Object Oriented Programming

Week 4 Part 3
Generics

Lecture

- What are generics
- Examples of the use of generics

What are generics?

Generics

- Java Generics let you specify types when defining classes, interfaces and methods
  - They use *type parameters*
    - Specify types such as Dog rather than objects such as Rex
- Generics are an example of parametric polymorphism.
  - The behavior of the method, class, etc is determined by a parameter

Java Generics

- Specified by angle brackets, “<>”
  - e.g. ArrayList<Wolf> wolves;
- ArrayList uses Generics to indicate the type of object stored
  - The actual storage is the a reference to the ArrayList class
  - The ArrayList Class has a field that contains the objects

ArrayList uses Generics

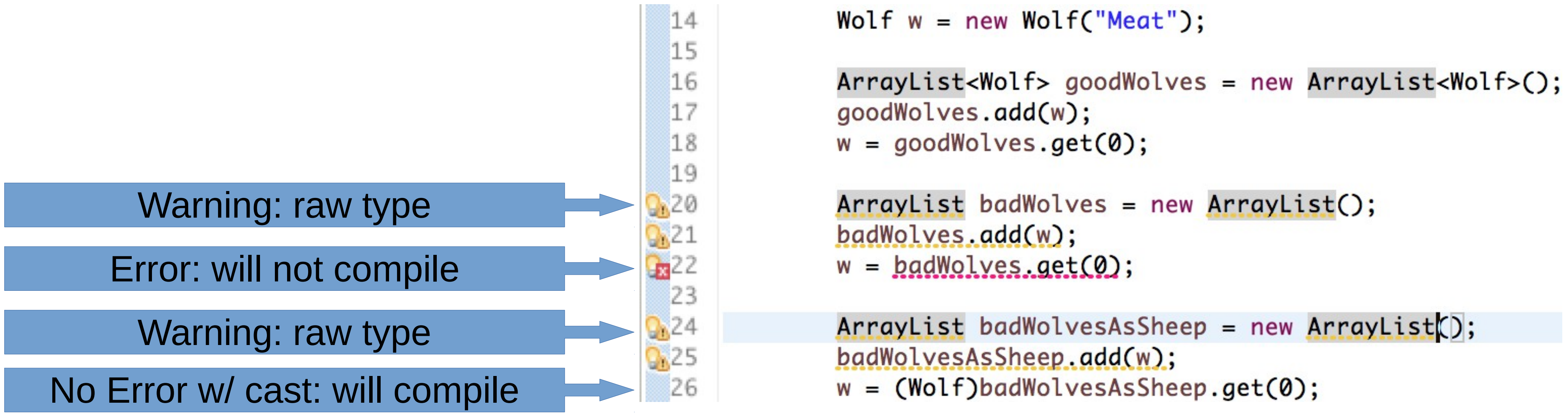
- ArrayList uses Generics to indicate the type of object stored
  - The actual storage is the a reference to the ArrayList class
  - An ArrayList has a field that contains the objects
    - The storage of this field is an collection of references to the objects contained
    - All objects take the same space in the same
    - The field may be an array, list, …, the programmer doesn't know or care
- By using generics ArrayList can check that the object being stored is the correct type

Advantages of Generics (1)

- The biggest advantage of generics is the compiler can do type checking
  - E.g, you cannot accidentally assign a Deer to a Pack of wolves.
    - The compiler will catch the error
  - Errors are possible because all references are the same size, so it is possible to assign an object to any array

Advantages of Generics (2)

- A secondary advantage is you do not need to cast variables when assigning from an array
- ArrayList without a type is a “raw type”
  - Allowed for backward compatibility
  - Requires explicit cast “(Wolf)” to assign to var



```
        Wolf w = new Wolf("Meat");

        ArrayList<Wolf> goodWolves = new ArrayList<Wolf>();
        goodWolves.add(w);
        w = goodWolves.get(0);

        ArrayList badWolves = new ArrayList();
        badWolves.add(w);
        w = badWolves.get(0);

        ArrayList badWolvesAsSheep = new ArrayList();
        badWolvesAsSheep.add(w);
        w = (Wolf)badWolvesAsSheep.get(0);
```

