Lecture 4

GENERICS. MOTIVATION

Old Way

- You have worked with *ArrayList* - a "collection"
  - `public void add (Object o);`
  - `public Object get (int index);`
- Every object in Java is of type `Object`; hence, these collections can store variables of any type.

```
ArrayList listOfStrings = new ArrayList();
listOfStrings.add("yo");

ArrayList listOfIntegers = new ArrayList();
listOfIntegers.add(new Integer(32));
```

Downside of downcasting

- Unfortunately, the fact that the List interface takes and returns *Objects* also means that we have to *downcast* the Object every time we call `get(index)`:

```
listOfStrings.add("hello");
String s = (String) listOfStrings.get(0);
```

- Having to downcast every time is both *tedious* and *distracting* because it litters the code with parentheses and class names
- Easy to introduce errors.

Downside of downcasting

```
1   ArrayList list1, list2, list3;
2   list1 = new ArrayList(); // for Strings
3   list2 = new ArrayList(); // for Integers
4   list3 = new ArrayList(); // for Students
5   list1.add("test");
6   list2.add(new Integer(17));
7   list2.add(new Integer(42));
8   ...
9   list3.add(new Student());
10  list1.add(new Student());
11  list2.add(new Integer(4));
12  list1.add("another string");
13
```

- A: Lines 6-7
- B: Line 9
- C: Line 10
- D: Line 11
- E: Line 12

Downside of downcasting

- If we later retrieve an `Object` from `list1` and assume (incorrectly) that it contains only Strings, our program will crash:

```
String s = (String) list1.get(1);
```
Given the code on previous slide, this will trigger a ClassCastException.

- It is still nice that the JVM catches our mistake at *run-time*, but it would be even *nicer* for the Java compiler to catch our mistake at *compile-time*.

Naïve fix

- How can we fix the problems of tedium, ugly code, and potential `ClassCastExceptions`?
- One naive strategy is to define a different `ArrayList` for every class we want to store in it, e.g.:

With specific types, we no longer have to downcast the result, and we're guaranteed that `get(index)` returns a `String`.

```
class ArrayListOfStrings {
  public void add (String s) { ... }
  public String get (int index) { ... }
}
class ArrayListOfIntegers {
  public void add (Integer i) { ... }
  public Integer get (int index) { ... }
}
class ArrayListOfShapes {
  public void add (Shape s) { ... }
  public Shape get (int index) { ... }
}
```

Better fix: "factor out" the type

```
class ArrayListOfStrings {
  public void add (String s) { ... }
  public String get (int index) { ... }
}
class ArrayListOfIntegers {
  public void add (Integer i) { ... }
  public Integer get (int index) { ... }
}
class ArrayListOfShapes {
  public void add (Shape s) { ... }
  public Shape get (int index) { ... }
}
```

- The *only* place these class definitions differ is in the *type* of the objects they hold.
- It seems like there should be a way to "factor out" the type...

JAVA GENERICS. PART 1.

Java Generics

- Since Java 1.5, Java has offered the ability to *parameterize* a class by a **type**.

- For example, when writing a "collection" class such as `ArrayList`, we can give it a type parameter `T`, or element `E`.

- Type parameters are typically given one-letter names:
  - K for "key", V for "value", E for "element, etc.

Generics for “ArrayListOfX”

```
class ArrayList<E> implements List<E> {
  //Interfaces too can be parameterized by a type
  E[ ] someStorage;
  int numElements = 0;

  void add (E element) {
    someStorage[numElements] = element;
    numElements++;
  }

  E get (int index) {
    return someStorage[index];
  }
}
```

Generics for “ListOfX”

- Similarly to classes, interfaces too can be parameterized by a type:

```
interface List<E> {
        void add (E element);
        E get (int index);
        void remove (int index);
}
```

Example

- For example, the following statement creates a list for strings:

```
ArrayList<String> list = new ArrayList<String>();
```

- You can now add *only strings* into the list. For instance,

```
list.add("Red");
```

- `list.add(new Integer(1)); //` this is NOT ok

Generic types

- Generic types must be *reference* types. You cannot replace a generic type with a primitive type such as **int, double,** or **char.**
- For example, the following statement is wrong:

```
ArrayList<int> intList = new ArrayList<int>();
```

