Introduction to Computer Science

Reichman University

Lecture 11-2

Generics

Lecture plan

➡ HashMap

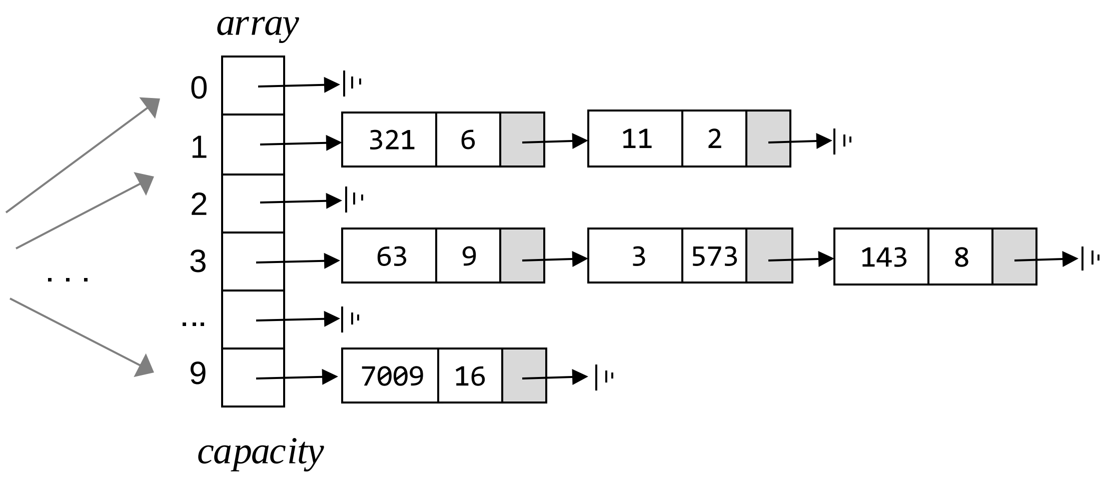
- Generic classes
- Wrapper classes
- Final example

HashMap

key	*value*
63	9
321	6
3	573
7009	16
143	8
11	2
...	

h(*key*) =

key % 10 =



put(*key* , *value*)

value = **get**(*key*)

HashMap

A commonly used Java class that implements the mapping algorithm.

HashMap: Example 1

```
/** Represents a  mapping of keys of type K on values of type V. */
public class HashMap<K,V> {

     /** Constructs an empty map with default capacity = 16. */
     public HashMap()

     /** Associates the given value with the given key. */
     public void put(K key, V value)

     /** Returns the value to which the given key is mapped. */
     public V get(K key)

     /** Returns true if the given key is contained in this map. */
     public boolean containsKey(K key)

     /** Returns a set of all the keys contained in this map. */
     public Set<K> keySet()

     /** Returns a set of all the values contained in this map. */
     public Set<V> values()
}
```

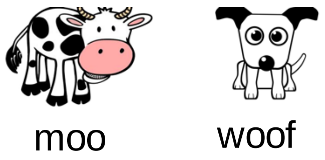
API

`HashMap<K,V>`

A *generic class*, designed for mapping objects *of any type* `K` on objects *of any type* `V`

Part of the *Java class library*;

Belongs to a package named `java.util`



We'll use a `HashMap` for mapping animals on the sounds that they make.

The sound can be a String, e.g. the name of a sound file.

HashMap: Example 1

```
/** Represents a  mapping of keys of type K on values of type V. */
public class HashMap<K,V> {

    /** Constructs an empty map with default capacity = 16. */
    public HashMap()

    /** Associates the given value with the given key. */
    public void put(K key, V value)

    /** Returns the value to which the given key is mapped. */
    public V get(K key)

    /** Returns true if the given key is contained in this map. */
    public boolean containsKey(K key)

    /** Returns a set of all the keys contained in this map. */
    public Set<K> keySet()

    /** Returns a set of all the values contained in this map. */
    public Set<V> values()
}
```

API

```
// Client code for managing <animal , sound> mappings
import java.util.HashMap;
class HashMapDemo1 {
   ...
```