Generic Classes

Defining Generic Classes

- ArrayList is a Generic Class
- A generic class, myClass is defined as
  - public class myClass<T> { … }
  - The T represents a class
  - The symbol T may be used anywhere a type would be used
    - e.g., T myField;
    - e.g., T getMyField() { … }

Multiple type Generics

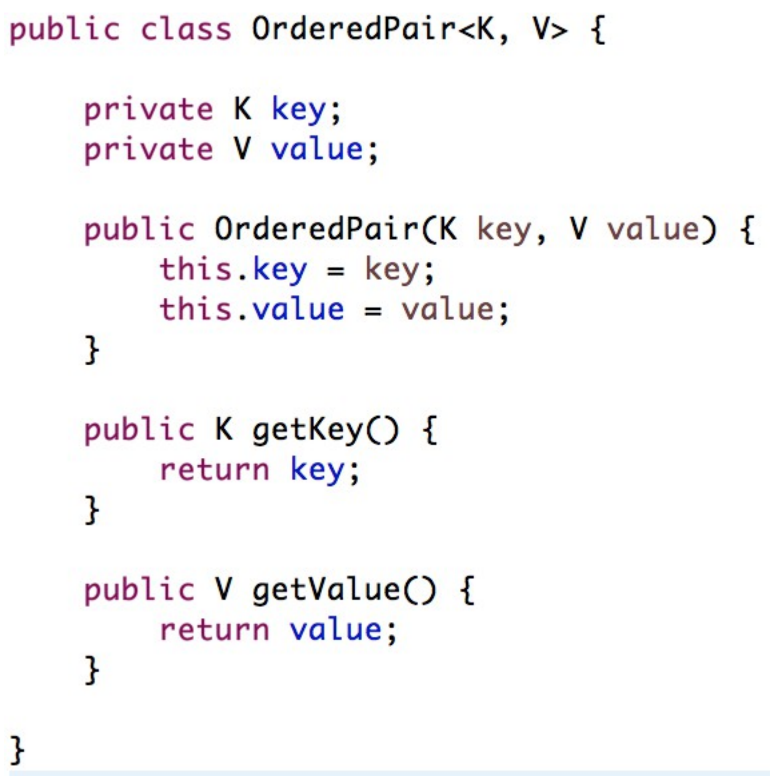
- May define a type based on multiple types
  - e.g. public class myClass <T1, T2, T3, … Tn> { … }
- By convention types in generics are referred to by a single upper case letter
  - E: Element
  - K: Key
  - N: Number
  - T: Type
  - V: Value
  - S, U, V: additional Types

Common Multiple Type Generics

- Multiple Type Generics appear most commonly in key value pairs
- To store pairs we might generate a class:
  - e.g., public class OrderedPair<K V> { … }
- We use the class by adding classes for K and V
  - e.g., OrderedPair<Integer, String> op;

