CECS 277 – Lecture 13 – Generics

Generic Type Parameter Names – When defining a generic type parameter name any legal identifier may be used. However, by convention, they are usually a single capital letter. This helps to distinguish them from actual class names. The letter T is most common because it represents a general type. The following are other commonly used generic parameter type names and their uses:

Generic Name	Usual Meaning
T	Represents a general type.
S	Represents a general type if T has already been used.
E	Represents an Element of a container type.
K	Represents a Key for a class that uses a key/value pair.
V	Represents a Value for a class that uses a key/value pair.

Defining Multiple Generic Types – A class doesn't have to be restricted to just a single generic type. A class can accept multiple generic type parameters, allowing it to define several different class types within the class.

```
public class Pair <S, T> {
     private S first;
     private T second;
     public Pair( S f, T s ) {
          setPair( f, s );
     }
     public S getFirst( ) {
          return first;
     }
     public T getSecond( ) {
          return second;
     }
     public void setPair( S f, T s ) {
          first = f;
          second = s;
     }
}

public class TestPair {
     public static void main( String [] args ) {
          Pair<String, Double> fruit = new Pair<String,
               Double>( "Apple", 0.79 );
          Pair<String, Integer> employee = new
               Pair<String, Integer>( "John", 12345 );
          Pair<String, String> animal = new Pair<String,
               String>( "Cat", "Meow" );
          System.out.println( fruit.getFirst() );
          System.out.println( fruit.getSecond() );
```

Wildcard Types – If you know what type a generic should be when you pass it to, or return from a method, then use that type, but if you do not know what types the generic object will have, then you can use a wildcard type instead by using a `?`.

Example: Method passing in a generic object with a defined type

```
public static int multiplyPair( Pair <Integer> p ) {
     return p.getFirst() * p.getSecond();
}
```

Example: Method passing in a generic object with a wildcard type

```
public static void displayPair( Pair <?> p ) {
     System.out.println( "A = " + p.getFirst() );
     System.out.println( "B = " + p.getSecond() );
}
```

You can't assume anything about the wildcard's type. All you know is that a Pair of any type is being passed in to the method. Use the ? type if you are only using functionality from the Object class, or methods of the generic class that don't depend on type.

Why Use Wildcards? – wildcards are often used because of a problem with generic type containers. If a parameter is expecting an ArrayList<Animal>, and you pass it an ArrayList<Dog>, you might expect that since Dog extends Animal, it would be fine. Unfortunately, it doesn't work. Instead, a wildcard type is needed, ArrayList<?>. If you need access to methods of the generic's type, then constraints can be used to limit the types the wildcard can take.

Bounding Wildcards – wildcards can be constrained using an upper or lower bound so that only certain types can be used in your method. An upper bound is created by using the keyword extends, and a lower bound using super. A wildcard may not use both an upper bound and lower bound.

Example: Generic parameter with an upper bounded wildcard – allows pairs with any types extending from Animal.

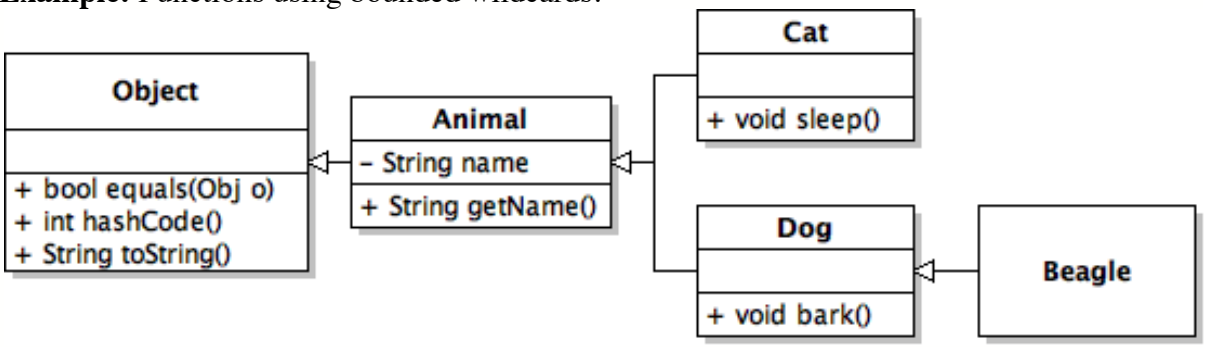
```
public static void dispAnim(Pair<? extends Animal> p){
     System.out.println("A1="+p.getFirst().getName());
     System.out.println("A2="+p.getSecond().getName());
}
```

Example: Generic parameter with a lower bounded wildcard – allows for pairs with types that Dog has extended from, such as Animal and Object.

```
public static void setDog( Pair<? super Dog> p ) {
     p.setPair( new Dog("Spot"), new Dog( "Woofy") );
}
```

In general, you should use an upper bound when an argument is an input parameter (when it is being used by the method), and you should use a lower bound when it is an output parameter (when the method modifies it). If the parameter can be used in the method regardless of type, then it should be unbounded.

Example: Functions using bounded wildcards.




```
import java.util.ArrayList;
public class Main {
  public static void main( String[] args ) {
    ArrayList<Animal> animals = new ArrayList<Animal>();
    ArrayList<Cat>cats = new ArrayList<Cat>();
    ArrayList<Dog>dogs = new ArrayList<Dog>();
    ArrayList<Beagle> beagles= new ArrayList<Beagle>();

    addAnimals( animals );

    addDogs( animals );
    addDogs( dogs );

    addDoggs( dogs );
    addDoggs( beagles );

    printBark( dogs );
    printBark( beagles );

    printNames( animals );
    printNames( cats );
    printNames( dogs );
    printNames( beagles );
  }

  //can only pass in lists of type Animal
  //can add any type of Animal (Animal, Dog, Cat, Beagle)
  //can use functions of Animal or Object
  public static void addAnimals( ArrayList<Animal> list ) {
    list.add( new Animal( "A" ) );
    list.add( new Cat( "C" ) );
    list.add( new Dog( "D" ) );
    list.add( new Beagle( "B" ) );
    System.out.println( list.get(0).getName() );
    //System.out.println( list.get(1).sleep() );
  }
```

```
  //can pass in lists of type Dog, Animal, or Object
  //can add any type of Dogs (Dog or Beagle)
  //can use functions of Object
  public static void addDogs(ArrayList<? super Dog> list){
      //list.add( new Animal( "A" ) );
      //list.add( new Cat( "B" ) );
      list.add( new Dog( "D" ) );
      list.add( new Beagle( "B" ) );
      System.out.println(list.get(0).equals( list.get(1)));
  }

  //can pass in list of any type of Dog (Dog, Beagle)
  //can't populate with anything
  //can use functions of Dog or above (Dog, Animal, Object)
  public static void addDoggs(ArrayList<? extends Dog> list){
      //list.add( new Dog( "D" ) );
      //list.add( new Beagle( "B" ) );
  }

  //can pass in list of any type of Dog (Dog, Beagle)
  //can't populate with anything
  //can use functions of Dog or above (Dog, Animal, Object)
  public static void printBark(ArrayList<? extends Dog>list){
    for( Dog d: list ) {
      System.out.print( d.getName() + " ");
      d.bark();
    }
  }

  //can pass in list of any type of Animal
  //(Animal, Cat, Dog, Beagle)
  //can't populate with anything
  //can use functions of Animal or above (Animal, Object)
  public static void printNames(ArrayList<? extends Animal> list){
    for( Animal a : list ) {
      System.out.println( a.getName() );
    }
  }
}
```