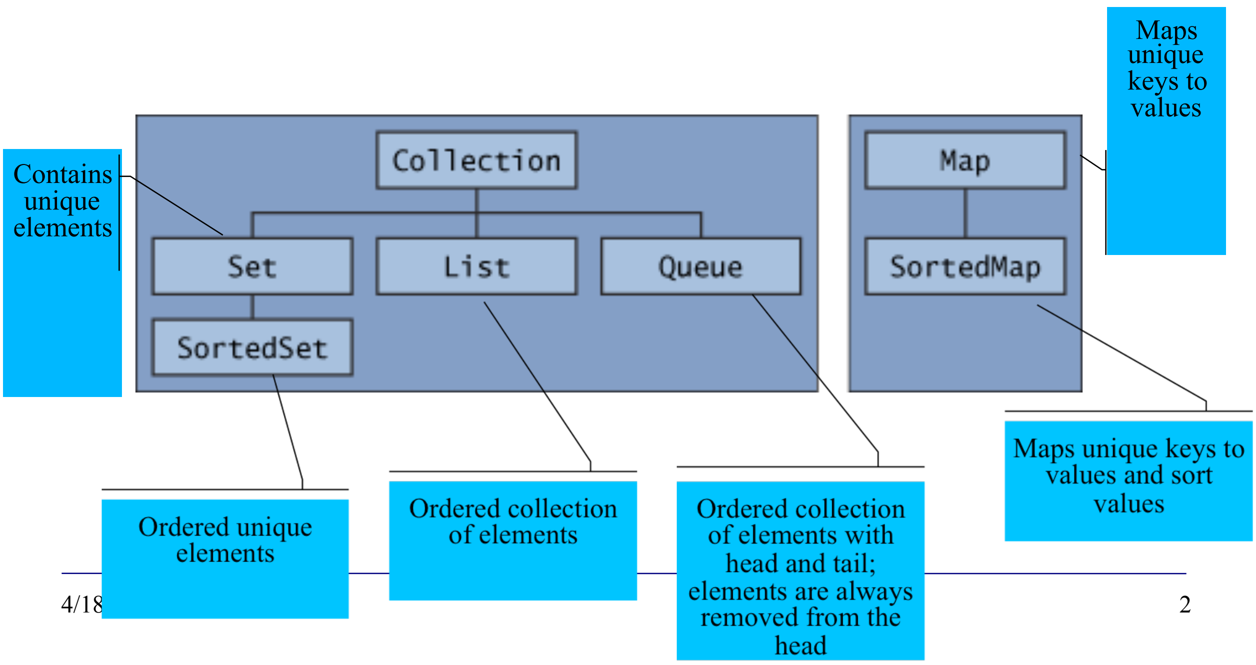
Generics in Java

Advanced Programming

The collections' interfaces



Collections in Java

- **Array**
  - **has a special language support**
- **Iterators**
  - **Iterator(I)**
- **Collections also called containers**
  - **Collection(I)**
  - **Set(I)**
    - **HashSet(c), TreeSet(c)**
  - **List(I)**
    - **ArrayList(c), LinkedList(c)**
  - **Map(I)**
    - **HashMap(c), TreeMap(c)**

Getting from a collection

- Let us consider this example:

```
List myIntegerList = new LinkedList();
myIntegerList.add(new Integer(0));
Integer x = (Integer) myIntegerList.iterator().next();
```

- The cast on line 3 is slightly annoying
  - The compiler can only guarantee that iterator returns an object of type Object

Getting from a collection

- The casting introduces a **run time error**, since the programmer might be mistaken
- What if programmers could mark a list as being of a particular data type?
- This is the idea behind **generics**

Generics

- Generics allow you to abstract over types
- The most common examples are container (e.g., arrays and lists) types, such as those in the Collection hierarchy
- List<Integer> is a generic type that says that the list is of integers.

```
List<Integer> aList = new List<Integer>();
```

Example

- casting

run time

```
List myIntegerList = new LinkedList();
myIntegerList.add(new Integer(0));
Integer x = (Integer)
    myIntegerList.iterator().next();
```

- with generics

compile time

```
List<Integer> myIntegerList = new
    LinkedList<Integer>();
myIntegerList.add(new Integer(0));
Integer x = myIntegerList.iterator().next();
```

No casting! we get an Integer object

Increasing robustness

- With generics, the compiler can check the type
- In contrast, the cast tells us something the programmer thinks is true at a single point in the code and it will be checked at run time

Generics and derivation

```
List<String> ls = new ArrayList<String>(); //Ok
List<Object> lo = ls;  // Compiler error!!!! Why?
```

- Observe line 2: is a List of String a List of Object?
- If yes, we could do the following:

```
lo.add(new Object()); // We can add an object of type Object
String s = ls.get(0); // attempts to assign an Object to a String!NO!
```

- The object referenced by ls does not hold only strings anymore! We need to have another instrument more flexible, the Wildcards

