

Generics 2

- Recap from Generics 1
- Bounded type parameters
- Wildcards
- Generic Methods

Generics 2

- Recap from Generics 1
 - It ensures that all the members of the collection are objects related by inheritance
 - It allows the data type to be determined later by the client of the class
 - Allows Java compiler to apply strong type checking
 - Improves flexibility and maintainability

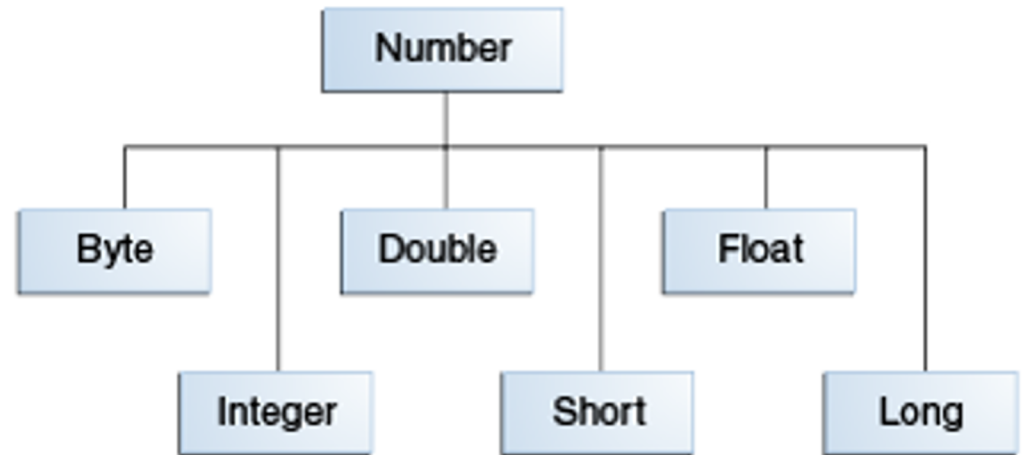


Image source:

Oracle. "The Numbers Classes." The Java™ Tutorials, Oracle, docs.oracle.com/javase/tutorial/java/data/numberclasses.html. Accessed 10 July 2020

Bounded Type Parameters

- Generics offer flexibility, but what about when you want to enforce restrictions on types?

```
public static void add(T first, T second) {  
    System.out.println("Sum = " + first.doubleValue() + second.doubleValue());  
}
```

Declaring Bounded Type Parameters

< typeParameterName extends UpperBound >

Bounded Type Parameters

```
public static <T extends Number> void add(T first, T second) {  
    System.out.println("Sum = " + first.doubleValue() + second.doubleValue());  
}
```

Generic Methods

method/access modifiers <genericParams> returnType methodName (methodParams)

Example Generic method declaration

```
public static <T extends Number> void add ( T first, T second )
```

Wildcards

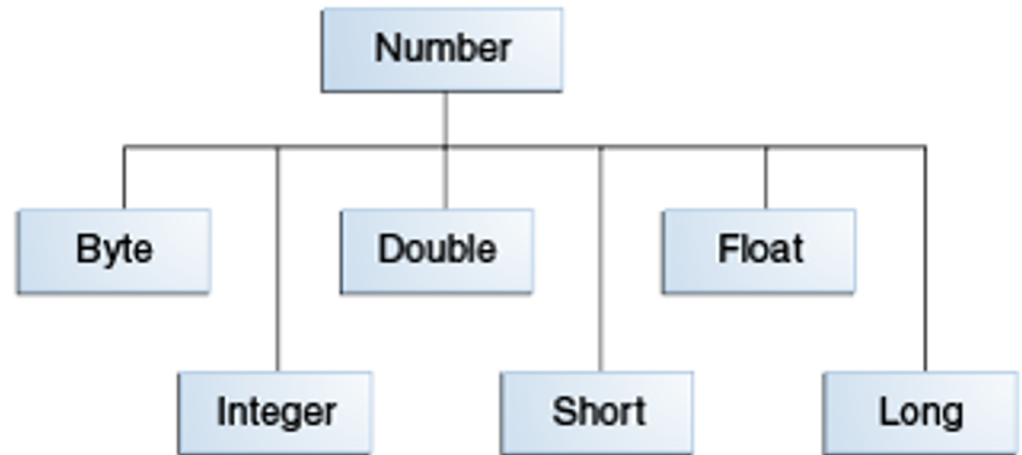
- The question mark (?)
 - The ? can be used as a placeholder to represent any class
- When should you use ?
 - parameter, field, local variable, or even as a return type
- Do not use
 - type argument