<u>Technical note</u>

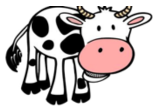
If a class wants to use a class that belongs to some other package, it must `import` that class;

<u>`HashMap<K,V>`</u>

A *generic class*, designed for mapping objects *of any type* `K` on objects *of any type* `V`

Part of the *Java class library*;

Belongs to a package named `java.util`


moo

woof

HashMap: Example 1

```
/** Represents a  mapping of keys of type K on values of type V. */
public class HashMap<K,V> {

    /** Constructs an empty map with default capacity = 16. */
    public HashMap()

    /** Associates the given value with the given key. */
    public void put(K key, V value)

    /** Returns the value to which the given key is mapped. */
    public V get(K key)

    /** Returns true if the given key is contained in this map. */
    public boolean containsKey(K key)

    /** Returns a set of all the keys contained in this map. */
    public Set<K> keySet()

    /** Returns a set of all the values contained in this map. */
    public Set<V> values()
}
```

API

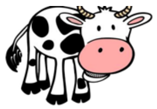
```
// Client code example
import java.util.HashMap;

class HashMapDemo1 {
    ...
    // Constructs a map of  <animal, sound>  pairs
    HashMap animalSound =
            new HashMap(<String,String>);
```

<u>`HashMap<K,V>`</u>

A *generic class*, designed for mapping objects *of any type* `K` on objects *of any type* `V`

Part of the *Java class library*;

Belongs to a package named `java.util`


moo

woof

<u>Note:</u>

The map is an instance (object) of `HashMap<K,V>`;

When we construct this object, we must specify `K` and `V`;

In this example, `K` and `V` are both `String` objects;

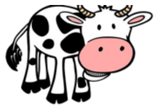
In general, they can be objects of *any type*.

HashMap: Example 1

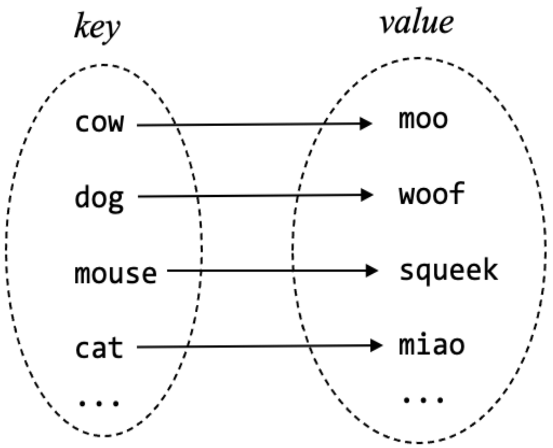
```
/** Represents a  mapping of keys of type K on values of type V. */
public class HashMap<K,V> {

    /** Constructs an empty map with default capacity = 16. */
    public HashMap()

    /** Associates the given value with the given key. */
    public void put(K key, V value)

    /** Returns the value to which the given key is mapped. */
    public V get(K key)

    /** Returns true if the given key is contained in this map. */
    public boolean containsKey(K key)

    /** Returns a set of all the keys contained in this map. */
    public Set<K> keySet()

    /** Returns a set of all the values contained in this map. */
    public Set<V> values()
}
```

API

```
// Client code example
import java.util.HashMap;

class HashMapDemo1 {
    ...
    // Constructs a map of  <animal, sound>  pairs
    HashMap animalSound =
            new HashMap(<String , String>);
    // Stores key-value entries
    animalSound.put("cow", "moo");
    animalSound.put("dog", "woof");
    animalSound.put("mouse", "squeek");
    animalSound.put("cat", "miao");
    ...
```


moo

woof



HashMap: Example 1

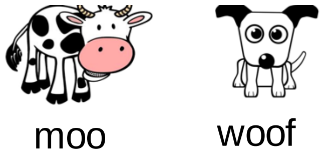
```
/** Represents a  mapping of keys of type K on values of type V. */
public class HashMap<K,V> {

    /** Constructs an empty map with default capacity = 16. */
    public HashMap()

    /** Associates the given value with the given key. */
    public void put(K key, V value)

    /** Returns the value to which the given key is mapped. */
    public V get(K key)

    /** Returns true if the given key is contained in this map. */
    public boolean containsKey(K key)

    /** Returns a set of all the keys contained in this map. */
    public Set<K> keySet()

    /** Returns a set of all the values contained in this map. */
    public Set<V> values()
}
```

