
Programming and Software Development
COMP90041

Lecture 12

# Generic Types

**Dr. Thomas Christy**
thomas.christy@unimelb.edu.au

# Generics

- Generic types, or parametric types are types (classes or interfaces) that have parameters
  - Somewhat like methods have parameters
- In Java, the type parameters are always other types
- Arrays are a special parameterised type: you can't declare just an array, only an array of some type
- New in Java 5: you can define classes that take type parameters
- This makes the type system more expressive; allows Java to catch more bugs for you at compile time
- In many cases it allows you to avoid casting

# A Non-generic Example

- Java methods can only return one value; what if you want to return two?
- Answer: return an object holding both
- So it would be useful to have a class that can hold two of anything
  - Then we wouldn't need to define a new class every time we want to return a pair of things
- A variable of type `Object` can hold any object
- So just define a type with two `Object` instance variables

# A Non-generic Pair Class

```
public class WeakPair {
    private final Object first;
    private final Object second;
    public WeakPair(Object first, Object second) {
        this.first = first;
        this.second = second;
    }
    public Object getFirst() { return first; }
    public Object getSecond() { return second; }
    public String toString() {
        return "(" + first.toString() + "," +
                second.toString() + ")";
    }
}
```

# Wrapper Classes

- A primitive value is not an object
- So how can you store a `Pair` of an `int` and a `String`?
- This is what the wrapper classes are for
- Each primitive type has a wrapper class that stores one primitive value
- Each has a one-argument constructor to create the object (boxing)
- Each has a no-argument getter to get back the primitive value (unboxing)
- The wrapper classes are all immutable

# Boxing and Unboxing

- Boxing example:

```
integer I = new Integer(42);
```

- Unboxing example:

```
int i = I.intValue();
```

- `Integer` is a descendent of `Object`, so `I` can be stored in a `WeakPair`:

```
WeakPair p1 = new WeakPair(I, "Everything");
```

# Autoboxing and Autoboxing

- As of Java 5, Java will automatically box and unbox as necessary
- Autoboxing example:

```
Integer I = 42;
```

- Autounboxing example:

```
int i = I;
```

- Java will even autobox to convert a primitive to the `Object` type:

```
WeakPair p1 = new WeakPair(42, "Everything");
```

# What's Wrong with `WeakPair`?

- Weak typing: it's easy to confuse order of arguments, and the compiler can't help

```
WeakPair p1 = new WeakPair(I, "Everything");
WeakPair p2 = new WeakPair("Everything", I);
```

- Java compiler doesn't know correct types of results of `getFirst` and `getSecond`, so you must explicitly down cast

```
int i1 = (int) p1.getFirst();
int i2 = (int) p2.getFirst();
```

# What's Wrong with `WeakPair`?

- Weak typing: it's easy to confuse order of arguments, and the compiler can't help

```
WeakPair p1 = new WeakPair(I, "Everything");
WeakPair p2 = new WeakPair("Everything", I);
```

- Java compiler doesn't know correct types of results of `getFirst` and `getSecond`, so you must explicitly down cast

```
int i1 = (int) p1.getFirst();
int i2 = (int) p2.getFirst();
```

# What's Wrong with `WeakPair`?

- The biggest problem is that mistakes are only discovered at runtime

```
int i1 = (int) p1.getFirst();
int i2 = (int) p2.getFirst();
```

- But `p2.getFirst()` returns a `String`, not an `Integer`
- Can't cast that to an `int`, but compiler doesn't know that
- Only find the problem at runtime

```
Exception in thread "main" java.lang.ClassCastException:
java.lang.String cannot be cast to java.lang.Integer
```

# Generic Classes

- Solution: don't just declare object is a pair, but what it is a pair of
  - `p1` is a `Pair` of `Integer` and `String`
  - `p2` is a `Pair` of `String` and `Integer`
- Allow class declaration to specify parameters
- Form: *`ClassName<Var1,...>`*
- Parameters, enclosed in angle brackets, are variables ranging over types rather than values
- Type variables are often single letters (often `T`), or at least very short
- The parameters *`Var1`*, ... are used as types inside the class definition

# A Generic Pair Class

```
public class Pair<T1,T2> {
    private final T1 first;
    private final T2 second;
    public Pair(T1 first, T2 second) {
        this.first = first;
        this.second = second;
    }
    public T1 getFirst() { return first; }
    public T2 getSecond() { return second; }
    public String toString() {
        return "(" + first.toString() + ","
                + second.toString() + ")";
    }
}
```