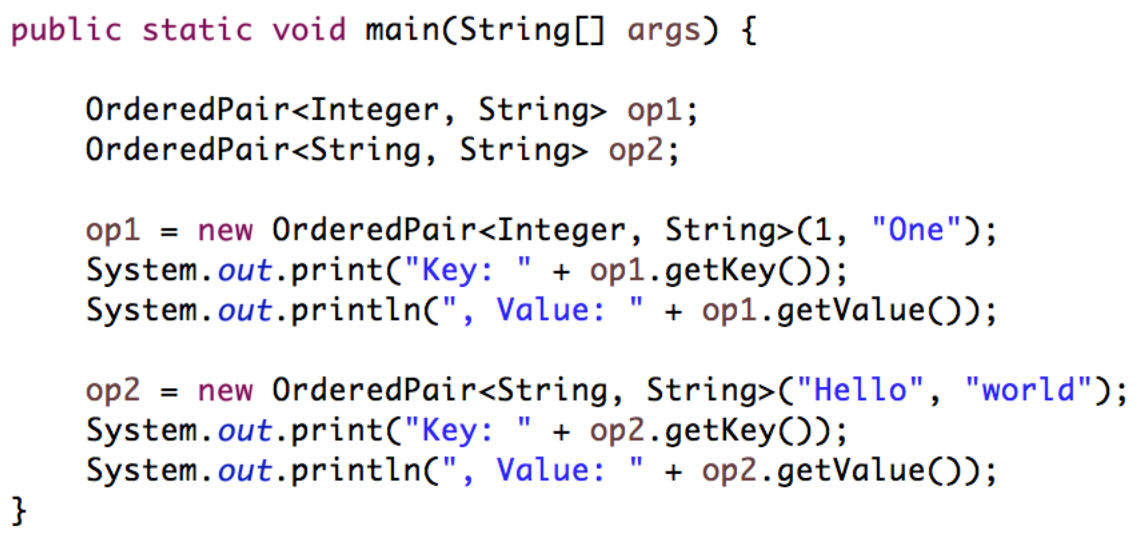
Multiple Type Generic Example

- OrderedPair



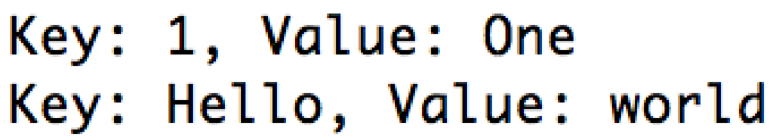
```java
public class OrderedPair<K, V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }

}
```

- Using OrderedPair



```java
public static void main(String[] args) {

    OrderedPair<Integer, String> op1;
    OrderedPair<String, String> op2;

    op1 = new OrderedPair<Integer, String>(1, "One");
    System.out.print("Key: " + op1.getKey());
    System.out.println(", Value: " + op1.getValue());

    op2 = new OrderedPair<String, String>("Hello", "world");
    System.out.print("Key: " + op2.getKey());
    System.out.println(", Value: " + op2.getValue());
}
```

- Output



```
Key: 1, Value: One
Key: Hello, Value: world
```

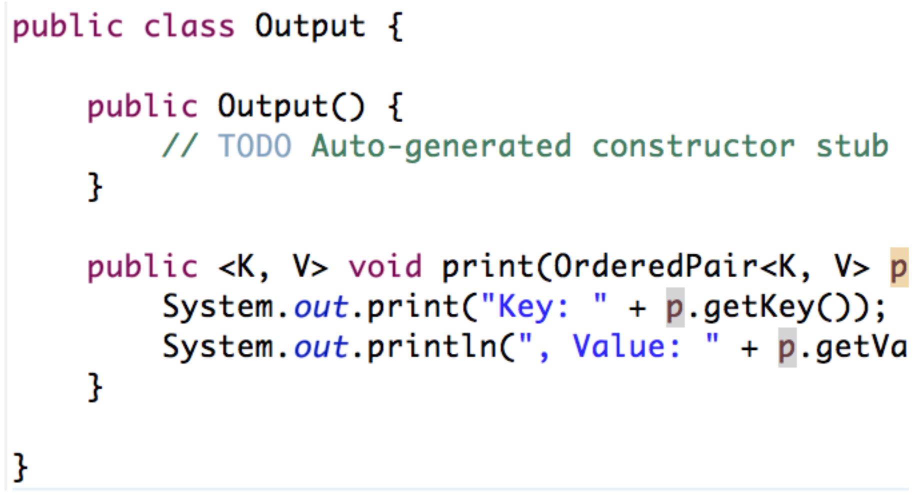
Generic Methods

Generic Methods

- We can create generic methods outside a generic class
- For example, we can create a print method in an Output class that can print a OrderedPair