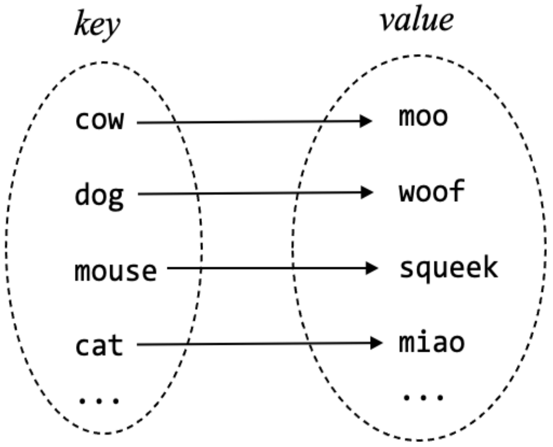
API

```
// Client code example
...
// Prints some animals, and their sounds.
System.out.println("Cow goes " +
                    animalSound.get("cow"));
System.out.println("Mouse goes " +
                    animalSound.get("mouse"));
...
```

output

```
Cow goes moo
Mouse goes squeak
```





HashMap: Example 1

```
/** Represents a  mapping of keys of type K on values of type V. */
public class HashMap<K,V> {

    /** Constructs an empty map with default capacity = 16. */
    public HashMap()

    /** Associates the given value with the given key. */
    public void put(K key, V value)

    /** Returns the value to which the given key is mapped. */
    public V get(K key)

    /** Returns true if the given key is contained in this map. */
    public boolean containsKey(K key)

    /** Returns a set of all the keys contained in this map. */
    public Set<K> keySet()

    /** Returns a set of all the values contained in this map. */
    public Set<V> values()
}
```

API

```
// Client code example
...

// Iterates over all the animals (keys) in the map;
// For each animal, prints the animal and its sound.
for (String animal : animalSound.keySet()) {
   String sound = animalSound.get(animal);
   System.out.println(animal + " goes " + sound);
}
```

output

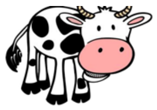
```
dog goes woof
cat goes miao
cow goes moo
mouse goes squeak
```

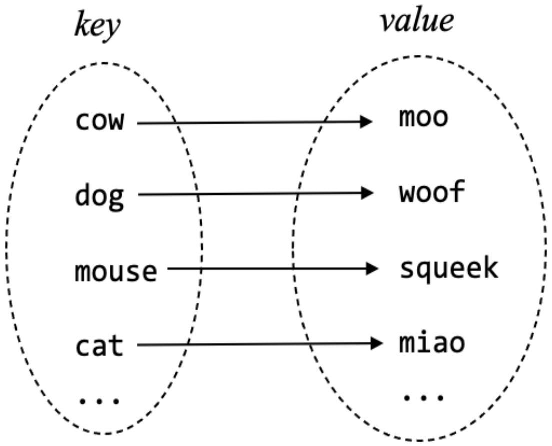
<u>`Set<V>`</u>

A generic, iterable, set of objects of any type `V`

also part of `java.util`;

Similar to the `Set` class that we wrote in lecture 9-2.