Wildcards

- As List<Integer> is not a subtype of List<Object> we cannot use some useful practices of the old good collections anymore
- For example, List can have any type of members whereas **List<Integer> can only have Integer** members
- Wildcards are used to get back classic behaviours for subtyping

Example: Collection of unknown

- Collection<?> …The type of collection is unknown

```
Collection<?> aCollection = new ArrayList<String>();
```

- aCollection is a reference of a Collection of unknown type and points to an object of type ArrayList of String
  - Note that Collection is an interface and ArrayList Implements List which extends Collection

Limitation – adding

- With the collection of unknown, **we cannot directly add** to Collection a specific **object**:

```
aCollection.add(0,new Object()); // compiler error!
```

- As we do not know of what type is the collection (it is unknown to the compiler!) and we can only pass elements that are **subtypes of the unknown**,
  - **since we do not know the unknown type -> we can only pass "null", which is subtype of any type**

Gaining - getting

- There is no compile time error to use get() and make use of the result, instead
- We **get back an unknown type**, but we always know that it **will be a subtype of Object**
- Thus we can assign the result of get() to a **variable of type Object (covariant property the return type: a return type can be a subtype of the return type: it** can be a subclass of Object)

With collection of unknown…

“Populating a list is uncertain getting from a list is certain”

Bounded Wildcards

```
List<? extends Shape>
```

- It is a **wildcard bounded by Shape**
  - This allows to use the Wildcards with all the subtypes of Shape
- As direct subtyping for generics is not allowed, bounded Wildcards allow to **extend behaviours to children**

Example

```
public abstract class Shape{
   public abstract void draw(Canvas c);
}
public class Circle extends Shape{
   public void draw(Canvas c){…};
}
public class Rectangle extends Shape{
   public void draw(Canvas c){…};
}

public class Canvas{
   public void draw(Shape s){
        s.draw(this);
   }
   public void drawAll(List<Shape> aList){
        for(Shape s : aList){
        s.draw(this);
    }
}
```

- drawAll() can be only used with Shape and it **cannot be used with any derived class**!
- Then we define

```
public void drawAllReally(List<? extends Shape>
    aList){
     for(Shape s : aList){
         s.draw(this);
     }
}
```

- Now we can use lists of any derived type of Shape

```
List<Circle> aListCircle= new List<Circle>();
myCanvas.drawAllReally(aListCircle);
```

see code LECT10

Careful!

- Again, it is illegal to write **directly** into a list through the body of a method

```
public void addRectangle(List<? extends Shape> aList){
    aList.add(0,new Rectangle()); //compile time error
}
```

- As we **do not know the subtypes of Shape** and whether the subtype of Shape is a Rectangle (or a parent class of Rectangle) i.e.:
  - Rectangle extends Base and Base extends Shape
- We need a new instrument: parametrised types and methods…

Parameterised type

- A parameterised type is a class

```
public class Map<E> {…} ;
```

  - Where E is a parameter (**it is known but not defined**)
- In the use of the class, all occurrences of the **formal type parameter (E) are replaced** by the actual type argument (e.g., Integer).

```
Map<Integer> aMap = new aMap<Integer>();
```

- **Map<Integer>** stands for a version of Map where E has been uniformly replaced by Integer

Note: Pseudo polymorphism with Marker Interfaces

- The parametrisation of a class can be done in another way: through the use of empty **interfaces** called **Marker**
- Makers allow to group classes that want to have the same services. **They are empty**
- Ex: all the classes that implement Cloneable (I) can use (and must override) the clone() method of Object
- Maker interfaces are not really a parameter like the <E>

Parametrised types …

```
public interface Map<K, V> {
    public void put(K key, V value);
    public V get(K key);
}
...
Map<String, String> m = new HashMap<String, String>();
```

a parameterised type can have more than one parameter

- where `HashMap<String,String>` defines an implementation of `Map<String,String>`

…and methods

- one or more parameters are inserted after the modifier parameters in method declaration

```
public <T> void add(T t, List<T> aList){
    aList.add(t); //correct as T is known now!
}
```


…and methods

```
public <T> void add(T t, List<T> aList){aList.add(t); //finally we can fill a list}
```

- or

```
public <T> void add(List<T> aList, List<? Extends T> aChildList){…};
```

- or

```
public <T,S extends T> void add(List<T> aList, List<S> aSmallList){…};
```

// equivalent to the one above if S extends T

- or

```
public <T> void add(List<T> aList, List<S extends T> aSmallLsit){…};
```

// equivalent to the one above

- or

```
public <T> List<T> returnNewList(List<T> aList){…};
```

Parameterising

- **With pseudo polymorphism;**
- java.lang.Comparable is an interface

```
class MySortedList{
    private Comparable [] elements;
    …
    public MySortedList (){
        elements = new Comparable[size];
    }
    public int add(Comparable t);
    public Comparable remove(int index);
    public int size();
}
```

- **With generics**

```
class MySortedList<T implements Comparable>{
    private T [] elements;
    …
    public MySortedList (){
        elements = new T[size];
    }
    public int add(T t);
    public T remove(int index);
    public int size();
}
```