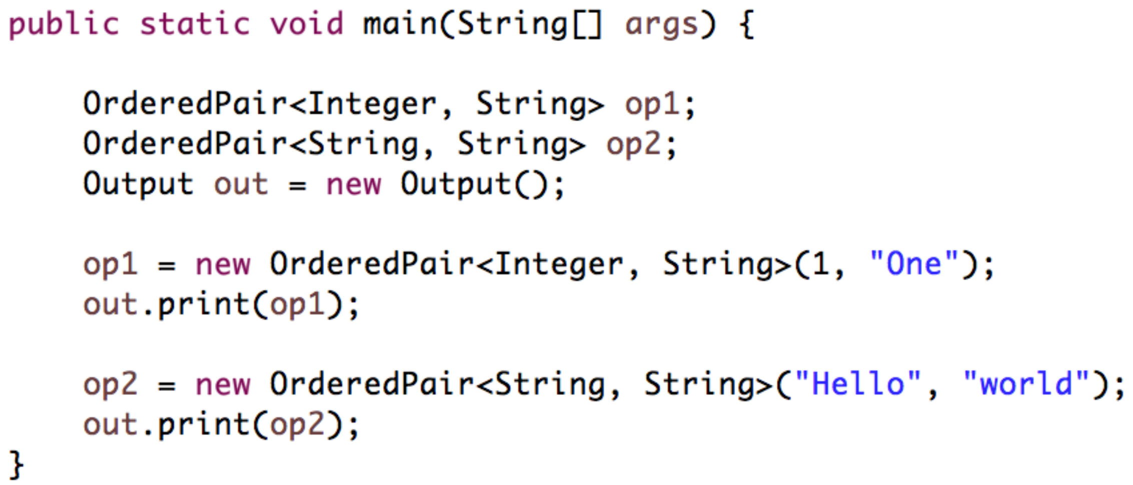
Generic Methods Example

- Output class



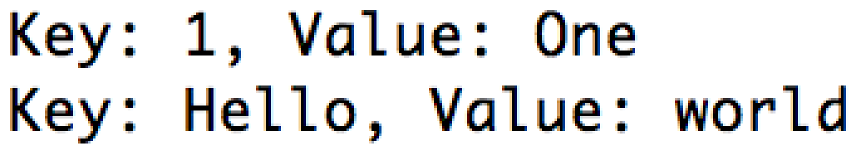
```java
public class Output {

    public Output() {
        // TODO Auto-generated constructor stub
    }

    public <K, V> void print(OrderedPair<K, V> p
        System.out.print("Key: " + p.getKey());
        System.out.println(", Value: " + p.getVa
    }
}
```

- Calling print



```java
public static void main(String[] args) {

    OrderedPair<Integer, String> op1;
    OrderedPair<String, String> op2;
    Output out = new Output();

    op1 = new OrderedPair<Integer, String>(1, "One");
    out.print(op1);

    op2 = new OrderedPair<String, String>("Hello", "world");
    out.print(op2);
}
```