# Using a Generic Class

- For a generic class, class name alone is not a type; must supply type arguments
- Form: *ClassName*<*Type1*,...>
- To construct a new object of generic type, specify both type arguments and constructor arguments
- Form:
  new *ClassName*<*Type1*,...>(*expr1*,...)
- *E.g.*:

```
Pair<Integer,String> p1 =
      new Pair<Integer,String>(I, "Everything");
Pair<String,Integer> p2 =
      new Pair<String,Integer>("Everything", I);
```

# Why This is Better

- Java compiler can track types through the code
- Can check that constructor stores right types of arguments in instance variables, because types agree
- Can check that accessors return the right types

```
int i2 = (int) p2.getFirst();
```

- The compiler now knows that `p2.getFirst()` returns a `String` rather than `Object`
- Now compiler points out the error:

```
PairTest.java:18: error: incompatible types: String can
not be converted to int
        int i2 = (int) p2.getFirst();
                                   ^
```

# QuickPoll:

## Which one of these is illegal?

Given these declarations:

```
Integer i;
int j;
String s, t;
Pair<String,Integer> p, q;
```

A. `new Pair<Integer,Integer>(i,j);`
B. `String u = q.getFirst();`
C. `new Pair<Integer,int>(i,j);`
D. `int k = p.getSecond();`
E. `new Pair<Pair<String,Integer>,String>(p,s);`

qp.unimelb.edu.au/tchristy

# QuickPoll:

## Which one of these is illegal?

Given these declarations:

```
Integer i;
int j;
String s, t;
Pair<String,Integer> p, q;
```

A. `new Pair<Integer,Integer>(i,j);`
B. `String u = q.getFirst();`
C. `new Pair<Integer,int>(i,j);`
D. `int k = p.getSecond();`
E. `new Pair<Pair<String,Integer>,String>(p,s);`

# Generic Constructor Syntax

- To use a generic type constructor, give type parameters, *e.g.*:

```
Pair<Integer,String> p1 =
      new Pair<Integer,String>(I, "Everything");
```

- But define constructor without type parameter:

```
public Pair(T1 first, T2 second) {
    this.first = first;
    this.second = second;
}
```

# Can't Instantiate Parameter Type

- Inside generic class, Java knows nothing about type variable
- So you can't use its constructor; this won't work:

```
public Pair() {
    this.first = new T1();
    this.second = new T2();
}
```

```
Pair.java:13: error: unexpected type
        this.first = new T1();
                         ^
  required: class
  found:    type parameter T1
  where T1 is a type-variable:
    T1 extends Object declared in class Pair
```

# Can't Create Generic Type Array

- You can't make an array with generic element type

```
public Pair(T1 first) {
    T1[] a = new T1[1];
    a[0] = first;
}
```

```
Pair.java:13: error: generic array creation
        T1[] a = new T1[1];
                 ^
```

- This is a perfectly reasonable thing to want to do
- Java doesn't allow it because it throws away all information about generics during compilation
- Workaround is beyond the scope of this subject

# Comparable Done Right

- The `Comparable` interface as discussed earlier works, but is clumsy
- It does not ensure in `o1.compareTo(o2)` that `o1` and `o2` are of same class
- A class that implements `Comparable` also needs to check that argument is right type and cast object
- As of Java 5, `Comparable` is a generic interface
- `Comparable<T>` means a type that can be compared with type `T`

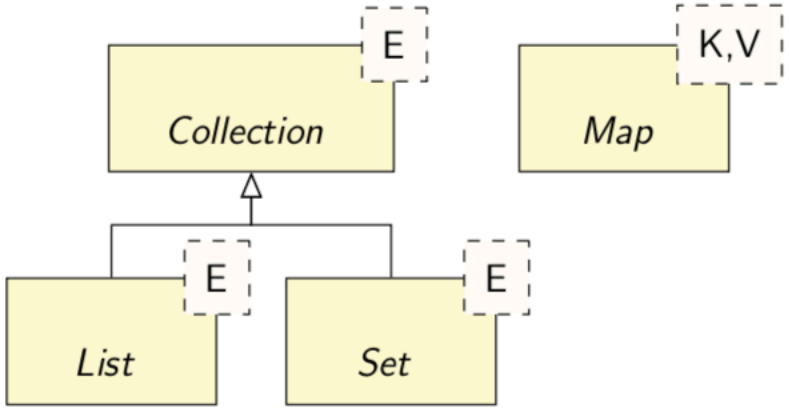
# Implementing a Comparable Class

- Make `Person` comparable
  - Order first by name; for same name, order by age

```
public class Person implements Comparable<Person> {
    private int age;
    private String name;
    :
    public int compareTo(Person o) {
        int result = name.compareTo(o.name);
        if (result == 0) {
            result = age - o.age;
        }
        return result;
    }
}
```