HashMap: Example 2

```
/** Represents a  mapping of keys of type K on values of type V. */
public class HashMap<K,V> {

    /** Constructs an empty map with default capacity = 16. */
    public HashMap()

    /** Associates the given value with the given key. */
    public void put(K key, V value)

    /** Returns the value to which the given key is mapped. */
    public V get(K key)

    /** Returns true if the given key is contained in this map. */
    public boolean containsKey(K key)

    /** Returns a set of all the keys contained in this map. */
    public Set<K> keySet()

    /** Returns a set of all the values contained in this map. */
    public Set<V> values()
}
```

```
import java.util.HashMap;

public class HashMapDemo2 {

    // A hashmap for memoizing results of the Fibonacci function
    static HashMap<Integer,Long> memoFib
                                         = new HashMap<>();

    // Computes the Fibonacci function
    public static long fib(int n) {
        if (n <= 1) return n;  // Base case

        // If already computed, returns the stored value
        if (memoFib.containsKey(n)) {
           return memoFib.get(n);
        }

        // Computes and stores the result
        long result = fib(n - 1) + fib(n - 2);
        memoFib.put(n, result);
        return result;
    }
}
```

Client code: Timing `fib(x)` function computations

```
Timer timer = new Timer();
System.out.println("Fibonacci(50) = " + fib(50) +
"; execution time: " + timer.elapsedTime() + " ms");

timer.reset();
System.out.println("Fibonacci(50) = " + fib(50) +
"; execution time: " + timer.elapsedTime() + " ms");
```

output

```
Fibonacci(50) = 12586269025; execution time: 0.255894 ms

Fibonacci(50) = 12586269025; execution time: 0.007202 ms
```

Lecture plan

- HashMap
- Generic classes
- Wrapper classes
- Final example

Designing a generic class

Regular class

```
/* A box that contains an int value. */
public class Box {

   private int x;   // contents of this box

   /* Puts x in this box */
   public void set(int x) {
      this.x = x;
   }

   /* Returns the contents of this box */
   public int get() {
      return x;
   }

   /* Textual representation of the contents of this box */
   public String toString() {
      return "[" + x + "]";
   }
}
```

the data of this class are values of type `int`

Designing a generic class

Generic class

```
/* A generic box that contains an object of type T. */
public class Box<T> {

    private T x;    // contents of this box

    /* Puts x in this box */
    public void set(T x) {
        this.x = x;
    }

    /* Returns the contents of this box */
    public T get() {
        return x;
    }

    /* Textual representation of the contents of this box */
    public String toString() {
        // whatever x is, have it describe itself
        return "[" + x.toString() + "]";
    }
}
```

the data of this class are objects of *any type* `T`

Generalizing to a generic class

Regular class

```
/* A box that contains an int value. */
public class Box {

    private int x;    // contents of this box

    /* Puts x in this box */
    public void set(int x) {
        this.x = x;
    }

    /* Returns the contents of this box */
    public int get() {
        return x;
    }

    /* Textual representation of the contents of this box */
    public String toString() {
        return "[" + x + "]";
    }
}
```

the data of this class are values of type `int`

- Generic classes are compiled like any other class;
- The symbol `T` is a *placeholder*;
- Commonly used placeholder symbols: `S`, `T`, `U`, `V`, `N`, `E`, ...

Using a generic class

Generic class

```
/* A generic box that contains an object of type T. */
public class Box<T> {

   private T x;   // contents of this box

   /* Puts x in this box */
   public void set(T x) {
      this.x = x;
   }

   /* Returns the contents of this box */
   public T get() {
      return x;
   }

   /* Textual representation of the contents of this box */
   public String toString() {
      // whatever x is, have it describe itself
      return "[" + x.toString() + "]";
   }
}
```

the data of this class are objects of *any type* T

Using a generic class

Generic class

```
/* A generic box that contains an object of type T. */
public class Box<T> {

   private T x;   // contents of this box

   /* Puts x in this box */
   public void set(T x) {
      this.x = x;
   }

   /* Returns the contents of this box */
   public T get() {
      return x;
   }

   /* Textual representation of the contents of this box */
   public String toString() {
      // whatever x is, have it describe itself
      return "[" + x.toString() + "]";
   }
}
```

the data of this class are objects of *any type* `T`

Client code demo (example 1)

```
// Creates some boxes, each containing a different thing
Box<String> b1 = new Box<String>();
b1.set("banana");

Box<Fraction> b2 = new Box<Fraction>();
b2.set(new Fraction(5,7));

Box<BankAccount> b3 = new Box<BankAccount>();
b3.set(new BankAccount("Maya"));
```

We created several `Box<T>` objects, each holding an object of a different data type.

