

Collections (the new style – 1.5)

including Generics

Homogeneous Collections

- New Collections framework supports collections that contain a single type of element.
 - The old style allowed you to put any mix of object types into any list, set or map.
- It's rare that we actually want collections that are not homogeneous...

Generics: type parameters

- We are used to dealing with variable parameters (method and constructor parameters).
- You can also write code that involved parameterized types.
 - The *actual* type is not specified in the code.
- For example, suppose you want to write code to sort *things*
 - without defining what a *thing* is!

Important note about Java *types*

- Generics (everything I'm about to talk about) deal with objects only (anything that extends Object somewhere in its inheritance hierarchy).
- You can't do *this stuff* with primitive data types (like `int`, `char`, `boolean`, `double`, etc.)

Type parameter syntax

- When writing *generic* code that can work with many (or even any) object type, you write the code using a place holder (a name) for the data type.
 - The code obviously needs to make sense for all possible data types (you can tell the compiler to place restrictions on the legal data types).
- To specify this *generic* type parameter you do this: **<E>**

Example a generic interface (from java.util)

```
public interface List<E> extends Collection<E>
```

You can write generic classes, interfaces and methods.

- when writing this code you use the type parameter as a type specifier.
- for example:

```
public E get(int index);
```

Same 'E'



Collection Framework (1.5)

- List, Set and Map are *deprecated*
- Now each interface (and concrete classes) are generic containers, parameterized by a *data type*:

List<E>

Set<E>

Map<K, V>

How does this change my code?

- Instead of a `List` (or whatever), you always deal with a `List` of *something*:

```
List<Integer>
```

```
ArrayList<Double>
```

```
SortedSet<Circle>
```

```
Map<String, Integer>
```

```
List oldList = new ArrayList();  
for (int i=0;i<args.length;i++)  
    oldList.add(args[i]);
```

Old Style

```
for (Iterator i = oldList.iterator(); i.hasNext(); ) {  
    String x = (String) i.next();  
    System.out.println( x );  
}
```

```
List oldList = new ArrayList();  
for (int i=0;i<args.length;i++)  
    oldList.add(args[i]);
```

New Style

```
for (Iterator i = oldList.iterator(); i.hasNext(); ) {  
    String x = i.next();  
    System.out.println( x );  
}
```

Big Deal?

- Actually this is a pretty big deal (once you write some significant code you will find the new style much easier to use).
- Summary:
 - Old style: collections of `Objects`
 - New style: collections of *some data type* (*specified at compile time by the programmer*).