# Java Collections Framework

- Java library provides several convenient classes and interfaces for working with collections of objects
- These are the most important interfaces in the hierarchy:




- All are generic; template parameters are shown in dotted boxes

# `Collection<E>` interface

- All `Collection` classes have no-argument constructor returning empty collection
- All are expandable to unbounded size
- All support foreach loop syntax
- All support these methods:

| Method | Action / Result |
|---|---|
| `boolean isEmpty()` | Is collection empty? |
| `int size()` | Number of elements in collection |
| `boolean add(E e)` | Add `e`; `true` if collection changed |
| `boolean contains(Object o)` | Does collection contain `o`? |
| `boolean remove(Object o)` | Remove `o` if present; `true` if so |
| `void clear()` | Remove all elements |
| `Object[] toArray()` | Collection content as an array |

# List<E> interface

- List<E> is a collection that preserves order and duplicates; similar to an array
- In addition to methods of Collection<E>, List<E> supports these:

| Method | Action / Result |
|---|---|
| `E get(int i)` | Return element at index `i` |
| `E set(int i, E elt)` | Replace obj at index `i` with `elt`; return old |
| `int indexOf(Object o)` | Return first index of `o`, or -1 if absent |
| `int lastIndexOf(Object o)` | Return last index of `o`, or -1 if absent |
| `void add(int i, E elt)` | Insert `elt` at index `i` |
| `E remove(int i)` | Remove and return object at index `i` |

# Most important `List<E>` Classes

- `ArrayList<E>` — usually the class you want
  - Fast to add to the end
  - Fast to get element by index
  - Slow to add in the middle
- `LinkedList<E>`
  - Fast to add to either end
  - Fast to insert anywhere
  - Slow to get element by index
- Only matters when you have thousands of elements
- Use `List<E>` interface as variable and parameter type — then code will work for either class of list
- But you must use `ArrayList<E>` or `LinkedList<E>` class with `new`

# QuickPoll:

## What Will This Print?

```
ArrayList<String> list = new ArrayList<String>();
list.add("one");
list.add("two");
list.add(1, "three");
list.add(1, "four");
for (String s : list) System.out.print(s + " ");
System.out.println();
```

A. one two three four
B. four three two one
C. four three one two
D. one four three two
E. three four one two

bit.ly/tchristy or qp.unimelb.edu.au/tchristy

# What Will This Print?

```
ArrayList<String> list = new ArrayList<String>();
list.add("one");
list.add("two");
list.add(1, "three");
list.add(1, "four");
for (String s : list) System.out.print(s + " ");
System.out.println();
```

A. one two three four
B. four three two one
C. four three one two
D. one four three two
E. three four one two

# `Set<E>` interface

- `Set<E>` does not preserve order or duplicates
- `Set<E>` does not support methods beyond those of `Collection<E>`
- `HashSet<E>` implements `Set<E>` — usually the class you want
  - Fast to add and remove elements
  - Fast to check if object is in set (faster than either kind of `list<E>`)

# `Map<K,V>` interface

- A map is sometimes called a dictionary or lookup table or finite function
- It associates a single value of type `V` with each key of type `K`
- *E.g.*, it could let you quickly look up the `Person` object (value) with a given name (key)
- This is very much faster than checking every `Person` object looking for the right name

# `Map<K,V>` interface methods

| Method | Action / Result |
|---|---|
| `Boolean isEmpty()` | Is the map empty? |
| `int size()` | The number of key-value mappings |
| `V get(Object key)` | Return object mapped to by `key`, or `null` |
| `V put(K key, V value)` | Make `value` the mapping for `key`; return old |
| `V remove(Object key)` | Remove and return the mapping for `key`, if any |
| `void clear()` | Empty the map |

- Most common map class is `HashMap<K,V>` — good performance

# Summary

- Generic types have types as parameters written in `<Angle,Brackets>`
- In class declaration, parameters are type variables, which can be used as types in the declaration
- In variable declaration, parameters should be reference types
- All primitive types have wrapper classes; Java will convert
- Usually use `ArrayList` for lists, `HashSet` for sets, `HashMap` for maps


THE UNIVERSITY OF
MELBOURNE