06 Generic Classes

Instructor: Ke Wei（柯韋）

A319 Ext. 6452 wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP212/19/

Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

October 3, 2019

Outline

1. Element Types
2. Parametric Types
3. Type Constraints
4. Example: An Ordered List
5. Practice: Implementing a Set Based on an Ordered List

Collections and Element Types

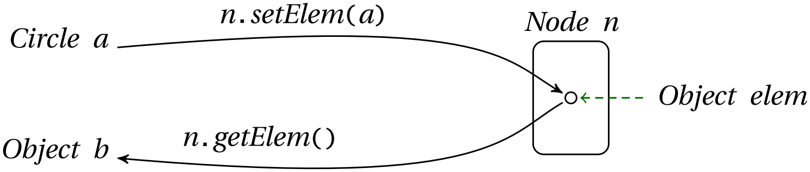
- A collection groups of a number of elements, usually having the same type.
- In most cases, the operations of a collection is independent to the element type, for example, to *put* an element into the collection.
- Sometimes, a collection can only store elements that can be ordered. Other properties of the elements are irrelevant.
- The require properties of the elements can be specified in a superclass or some interfaces.

```
class Node {
    private Object elem;
    public Object getElem()
        { return elem; }
    public void setElem(Object o)
        { elem = o; }
}
```

```
class OrdNode {
    private Comparable elem;
    public Comparable getElem()
        { return elem; }
    public void setElem(Comparable o)
        { elem = o; }
}
```

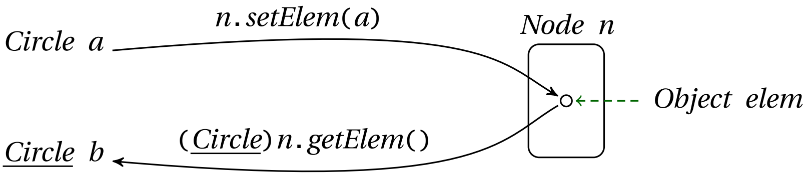
Down-casting Everywhere

- When we have a particular element to put into a collection, we know what exactly it is, for example, a *Circle*.
- However, when we fetch the element out of the collection, its type has been widened to the superclass *Object* or some less specific `interface` such as *Comparable*.
- Some type information has been lost at compile time, and we must *downcast* the less specific type to the type we know (assume), and the compiler has no mean to check this. The type checking is delayed to runtime.



Down-casting Everywhere

- When we have a particular element to put into a collection, we know what exactly it is, for example, a *Circle*.
- However, when we fetch the element out of the collection, its type has been widened to the superclass *Object* or some less specific `interface` such as *Comparable*.
- Some type information has been lost at compile time, and we must *downcast* the less specific type to the type we know (assume), and the compiler has no mean to check this. The type checking is delayed to runtime.



Parametric Classes and Interfaces

- Java allows *parametric* types, called *generics*, which enable us to specify the element type as a *type parameter* (*type variable*) of a collection.
- At the time we use the generic type, we specify the actual element type to *instantiate* a new type. This is called the *instantiation* of a generic type.
- We can now tell the compiler the element type that was previously only in our minds, while still having collections of various element types in one definition.
- This kind of polymorphism — one definition for many element types — is called *parametric polymorphism*.

```
class Node<T> {
    private T elem; ...
}
```

```
class OrdNode<T extends Comparable<T>> {
    private T elem; ...
}
```

Instantiations

Generic Class: $Lst\langle T\rangle$

```
class Lst⟨T⟩
    T head;
    Lst⟨T⟩ tail;

    T getHead() ...
```

$T \mapsto \underline{String}$

Instantiation: $Lst\langle\underline{String}\rangle$

```
class Lst⟨String⟩
    String head;
    Lst⟨String⟩ tail;

    String getHead() ...
```

$T \mapsto \underline{Circle}$

Instantiation: $Lst\langle\underline{Circle}\rangle$

```
class Lst⟨Circle⟩
    Circle head;
    Lst⟨Circle⟩ tail;

    Circle getHead() ...
```

Declaring Generic Classes and Interfaces

- A generic class or interface declaration is like a normal class or interface declaration, except that we put a type parameter list enclosed by a pair of angle brackets following the class or interface name.

```
public class Lst<T> ...
```

- Within the generic declaration, the type parameters can be used as normal types, for example, to declare variables and parameters.

```
public interface UnaryOp<A,B> {
    B op(A x);
}
```

```
public interface BinaryOp<A,B,C> {
    C op(A x, B y);
}
```

- Because all classes can be used to substitute for a type parameter, we can do little with a variable declared by a type parameter, except to invoke the methods available in the *Object* class.