Parameterising

```
public static void main(String [] args){
    MySortedList list =
    new MySortedList();
    // adding Integers
…
    Integer i = (Integer)list.remove(0);
}
```

As I do not know what will be the implementation type of the object at 0, **I have to cast** in any case

```
public static void main(String [] args){
    MySortedList<Integer> list =
    new MySortedList<Integer>();
    // adding Integers
…
    Integer i = list.remove(0);
}
```

Here I only know that **T implements Comparable**.

http://docs.oracle.com/javase/1.3/docs/api/java/lang/Comparable.html

Inference of types

- What does it happen when types in parametrised methods are different?
- The compiler infers types
  - It always infer the most generic

Compiler's inference - Example

```
Static <T> fromArrayToCollection(T[] a, Collection<T> c){
    for(T o : a){
        c.add(o);
    }
}

Object[] aCO = new Object[100];
Collection<Object> aCollectionObject = new ArrayList<Object>();

String[] aCS = new String[100];
Collection<String> aCollectionString = new ArrayList<String>();

Integer[] aCI = new Integer[100];
Float[] aCF = new Float[100];
Number[] aCN = new Number[100];

Collection<Number> aCollectionNumber = new ArrayList<Number>();
```

```
fromArrayToCollection(aCO,aCollectionObject);
//T is inferred to be Object
fromArrayToCollection(aCS,aCollectionString);
// T is inferred to be String
fromArrayToCollection(aCS,aCollectionObject);
// T is inferred to be Object
fromArrayToCollection(aCI,aCollectionNumber);
// T is inferred to be Number
fromArrayToCollection(aCF,aCollectionNumber);
// T is inferred to be Number
fromArrayToCollection(aCN,aCollectionNumber);
// T is inferred to be Number
fromArrayToCollection(aCN,aCollectionString);
// T compile time error
```

The compiler infers from the less specialised type

from: http://download.oracle.com/javase/tutorial/extra/generics/methods.html

Raw type

- A **raw type** is the classic type
- For example
  - Collection is a classic type
  - Collection<V> is the corresponding generic with type V. The raw type of Collection<V> is Collection

Type erasure

- Type Erasure is the phase after Inference of types in which the compiler translates the source into bytecode.
- Type erasure exists to have compliance with non generics code (legacy code)

Type erasure

- At erasure the **generic type are removed**
  - List<Number> becomes List which can contain any type of object
- The compiler just check the correctness of the types and then save byte code as in traditional Java compiled code
- At run time it is impossible to deduce the original type

Original Code

```
class Pair<elem> {
    elem x;  elem y;
    Pair (elem x, elem y) {this.x = x; this.y = y;}
    void swap () {elem t = x; x = y; y = t;}
}

Pair<String> p = new Pair("world!", "Hello,");
p.swap();
System.out.println(p.x + p.y);
```

Compiler's Translation

```
class Pair {
    Object x;  Object y;
    Pair (Object x, Object y) {this.x = x; this.y = y;}
    void swap () {Object t = x; x = y; y = t;}
}
Pair p = new Pair((Object)"world!", (Object)"Hello,");
p.swap();
System.out.println((String)p.x + (String)p.y);
```

Two ways to handle parameterized types

- Specialization of objects
  - each instance of the parameterized type creates a new representation. List<Integer> and List<Float> are two different representations of List<T>
- Sharing of objects
  - the code for List<T> is generated by the compiler for one representation and all the instances created refer to this representation
- Java uses the second approach
  - Some problems with simple types: **a generic with simple type is not allowed as they are treated differently by the compiler**

Getting an instance of a parametric type

- it is illegal to write (code will not compile)

new T();

- where T is a parametric type as we do not know the true type of the object and as such we cannot call its constructor

Static generic type class and method

- A static member cannot be implemented as generics
- This is because it is shared by all the objects and the objects of a generic type are of unknown type

Example

```
interface MinMax<T extends Comparable<T>> {
   T min();
   T max();
 }
class MyClass<T extends Comparable<T>>
implements MinMax<T> {
   T[] vals;
   MyClass(T[] o) { vals = o; }

   public T min() {
       T v = vals[0];
       for(int i=1; i < vals.length; i++){
           if(vals[i].compareTo(v) < 0) v = vals[i];
       }
           return v;
   }

   public T max() {
       T v = vals[0];
       for(int i=1; i < vals.length; i++){
           if(vals[i].compareTo(v) > 0) v = vals[i];
       }
       return v;
   }
}
```

```
public class GenIFDemo {

  public static void main(String args[]) {

    Integer inums[] = {3, 6, 2, 8, 6 };

    Character chs[] = {'b', 'r', 'p', 'w' };

    MyClass<Integer> iob = new MyClass<Integer>(inums);

    MyClass<Character> cob = new MyClass<Character>(chs);


    System.out.println("Max value in inums: " + iob.max());
    System.out.println("Min value in inums: " + iob.min());
    System.out.println("Max value in chs: " + cob.max());
    System.out.println("Min value in chs: " + cob.min());
    }

 }
```