To create an **ArrayList** object for **int** values, you have to use:

```
ArrayList<Integer> intList = new ArrayList<Integer>();
```

- You can add an **int** value to **intList.** For example,

```
intList.add(5);
```

Java automatically wraps **5** into **new Integer(5).** This is called ***autoboxing***.

```java
public class OldWay {

    private Object t;

    public Object get() {
        return t;
    }

    public void set(Object t) {
        this.t = t;
    }

        public static void main(String args[]){
        OldWay type = new OldWay();
        type.set("Marina");
        String str = (String) type.get(); //type casting,can cause ClassCastException
    }
}
```

```java
public class NewWay<T> {
    private T t;

    public T get(){
        return this.t;
    }

    public void set(T t1){
        this.t=t1;
    }

    public static void main(String args[]){
        NewWay<String> type = new NewWay<String>();
        type.set("Marina"); //valid

        NewWay type1 = new NewWay(); //raw type
        type1.set("Marina"); //valid
        type1.set(10); //valid and autoboxing support
    }
}
```

Raw type

- If we don't provide the type at the time of creation, compiler will produce a warning that

"NewWay is a raw type. References to generic type NewWay<T> should be parameterized".

- When we don't provide type, the type becomes `Object` and hence it's allowing both `String` and `Integer` objects.

- We should always try to avoid this because we will have to use type casting while working on raw type that can produce runtime errors.

- Done for a backward compatibility.

Diamond operator <>, idea

- Before JDK 7:
- Explicitly specifying generic class's instantiation parameter type.

```
ArrayList<String> list = new ArrayList<String>();
```

After JDK 7:

```
ArrayList<String> list = new ArrayList<>();
```

//interesting discussion on stackoverflow about it.

EXCEPTIONS

Handling bad input

```
public class Restrictions<T>{
  private T lastElement;
  private int numElements;

  public void add (T element)
  {
     if (element == null)
         // some code goes here
     lastElement = element;
     numElements++;
  }
}
```

- What code could I add if I want to prevent null elements from being added to this class?
- A: `throw new NullPointerException();`
- B: `return -1;`
- C: Either A or B
- D: None of the above

Exceptions

```
public class Restrictions<T>{
  private T lastElement;
  private int numElements;

  public void add (T element)
  {
     if (element == null)
        throw new NullPointerException(); //added line
     lastElement = element;
     numElements++;
  }
}
```

- Will this code **compile** even though it does not declare the *NullPointerException* being thrown?

- A: Yes (why?)
- B: No (why not?)

“Unchecked” exceptions

- There is a class of exceptions that I do not need to declare. We just can throw them
- We also do not need to handle them explicitly with catch/try block
- If you declare them, it is also OK.
- **Reason**: these exceptions are very common
- (show the example of the checked exception, eclipse)

JAVA COLLECTIONS

Collection

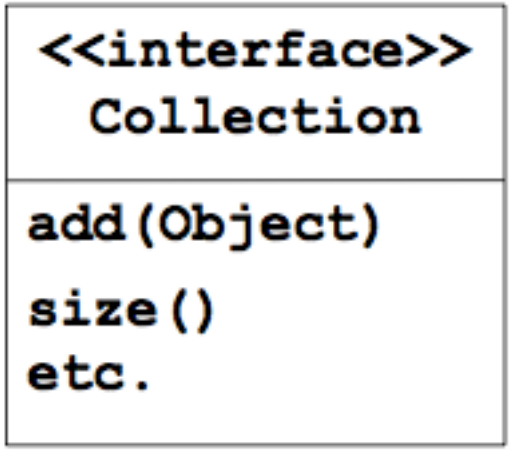
- What is a collection in real life?
  - Collection of similar objects

- A data structure (abuse of notation) is a collection of data organized in some fashion. The structure not only stores data but also supports operations for accessing and manipulating the data.