Constraints on Type Parameters

- The methods in *Object* can be invoked because every class is a subclass of *Object* and must have those methods.
- If the class to substitute for a type parameter T is sure to be a subclass of some superclass, say *Shape*, we can invoke a method of the superclass on a variable x of T, say $x.getArea()$ without causing an error in the instantiations.
- The requirement that "the class to substitute for the type parameter T must be a subclass of *Shape*" is specified by a *constraint*.

  ```
  class ShapeList<T extends Shape> ...
  ```

- A constraint can also be a list of one or more interfaces, or, a superclass followed by a list of interfaces.

  ```
  class OutlinedShapeSet<T extends Shape & Stroke> ...
  ```

- We can have *OutlinedShapeSet⟨OutlinedCircle⟩* because *OutlinedCircle* is a subclass of *Shape* and implements *Stroke*.

Example: An Ordered List — Insertion

An ordered list *OrdLst* stores elements increasingly in a *circular* singly linked list with a dummy node.

```
public class OrdLst<T extends Comparable<T>> implements Iterable<T> {
    private OrdNode<T> dummy;
    private int len;
    public OrdLst() { dummy = new OrdNode<>(); len = 0; }
    public void ins(T x) {
        OrdNode<T> q = dummy, p = q.getNext();
        while ( p != dummy && p.getElem().compareTo(x) <= 0 ) {
            q = p;
            p = q.getNext();
        }
        q.setNext(new OrdNode<>(x, p));
        ++len;
    }
```

Example: An Ordered List — Finding

The *indexOf*(*o*) method returns the index of the first *o* in the list.

```
public int indexOf(T o) {
    OrdNode<T> p = dummy.getNext();
    int i = 0;
    while ( p != dummy ) {
        int c = p.getElem().compareTo(o);
        if ( c == 0 )
            return i;
        if ( c > 0 )
            return -1;
        ++i;
        p = p.getNext();
    }
    return -1;
}
```

Example: An Ordered List — Getter

The *get*(*i*) and *del*(*i*) methods operate on the index of an element.

```
public T get(int i) {
    checkIndex(i);
    OrdNode<T> p = dummy.getNext();
    while ( i > 0 ) {
        --i;
        p = p.getNext();
    }
    return p.getElem();
}
```

Example: An Ordered List — Deletion

```
public T del(int i) {
    checkIndex(i);
    OrdNode<T> q = dummy, p = q.getNext();
    while ( i > 0 ) {
        --i;
        q = p;
        p = q.getNext();
    }
    T x = p.getElem();
    q.setNext(p.getNext());
    --len;
    return x;
}
```

Example: An Ordered List — Iterator

The anonymous class captures the enclosing context, say, *dummy*.

```
    @Override public Iterator<T> iterator() {
        return new Iterator<T>() {
            OrdNode<T> curr = dummy.getNext();
            @Override public boolean hasNext() {
                return curr != dummy;
            }
            @Override public T next() {
                T x = curr.getElem();
                curr = curr.getNext();
                return x;
            }
        };
    }
}
```

Example: An Ordered List — Testing

```
OrdLst<String> ls = new OrdLst<>();
ls.ins("banana");
ls.ins("apple");
ls.ins("watermelon");
ls.ins("orange");
ls.ins("grape");
for ( int i = 0; i < ls.size(); ++i )
    System.out.println(ls.get(i));
System.out.println("DEL: "+ls.del(ls.indexOf("grape")));
for ( String s : ls )
    System.out.println(s);
while ( ls.size() > 0 )
    ls.del(0);
for ( String s : ls )
    System.out.println(s);
```

Example: An Ordered List — Testing

```
OrdLst<String> ls = new OrdLst<>();
ls.ins("banana");
ls.ins("apple");
ls.ins("watermelon");
ls.ins("orange");
ls.ins("grape");
for ( int i = 0; i < ls.size(); ++i )
    System.out.println(ls.get(i));
System.out.println("DEL: "+ls.del(ls.indexOf("grape")));
for ( String s : ls )
    System.out.println(s);
while ( ls.size() > 0 )
    ls.del(0);
for ( String s : ls )
    System.out.println(s);
```

```
apple
banana
grape
orange
watermelon
DEL: grape
apple
banana
orange
watermelon
```

Practice: Implementing a Set Based on an Ordered List

- An ordered list is suitable for set operations, because it is easy to remove duplicates.
- Write a method *uniq* for the *OrdLst* class to remove all duplicated elements.
- Define a generic class *OrdSet* of element type T, with a field of class $OrdLst\langle T\rangle$ as its underlying store. Note that the class is mutable.
- Delegate the $size()$ and $iterator()$ methods to the underlying *OrdLst*.
- Define three methods for *OrdSet* each operating on one element — $contains(x)$, $add(x)$ and $remove(x)$.
- Define three methods for *OrdSet* each operating on an iterable of elements — $addAll(xs)$, $removeAll(xs)$ and $keepAll(xs)$. Note that $keepAll(xs)$ keeps only those elements listed in xs and removes all other elements.
- Write some code to test the functionality of class *OrdSet*.
- Zip your source files, including `OrdNode.java`, `OrdLst.java`, `OrdSet.java` and `Test.java` into `OrdSet.zip`. Upload the `OrdSet.zip`.