Module 7

Java Generics

Generics

Before Java 5 – Object used as a universal class

```
public class ArrayList
{
     public Object get(int i) { . . . }
     public void add(Object o) { . . . }
     . . .
     private Object[] elementData;
}
```

Generics

Using Object - problems

- ClassCastException
- type casting

```
List array = new ArrayList();
array.add(10);
array.add("Str");
for (Object o : array) {
      Integer number = (Integer) o;
}
```

Generics

Generic is simple

- check errors on stage of compilation
- no type casting

```
List<Integer> array = new ArrayList<Integer>();
array.add(10);
// array.add("Str");
for (Integer o : array) {
	Integer number = o;
}
```

Generics

Generic class example

```
class Pair<T> {
    private T first;
    private T second;

    public Pair() { first = null; second = null; }
    public Pair(T first, T second) {
        this.first = first; this.second = second; }
    public T getFirst() { return first; }
    public T getSecond() { return second; }
    public void setFirst(T newValue) { first = newValue; }
    public void setSecond(T newValue) { second =
newValue; }
}
```

Generics

Use of generic class

```
public static void main(String[] args) {
    Pair<String> pair = new Pair<String>("Java", "World");
    System.out.println(pair.getFirst() + pair.getSecond());
}
```

Generics

Generic class example

```
class Pair<T> {
    private T first;
    private T second;

    public Pair() { first = null; second = null; }
    public Pair(T first, T second) {
        this.first = first; this.second = second; }
    public T getFirst() { return first; }
    public T getSecond() { return second; }
    public void setFirst(T newValue) { first = newValue; }
    public void setSecond(T newValue) { second =
newValue; }
}
```

Generics

Type erasing

```
class Pair {
    private Object first;
    private Object second;
    public Pair() { first = null; second = null; }
    public Pair(Object first, Object second) {
        this.first = first; this.second = second; }
    public Object getFirst() { return first; }
    public Object getSecond() { return second; }
    public void setFirst(Object newValue) { first =
newValue; }
    public void setSecond(Object newValue) {
        second = newValue; }
}
```

Generics

Restricting of T

```
class Pair <T extends String> {
        . . .
}
// After erasing with the restriction:
class Pair {
        private String first;
        private String second;
        public Pair() { first = null; second = null; }
        public Pair(String first, String second) {
                this.first = first; this.second = second; }
        . . .
}
```

Generics

Restrictions

- **work with primitive types**
  Pair<int> // error

- **get type at execution time**
  a instanceof Pair<Date> == a instanceof Pair<Integer>

- **generic type cannot extend Throwable**
  class Problem<T> extends Exception

- **cannot be used in catch**
  catch(T t) // Error

Generics

Restrictions

> Generic type **instance cannot be created**

```
public Pair() {
	first = new T();
	second = new T();
} // Error
```

Class<T> can be used for that:

```
public static <T> Pair<T> makePair(Class<T> cl) {
	try {
		return new Pair<T>(cl.newInstance(),
		cl.newInstance())
	} catch (Exception ex) { return null; }
}
```

Generics

Generic method

```
class ArrayAlg {
    public static <T> T getMiddle(T[] a) {
        return a[Math.round(a.length / 2)];
    }
}
```

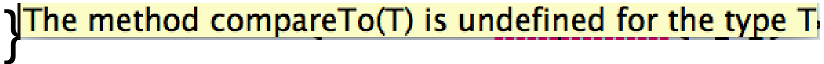
Usage:

```
String[] names = { "John", "Q.", "Public" };
String middle = ArrayAlg.getMiddle(names); // Java 6+
```

Generics

Restriction: T should extend Comparable

```
class ArrayAlg {
    public static <T> T min(T[] a) {
        if (a == null || a.length == 0) {
            return null; }
        T smallest = a[0];
        for (int i = 1; i < a.length; i++) {
            if (smallest.compareTo(a[i]) > 0) {
                smallest = a[i]; }
        }
        return smallest;
    }
}
```



The method compareTo(T) is undefined for the type T

Generics

Restriction: T extends Comparable

```
class ArrayAlg {
    public static <T extends Comparable> T min(T[] a) {
        if (a == null || a.length == 0)
            return null;
        T smallest = a[0];
        for (int i = 1; i < a.length; i++)
            if (smallest.compareTo(a[i]) > 0)
                smallest = a[i];
        return smallest;
    }
}
```

