Wild Cards and Bounds

Based on the notes from David Fernandez-Baca and Steve Kautz

Bryn Mawr College
CS206 Intro to Data Structures

Two Ways to Compare Objects

Generic sorting methods typically come in one of two forms, which correspond to two ways to compare objects in Java.

- For an existing class Foo, we can create an implementation of Comparator<Foo> and override the compare() method.
  Comparator<? super T>
- If we are starting from scratch to create a class, say T, we can endow T with a "natural" ordering by having T implement the Comparable<T> interface, and then overriding the compareTo() method.
  <T extends Comparable<? super T>>

The Meaning of Comparator<? super T>

Suppose we need a version of selectionSort that works for any type T, by using an external Comparator supplied by the caller.

```
//first attempt, doesn't work.
public static<T> void selectionSort(T[] arr,
                                    Comparator<T> comp ) { … }
```

Suppose you have:

- defined Vehicle class
- created VehicleComparator implementing Comparator<Vehicle> which compares vehicles by price
- created Truck extends Vehicle
- Truck[] arr; VehicleComparator myComp;

T in selectionSort is inferred to be Truck, but myComp implements Comparator<Vehicle>.

```
selectionSort(arr, myComp ); //Incorrect! Why?
```

Solution: Wildcard

- Goal: allow subclasses to use a comparison function that was defined in a superclass.
- Solution: use a wildcard to indicate that selectionSort method can use not just a comparator for type T, but a comparator defined for any supertype of T as well.
- "? super T" means "an unknown supertype of T" (possibly an interface, possibly T itself).
- T is the lower bound for the unknown type.

```
//corrected
public static<T> void selectionSort(T[] arr,
                                    Comparator<? super T> comp ) { … }
```

The Meaning of <T extends Comparable<? super T>>

If a type already implements the Comparable interface, the caller does not have to supply a Comparator. Thus, we would also like a method like this:

```
//first attempt, doesn't work.
public static<T> void selectionSort(T[] arr) { … }
```

Problem: To implement the method, we will have to invoke compareTo() on elements of arr. But we cannot, because the compiler has no way to know that elements of type T implement the Comparable interface.

<T extends Comparable<? super T>>

Let's try to put an **upper bound** on the type parameter to indicate that the type T extends Comparable<T>. Then, within the method, we can invoke compareTo() on elements of type T.

```
//second attempt, doesn't work.
public static<T extends Comparable<T>>
        void selectionSort(T[] arr) { … }
```

Suppose you have:

- declared class Vehicle implements Comparable<Vehicle> {...}
- Created Truck extends Vehicle and a variable arr of Truck

Calling selectionSort(arr) does not compile because Truck does not implement Comparable<Truck>, which is what we required by using the upper bound "T extends Comparable<T>". So, as before, we have use "? super T" to allow for the fact that the compareTo method was defined in a superclass.

<T extends Comparable<? super T>>

```
//success!
public static<T extends Comparable<? super T>>
  void selectionSort(T[] arr) { … }
```

Rule of thumb:

- If T is a type parameter and you write Comparator<T>, there is a good chance you actually want Comparator<? super T>.
- If T is a type parameter and you write "T extends Comparable<T>", there is a good chance you want "T extends Comparable<? super T>".

Raw Types

- Prior to the introduction of generic types, a class such as ArrayList always just stored type Object.

```
ArrayList arr = new ArrayList();
arr.add(new Dog("Rolf"));
arr.add(new Retriever("Clover"));
arr.add(new Volkswagen());
for (int i = 0; i < arr.size(); ++i) {
    Dog d = (Dog) arr.get(i); //need downcast
    d.speak();
}
```

It compiles, but gives you a type error (ClassCastException) at runtime.

Raw Types – cont.

- As of 1.5, ArrayList is a generic class:
  - public class ArrayList<E> extends AbstractList<E>

```
ArrayList<Dog>  arr = new ArrayList<Dog>();
arr.add(new Dog("Rolf"));
arr.add(new Retriever("Clover"));
//arr.add(new Volkswagen()); //error is caught at compile time
for (int i = 0; i < arr.size(); ++i) {
   Dog d = arr.get(i); //no downcast
   d.speak();
}
```

You can still use ArrayList the old way, but it is called a "raw type" and you get compiler warnings.

Erasure

- The Java compiler uses the type arguments to check whether you are doing anything that might cause a type error.
- It then **erases** the type arguments when it produces the class file itself. What actually appears in the class file is the **upper bound** of the type parameter, which defaults to Object, unless specified otherwise.
- The JDK docs call this phenomenon **erasure**; the technical term for this is that generic types in Java are "non-reified".

Erasure Example – Before Erasure

```
public class GenTest {
   public static void main(String[] args) {
      ArrayList<String> arr1 = new ArrayList<String>();
      ArrayList<Pet> arr2 = new ArrayList<Pet>();
      arr1.add("Hello, world!");
      arr2.add(new Dog("Rolf", null));
      String s = arr1.get(0);
      Pet p = arr2.get(0);
      System.out.println(s);
      System.out.println(p.getName());
      System.out.println(arr1.getClass() == arr2.getClass());
   }
}
```

Erasure Example – After Erasure

```
public class GenTest {
   public GenTest() {}
   public static void main(String[] args) {
      ArrayList arr1 = new ArrayList ();
      ArrayList arr2 = new ArrayList ();
      arr1.add("Hello, world!");
      arr2.add(new Dog("Rolf", null));
      String s = (String) arr1.get(0);
      Pet p = (Pet) arr2.get(0);
      System.out.println(s);
      System.out.println(p.getName());
      System.out.println(arr1.getClass() == arr2.getClass());
   }
}
```

Erasure

- Since the actual type information is not present in the compiled class files, you generally cannot do anything with a type parameter that would require the type to be known at runtime.
  - For example, statements such as new T(), new T[], x instanceof T, will not compile.