Using a generic class

Generic class

```
/* A generic box that contains an object of type T. */
public class Box<T> {

   private T x;   // contents of this box

   /* Puts x in this box */
   public void set(T x) {
      this.x = x;
   }

   /* Returns the contents of this box */
   public T get() {
      return x;
   }

   /* Textual representation of the contents of this box */
   public String toString() {
      // whatever x is, have it describe itself
      return "[" + x.toString() + "]";
   }
}
```

the data of this class are objects of *any type* T

Client code demo (example 1)

```
// Creates some boxes, each containing a different thing
Box<String> b1 = new Box<String>();
b1.set("banana");

Box<Fraction> b2 = new Box<Fraction>();
b2.set(new Fraction(5,7));

Box<BankAccount> b3 = new Box<BankAccount>();
b3.set(new BankAccount("Maya"));

// Prints the contents of the boxes
System.out.println(b1);
System.out.println(b2);
System.out.println(b3);

// Declares another box and puts the first box in it
Box<Box<String>> boxInaBox =
                 new Box<Box<String>>();
boxInaBox.set(b1);
System.out.println(boxInaBox);
```

output

```
[banana]
[5/7]
[1 Maya 0.0]
[[banana]]
```

We created several `Box<T>` objects, each holding an object of a different data type.

Using a generic class

Generic class

```
/* A generic box that contains an object of type T. */
public class Box<T> {

   private T x;   // contents of this box

   /* Puts x in this box */
   public void set(T x) {
      this.x = x;
   }

   /* Returns the contents of this box */
   public T get() {
      return x;
   }

   /* Textual representation of the contents of this box */
   public String toString() {
      // whatever x is, have it describe itself
      return "[" + x.toString() + "]";
   }
}
```

The type parameters *must be objects*, not primitive types

Therefore: This code does not compile:

```
// Trying to create boxes that hold an int and a double values
Box<int> b1 = new Box<int>();
b1.set(17);

Box<double> b2 = new Box<double>();
b2.set(3.14159);
```

Workaround: Use *wrapper classes*:

```
// Creates boxes that hold an Integer  object and a Double object
Box<Integer> b1 = new Box<Integer>();
b1.set(Integer.valueOf(17));

Box<Double> b2 = new Box<Double>();
b1.set(Double.valueOf(3.14159));
```

```
// Simpler syntax convention (explained later)
Box<Integer> b1 = new Box<Integer>();
b1.set(17);

Box<Double> b2 = new Box<Double>();
b1.set(3.14159);
```

Wrapper classes

Java features eight *wrapper classes*:

Primitive types	Wrapper classes
`int`	`Integer`
`long`	`Long`
`byte`	`Byte`
`short`	`Short`
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`
`char`	`Character`

Wrapper classes ...

- Provide workarounds for having generic classes handle primitive values
- Provide some useful, type-specific, constants and services.

Wrapper classes: Working with primitive values

```
// Creating a “boxed” value:
Integer xObj = Integer.valueOf(19); // Or simply Integer xObj = 19; (see last example below)
System.out.println(xObj);           // Prints "19" (via the Integer class's toString method)

// Down-casting (retrieving the primitive value from the Integer object):
int x = xObj; // sometimes called “unboxing”

// Up-casting (assigning a primitive value to an Integer object):
xObj = 17; // sometimes called “autoboxing”; Shorthand of xObj = Integer.valueOf(17)
```

Explanation

In the Java class library, data structures (`Set`, `Stack`, `HashMap`, ...) are implemented as *generic classes*;

These classes are widely-used;

Therefore, there is an ongoing need to set `Integer` objects to `int` values, and vice versa (and the same is true for all wrapper classes and primitive data types);

Therefore, Java makes the *boxing* / *unboxing* casting operations easier for the programmer.

Wrapper classes: Useful constants and services

Java’s wrapper classes also provide useful type-specific services and constants.

