

The Meaning of

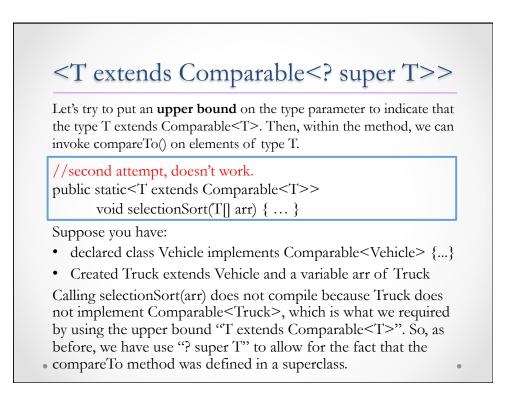
<T extends Comparable<? super T>>

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.

•



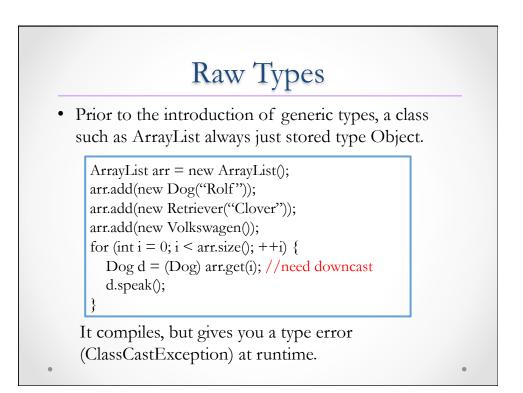
<T extends Comparable<? super T>>

//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>".



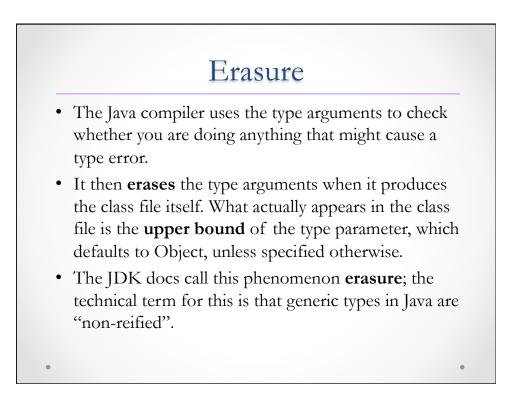
Raw Types - cont.

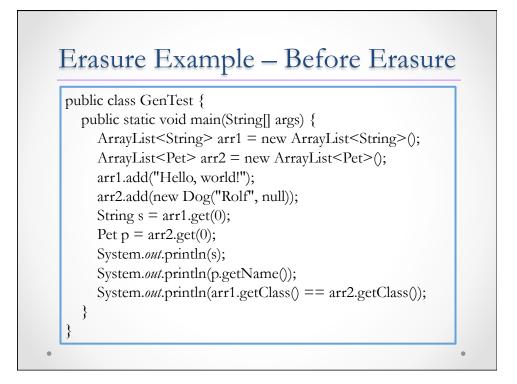
• As of 1.5, ArrayList is a generic class:

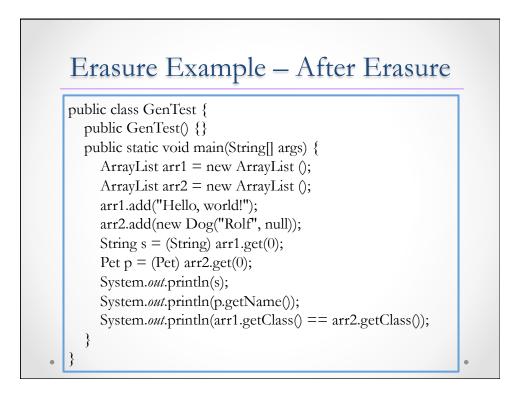
o public class ArrayList<E> extends AbstractList<E>

```
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

- 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 instance of T, will not compile.