Generics

Limitations

```
class ArrayAlg {
     public static <T extends Comparable & Serializable>
T
      ....
}
```

Generics

Static context

> **You cannot use generics in static context**

```
public class Singleton<T> {
    private static T singleInstance; // Error
    public static T getSingleInstance() { // Error
        if (singleInstance == null) // construct new instance of
    T
            return singleInstance;
    }
}
```

If static fields of type parameters were allowed, then the following code would be confusing:

```
    Singleton<Bank> bank = new Singleton<>();
    Singleton<Client> client = new Singleton<>();
    Singleton<Account> account = new Singleton<>();
```

Because the static field singleInstance is shared by bank, client, and account, what is the actual type of singleInstance? It cannot be Bank, Client and Account at the same time.

You cannot, therefore, create static fields of type parameters.

Generics

Problems with erasing – it’s not allowed:

```
public class Pair<T> {
    public boolean equals(T value) {
        return first.equals(value) &&
    second.equals(value);
    }
. . .
}
```

Generics

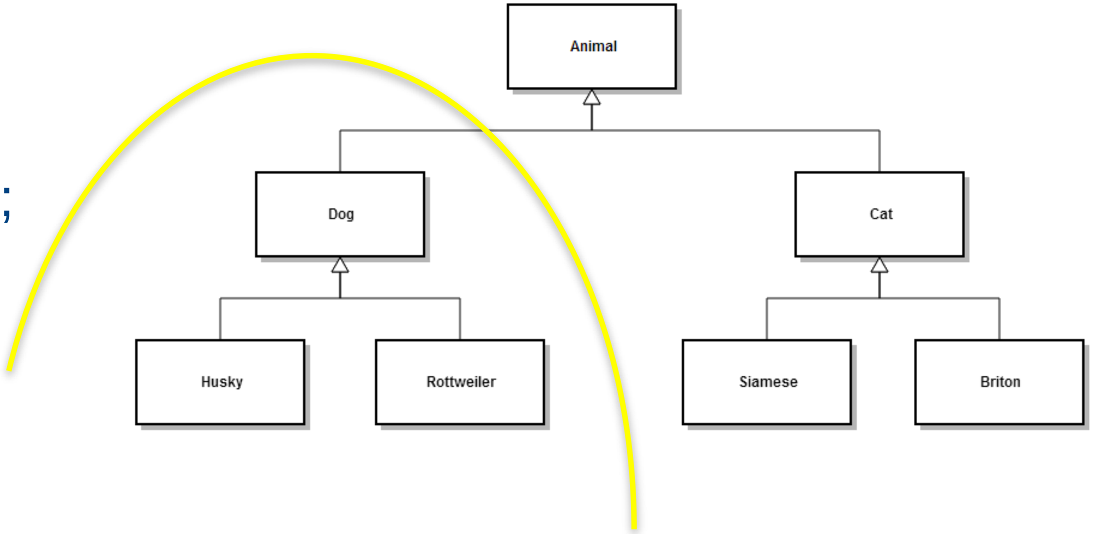
Problems with erasing

For Pair<String> will be 2 implementations:
boolean equals(String) // defined in Pair<T>
boolean equals(Object) // inherited from Object

But on erasing we get T -> Object.
2 same methods? Disallowed.

Wildcards (classes)

```
class Box <T extends Dog>  {
        private T dog;
        Box (T dog) {
                this. dog = dog;
        }
}
```

