# Java

## Generics

# Generics

- Since JDK 1.5 (Java 5), the Collections framework has been parameterized.
- A class that is defined with a parameter for a type is called a generic or a parameterized class. In C++, there were referred to as template classes.
- If you compare the Collection interface in the API for 1.4.2 to the one in version 1.5.0, you will see the interface is now called Collection<E>.

# Collection <E> Interface

- The E represents a type and allows the user to create a homogenous collection of objects.
- Using the parameterized collection or type, allows the user to retrieve objects from the collection without having to cast them.

Before:

```
List c = new ArrayList();
c.add(new Integer(34));
Integer i = (Integer) c.get(0);
```

After:

```
List<Integer> c = new ArrayList<Integer>();
c.add(new Integer(34));
Integer i = c.get(0);
```

# Generic Cell Example

```
public class CellDemo
{

   public static void main (String[ ] args)
   {
        // define a cell for Integers
        Cell<Integer> intCell = new Cell<Integer>( new Integer(5) );

        // define a cell for Floats
        Cell<Float> floatCell = new Cell<Float>( new Float(6.7) );

        // compiler error if we remove a Float from Integer Cell
        Float t = (Float)intCell.getPrisoner( );
        System.out.println(t);
   }
}
class Cell< T >
{
   private T prisoner;
   public Cell( T p)
   { prisoner = p; }
   public T getPrisoner(){return prisoner; }
}
```

# Dont's of Generic Programming

- Like C++, you CANNOT use a parameter in a constructor.

```
T obj = new T();
T [] array = new T[5];
```

- Like C++, you CANNOT create an array of a generic type.

```
Collection <Integer> c[ ] =
            new Collection<Integer>[10];
```

# Do's of Generic Programming

- The type parameter must always represent a reference data type.
- Class name in a parameterized class definition has a type parameter attached.

```
class Cell<T>
```

- The type parameter is not used in the header of the constructor.

```
public Cell( )
```

- Angular brackets are not used if the type parameter is the type for a parameter of the constructor.

```
public Cell3(T prisoner );
```

- However, when a generic class is instantiated, the angular brackets are used

```
List<Integer> c = new ArrayList<Integer>();
```

# Bounding the Type

- You will see in the API a type parameter defined as follows <? extends E>. This restricts the parameter to representing only data types that implement E, i.e. subclasses of E

```
boolean addAll(Collection<? extends E> c)
```

# Bounding Type Parameters

- The following restricts the possible types that can be plugged in for a type parameter **`T`**.

```
public class RClass<T extends Comparable>
```

  - "**`extends Comparable`**" serves as a *bound* on the type parameter **`T`**.
  - Any attempt to plug in a type for **`T`** which does not implement the **`Comparable`** interface results in a compiler error message

# More Bounding

- In the API, several collection classes contain <? super T> in the constructor. This bounds the parameter type to any class that is a supertype of T.

```
interface Comparator<T>
{ int compare(T fst, T snd); }

TreeSet(Comparator<? super E> c)
```

# Generic Sorting

```
public class Sort
{
   public static <T extends Comparable<T>>
   void bubbleSort(T[] a)
   {
       for (int i = 0; i< a.length - 1; i++)
           for (int j = 0; j < a.length -1 - i; j++)
             if (a[j+1].compareTo(a[j]) < 0)
             {
                    T tmp = a[j];
                    a[j] = a[j+1];
                    a[j+1] = tmp;
             }
   }
}
```

# Generic Sorting (cont.)

- Given the following:

```
class Animal implements Comparable<Animal> { ...}
class Dog extends Animal { ... }
class Cat extends Animal { ... }
```

- Now we should be able to sort dogs if contains the *compareTo* method which compares animals by weight.
- BUT… bubblesort only sorts objects of type T which extend T. Here the super class implements Comparable.
- New and improved sort on next page can handle sorting Dogs and Cats.

# Generic Sorting (cont.)

```
public class Sort
{
   public static <T extends Comparable<? super T>>
   void bubbleSort(T[] a)
   {
       for (int i = 0; i< a.length - 1; i++)
           for (int j = 0; j < a.length -1 - i; j++)
             if (a[j+1].compareTo(a[j]) < 0)
             {
                    T tmp = a[j];
                    a[j] = a[j+1];
                    a[j+1] = tmp;
             }
   }
}
```