- Output



```
Key: 1, Value: One
Key: Hello, Value: world
```

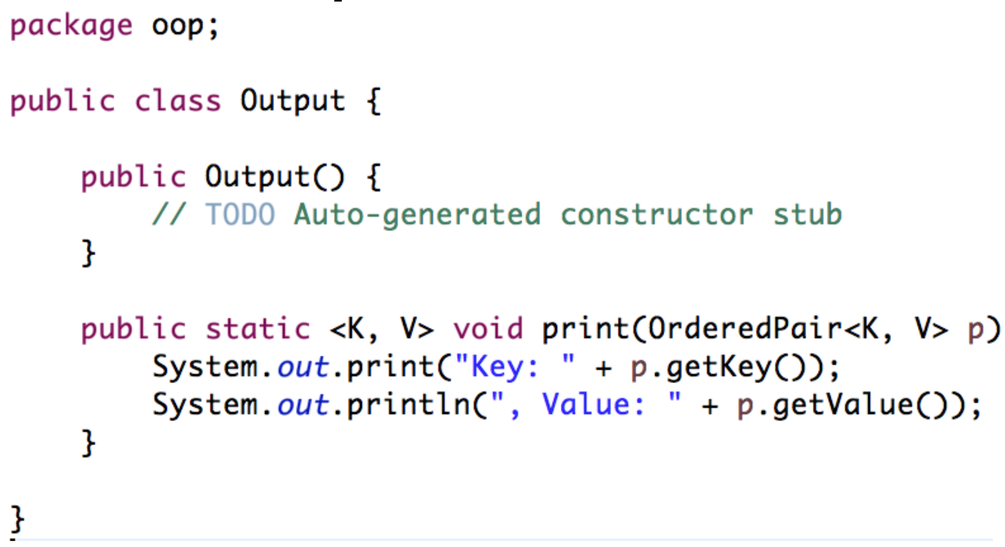
Static Generic Methods

Static print method

- To use the Output class as defined, we need to create an Output object.
- The object adds nothing to the behavior of the print method
  - A better solution is to make the method static
- A static method can be called from the class, not from an object of the class

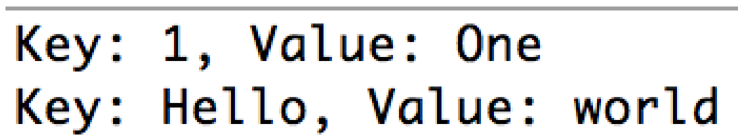
Changing print to static

- Output class



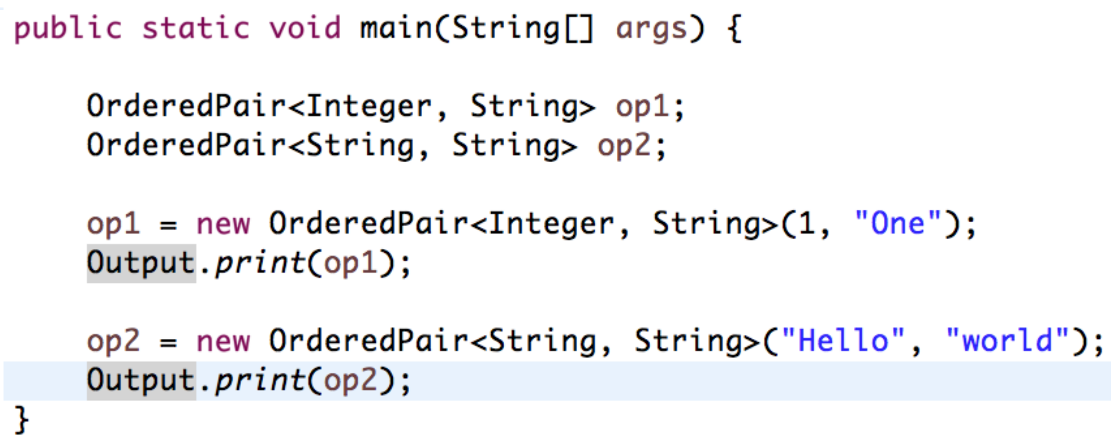
```
package oop;

public class Output {

    public Output() {
        // TODO Auto-generated constructor stub
    }

    public static <K, V> void print(OrderedPair<K, V> p) {
        System.out.print("Key: " + p.getKey());
        System.out.println(", Value: " + p.getValue());
    }

}
```

- Output



```
Key: 1, Value: One
Key: Hello, Value: world
```

- Calling Output.print



```
public static void main(String[] args) {

    OrderedPair<Integer, String> op1;
    OrderedPair<String, String> op2;

    op1 = new OrderedPair<Integer, String>(1, "One");
    Output.print(op1);

    op2 = new OrderedPair<String, String>("Hello", "world");
    Output.print(op2);
}
```