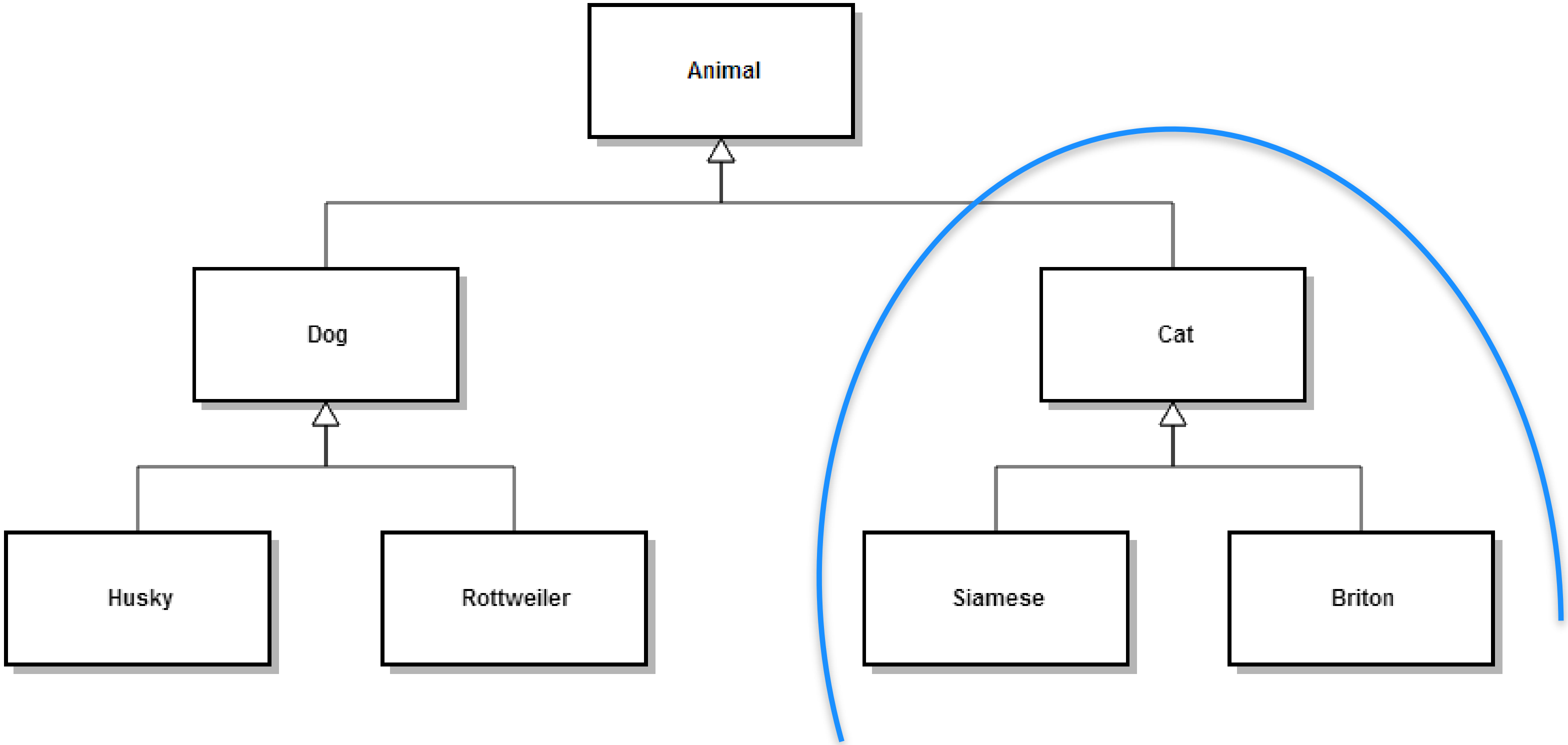


Wildcards

```
class ClassName <Type extends A & B & C & D>

class Process <T extends Entity & Serializable & Comparable>

public class TreeMap<K, V> extends AbstractMap<K, V>
```

Bounded Wildcard

```
void doSomething ( Box < ? extends Cat > ) {
            //method takes only boxes with cats and its descendants
}
```



Bounded Wildcards (problem)

