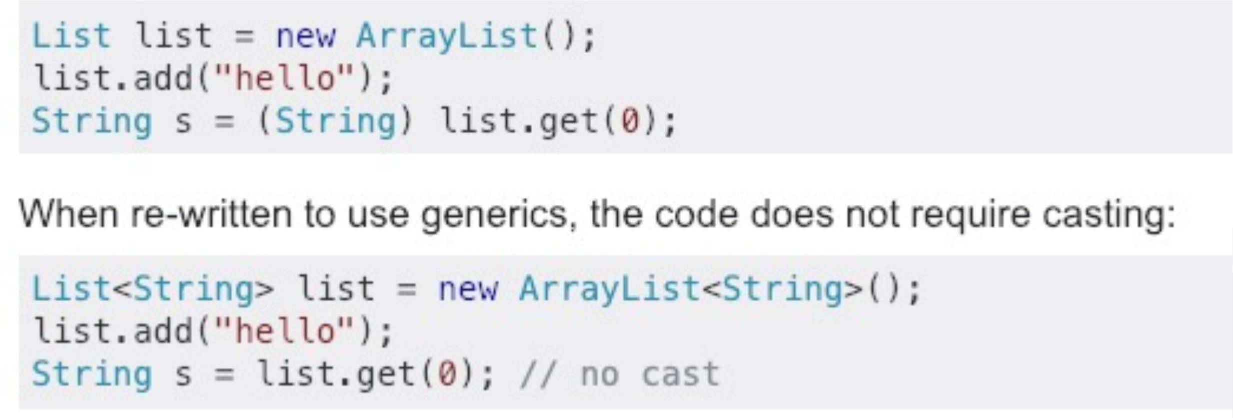
Why Use Generic Collections?

- Better type-checking: catch more errors, catch them earlier



```
// without Generics
List list = new ArrayList();
list.add("hello");

// With Generics
List<Integer> list = new ArrayList<Integer>();
list.add("hello"); // will not compile
```