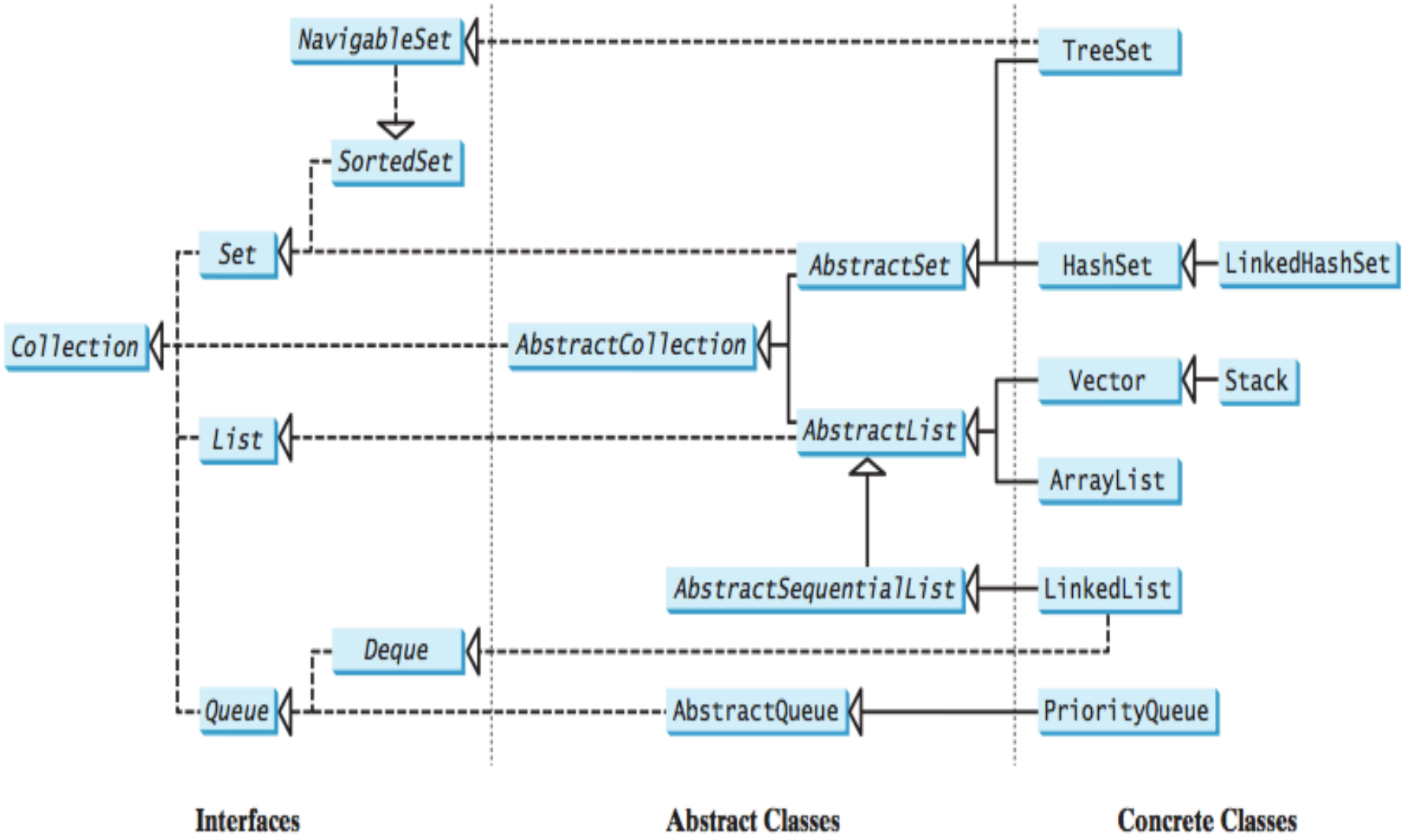
- *The* **Collection** *interface defines the common operations for lists, vectors, stacks, queues, priority queues, and sets.*

Collections

- Fundamentally, what we as programmers do with data is to store it and retrieve it and then operate on it.
- A **collection** is an ADT that contains data elements, and provides operations on them.
- There are different ways that elements can be collected:
  - Set, List, Sorted List…
- **All collections implement the interface Collection**



A collection is a container that stores objects.



public interface Collection<E> extends Iterable<E>

- What does the <E> mean in the above code?

- A: That this collection can only be used with objects of a built-in Java type called E
- B: That an object that implements collection can be instantiated to work with any object type
- C: That a single collection can hold objects of different types.