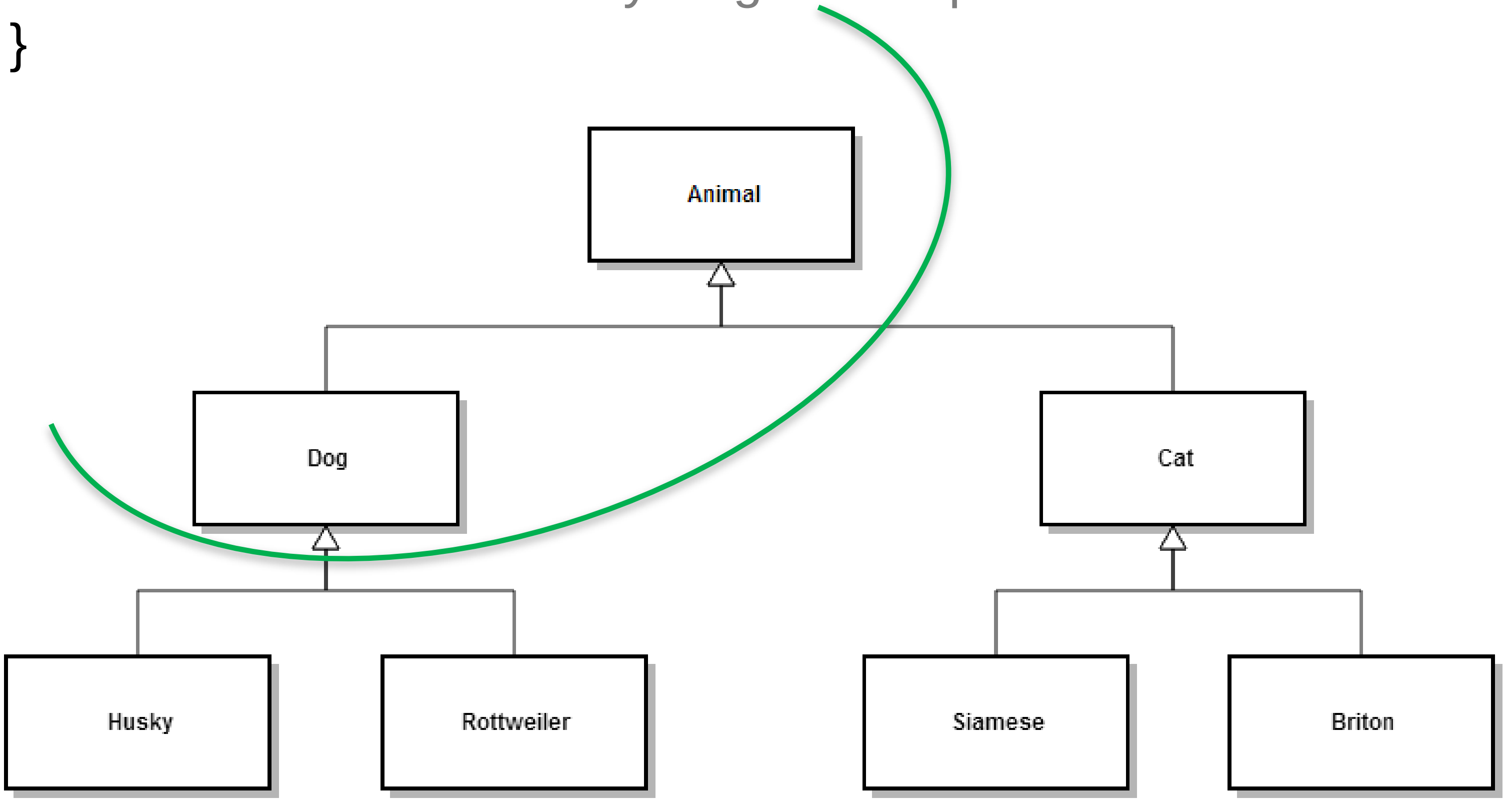
```
void draw(List<Shape> c) {
        for (Iterator<Shape> i = c.iterator(); i.hasNext(); ) {
            Shape s = i.next();
            s.draw();
        }
    }

List<Shape> l;
draw(l); // ok
List<Circle> l;
draw(l); // compile error
```

Bounded Wildcards (solution)

```
void draw(List<? extends Shape> c) {
        for (Iterator< ? extends Shape > i = c.iterator(); i.hasNext(); ) {
            Shape s = i.next();
            s.draw();
        }
    }
```

Wildcards (methods)

```
void doSomething ( Box < ? super Dog> ) {
        //method take only Dog and its parents
}
```



Wildcards (methods)

? Super example

```
public class Collections {
  public static <T> void copy
    ( List<? super T> dest, List<? extends T> src) {
      for (int i=0; i<src.size(); i++)
        dest.set(i,src.get(i));
  }
}
```

- *PECS principle*: **"**Producer Extends, Consumer Super**“**
- The Get and Put Principle: use an extends wildcard when you only get values out of a structure, use super wildcard when you only put values into a structure, and don't use a wildcard when you both get and put.

Exercise

Lab guide:

- Exercise 14
- Exercise 15