- Documents intent
- Avoids the need to downcast from `Object`



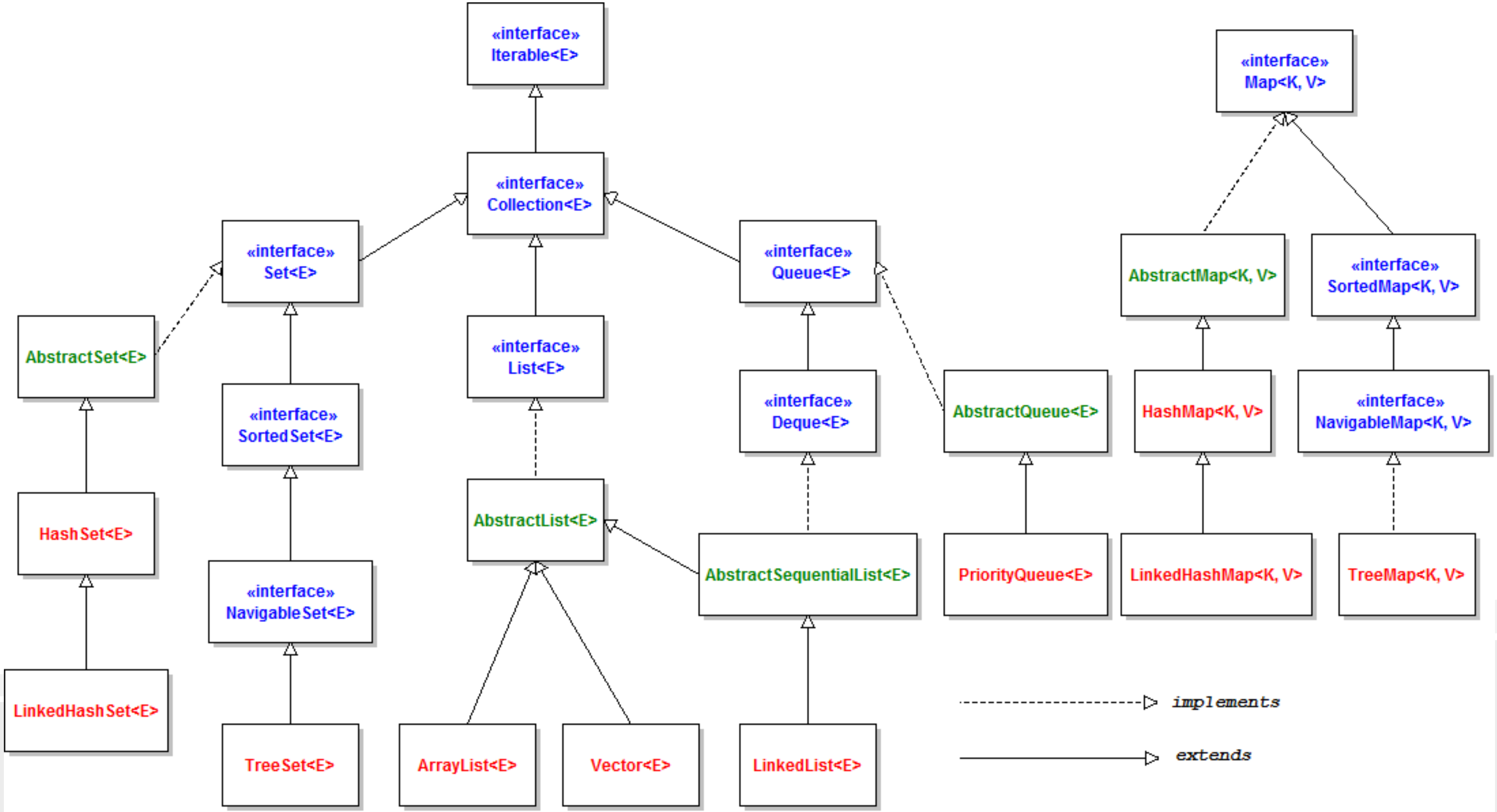
```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

When re-written to use generics, the code does not require casting:

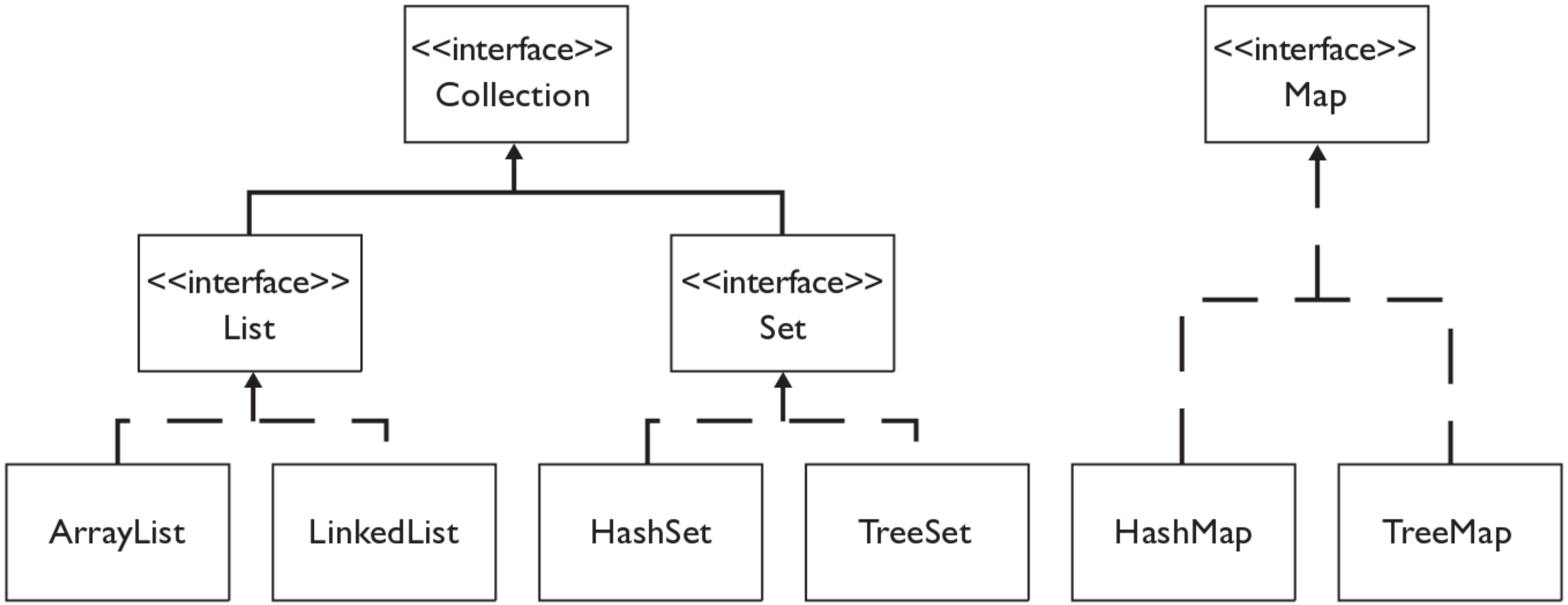
```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0); // no cast
```

Overview of Java Collections Framework (java.util.*)

```
public class ArrayList<E> extends AbstractList<E> implements List<E>, …
```



Java collections framework


<<interface>>
Collection
<<interface>>
List
<<interface>>
Set
<<interface>>
Map
ArrayList
LinkedList
HashSet
TreeSet
HashMap
TreeMap