Client code example (using some static services of the `Integer` class):

```
// Parsing strings to integers:
int x = Integer.parseInt("43"); // Converts the string "43"  to the int 43
System.out.println(x + 2);      // Prints 45

// Converting integer to binary:
String xBin = Integer.toBinaryString(x);  // Computes the binary representation of  x
System.out.println(xb);                   // Prints  101011

// Supplying the Min and Max values of int:
System.out.println(Integer.MIN_VALUE);    // Prints -2147483648
System.out.println(Integer.MAX_VALUE);    // Prints 2147483647
```

Endnotes / Side comments

Q: How does the compiler know that a class is designed to represent objects?

A: If the class features at least one method that is not `static`, it is designed for creating and managing objects;

Another way to tell:

The class declares private variables (fields).

```
/* A box that contains an int value. */
public class Box {

    private int x;   // field declaration

    /* Puts x in this box */
    public void set(int x) {
        this.x = x;
    }

    /* Returns the contents of this box */
    public int get() {
        return x;
    }

    ...
}
```

Endnotes / Side comments

```
/* A box that contains an int value. */
public class Box {
   private int x;   // field declaration
   /* Puts x in this box */
   public void set(int x) {
      this.x = x;
   }
   /* Returns the contents of this box */
   public int get() {
      return x;
   }
   ...
}
```

This class is designed to represent objects;

But... it has no *constructors*!

How can the class create objects?

When you don’t write a constructor,
the compiler adds a default constructor.

Example:

```
public class AnyClass {
   private int x;
   ...
}
```

As if you wrote:

```
public class AnyClass {
   private int x;
   // default constructor
   public AnyClass() {
   }
   ...
}
```

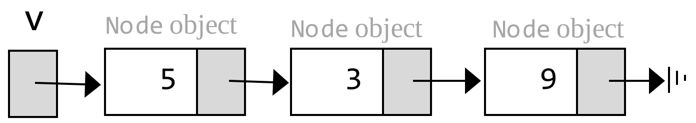
The default constructor is designed to do one thing only: Create an object of the class type.

Lecture plan

- HashMap
- Generic classes
- Wrapper classes
- Final example

Generic List

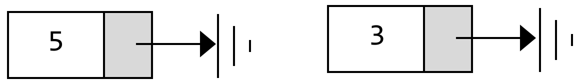
In lecture 10-1 we built `Node` and `List` classes for representing lists of *integers*:



Can we generalize these classes to represent lists of *any data type*?

```
/** Represents a node containing an int value */
public class Node {

   int data;
   Node next;

   /** Constructs a node with the given data.
   *   The new node will point to the next node. */
   public Node(int data, Node next) {
      this.data = data;
      this.next = next;
   }

   /** Constructs a node with the given data.
   *   The new node will point to null. */
       public Node(int data) {
       this(data, null);
   }

   /** Textual representation of this node. */
   public String toString() {
       return "" + data;
   }
}
```

Lecture 10-2



Generic List

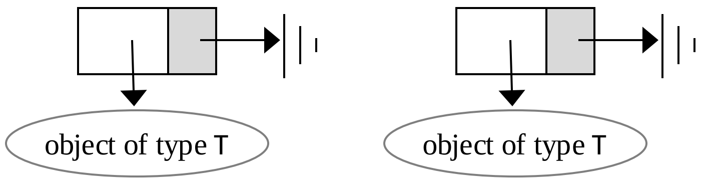
Generic node

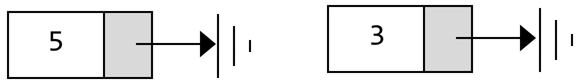
```
/** Represents a generic node in a generic linked list. */
public class Node<T> {

   T data;          // an object of type T
   Node<T> next;

   /** Constructs a node with the given data.
   *   The new node will point to the next node. */
   public Node(T data, Node<T> next) {
      this.data = data;
      this.next = next;
   }

   /** Constructs a node with the given data.
   *   The new node will point to null. */
       public Node(T data) {
       this(data, null);
   }

   /** Textual representation of this node. */
   public String toString() {
       return "" + data;
   }
}
```

Generalizing to a generic class

```
/** Represents a node containing an int value */
public class Node {

   int data;
   Node next;

   /** Constructs a node with the given data.
   *   The new node will point to the next node. */
   public Node(int data, Node next) {
      this.data = data;
      this.next = next;
   }

   /** Constructs a node with the given data.
   *   The new node will point to null. */
       public Node(int data) {
       this(data, null);
   }

   /** Textual representation of this node. */
   public String toString() {
       return "" + data;
   }
}
```

Lecture 10-2





Generic List

```
/** A list of integers. */
public class List {

   private Node first; // Pointer to the first list element
   private int size;   // Number of elements

   /** Constructs an empty list. */
   public List() {
      first = null;
      size = 0;
   }

   ...

   /** Adds the given int value to the end of this list. */
   public void add(int data) { // code omitted }

   /** Returns a string representation of this list,
    * in the form of (element element , ..., element) */
   public String toString() {
      String str = "(";
      Node current = first;
      while (current != null) {
         str += current.data + " ";
         current = current.next;
      }
      return str.substring(0, str.length()-1) + ")";
   }
}
```

Lecture 10-2

Generic List

Generic list

```
/** A generic list of objects of any type. */
public class List<T> {

   private Node<T> first;  // Pointer to the first list element
   private int size;       // Number of elements

   /** Constructs an empty list. */
   public List() {
      first = null;
      size = 0;
   }

   ...

   /** Adds the given data (an object) to the end of this list. */
   public void add(T data) { // code omitted }

   /** Returns a string representation of this list,
     * in the form of (element element , ..., element) */
   public String toString() {
      String str = "(";
      Node<T> current = first;
      while (current != null) {
         str += current.data + " ";
         current = current.next;
      }
      return str.substring(0, str.length()-1) + ")";
   }
}
```

Generalizing to a generic class

```
/** A list of integers. */
public class List {

   private Node first;  // Pointer to the first list element
   private int size;    // Number of elements

   /** Constructs an empty list. */
   public List() {
      first = null;
      size = 0;
   }

   ...

   /** Adds the given int value to the end of this list. */
   public void add(int data) { // code omitted }

   /** Returns a string representation of this list,
     * in the form of (element element , ..., element) */
   public String toString() {
      String str = "(";
      Node current = first;
      while (current != null) {
         str += current.data + " ";
         current = current.next;
      }
      return str.substring(0, str.length()-1) + ")";
   }
}
```

Lecture 10-2

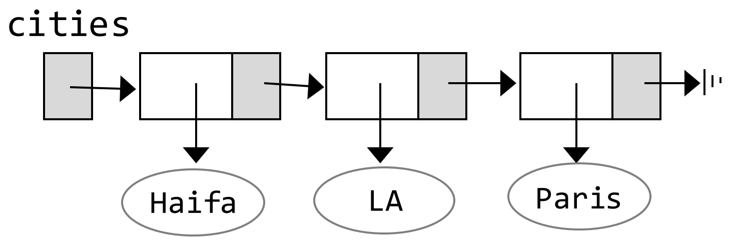
Generic List

Generic list

```
/** A generic list of objects of any type. */
public class List<T> {

   private Node<T> first;  // Pointer to the first list element
   private int size;       // Number of elements

   /** Constructs an empty list. */
   public List() {
      first = null;
      size = 0;
   }

   ...

   /** Adds the given data (an object) to the end of this list. */
   public void add(T data) {  // code omitted }

   /** Returns a string representation of this list,
    * in the form of (element element , ..., element) */
   public String toString() {
      String str = "(";
      Node<T> current = first;
      while (current != null) {
         str += current.data + " ";
         current = current.next;
      }
      return str.substring(0, str.length()-1) + ")";
   }
}
```

Client code (example)

```
...
List<String> cities = new List<String>();
cities.add("Haifa");
cities.add("LA");
cities.add("Paris");
System.out.println(cities)   // [Haifa, LA, Paris]
```



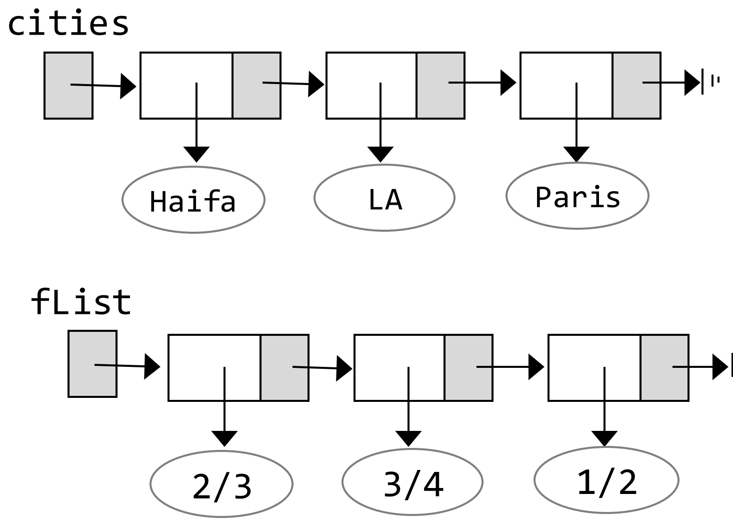
Generic List

Generic list

```
/** A generic list of objects of any type. */
public class List<T> {

   private Node<T> first;  // Pointer to the first list element
   private int size;       // Number of elements

   /** Constructs an empty list. */
   public List() {
      first = null;
      size = 0;
   }

   ...

   /** Adds the given data (an object) to the end of this list. */
   public void add(T data) {  // code omitted }

   /** Returns a string representation of this list,
     * in the form of (element element , ..., element) */
   public String toString() {
      String str = "(";
      Node<T> current = first;
      while (current != null) {
         str += current.data + " ";
         current = current.next;
      }
      return str.substring(0, str.length()-1) + ")";
   }
}
```

Client code (example)

```
...
List<String> cities = new List<String>();
cities.add("Haifa");
cities.add("LA");
cities.add("Paris");
System.out.println(cities)    // (Haifa, LA, Paris)
...
List<Fraction> fList = new List<Fraction>();
fList.add(new Fraction(7,8));
fList.add(new Fraction(1,2));
fList.add(new Fraction(5,9));
System.out.println(fList);    // (7/8, 1/2, 5/9)
```



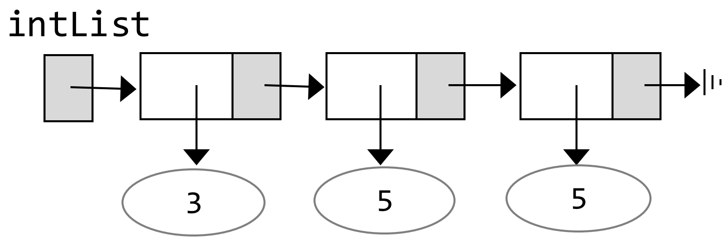
Generic List

Generic list

```
/** A generic list of objects of any type. */
public class List<T> {

   private Node<T> first; // Pointer to the first list element
   private int size;      // Number of elements

   /** Constructs an empty list. */
   public List() {
      first = null;
      size = 0;
   }

   ...

   /** Adds the given data (an object) to the end of this list. */
   public void add(T data) { // code omitted }

   /** Returns a string representation of this list,
    * in the form of (element element , ..., element) */
   public String toString() {
      String str = "(";
      Node<T> current = first;
      while (current != null) {
         str += current.data + " ";
         current = current.next;
      }
      return str.substring(0, str.length()-1) + ")";
   }
}
```

Client code (example)

```
...
// a list of int values
List<Integer> intList = new List<Integer>();
intList.add(Integer.valueOf(5));
intList.add(3); // (uses autoboxing)
intList.add(9);
System.out.println(intList); // (5, 3, 9)
```


intList
3
5
5