Bounded Wildcards

- Recall that ? Is a placeholder representing any class
- Therefore
 - <? super MyClass>
 - ? represents any superclass of MyClass, so MyClass is the lower bound
 - <? extends MyClass>
 - ? represents any subclass of MyClass, so MyClass is the upper bound

Upper Bounded Wildcard

- Consider the effect of the following:
 - ? extends Number



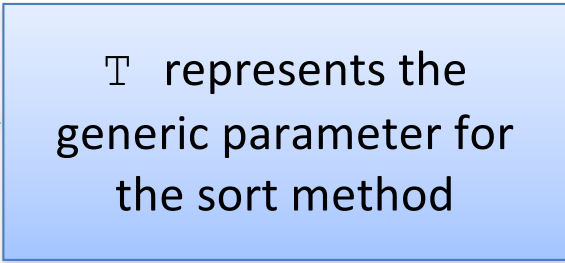
Source:

<https://docs.oracle.com/javase/tutorial/java/data/numberclasses.html>

Declaring a Generic Method

Sample declarations:

```
public static <T> void sort(T[] items, Comparator<? super T> comp)
```



T represents the
generic parameter for
the sort method

Declaring a Generic Method (cont.)

Sample declarations:

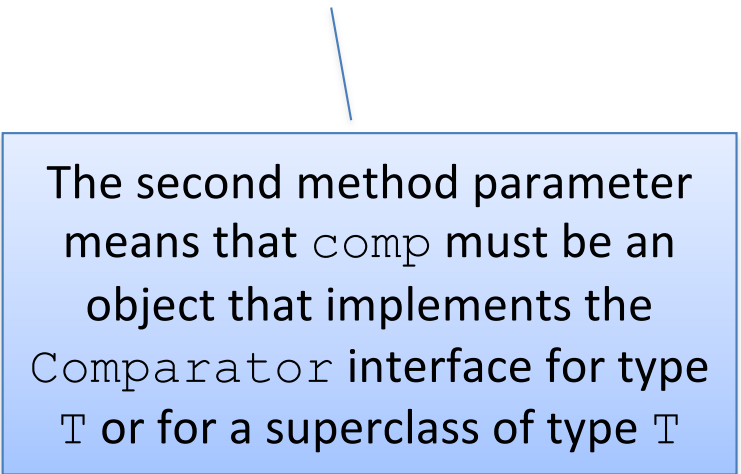
```
public static <T> void sort(T[] items, Comparator<? super T> comp)
```

T should also appear in
the method parameter
list

Declaring a Generic Method (cont.)

Sample declarations:

```
public static <T> void sort(T[] items, Comparator<? super T> comp)
```

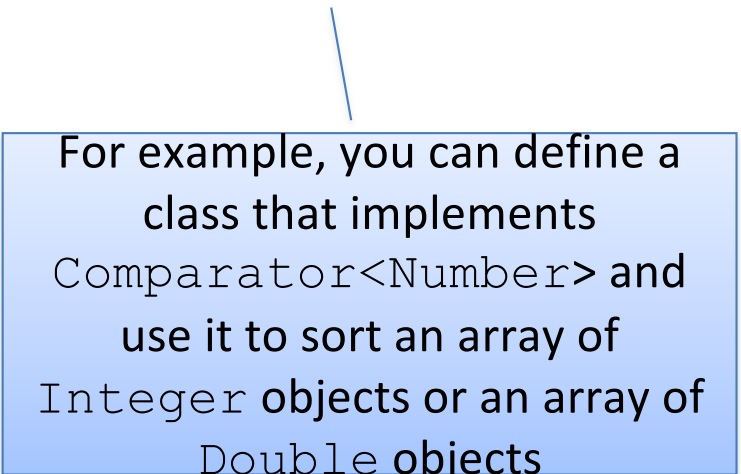


The second method parameter means that `comp` must be an object that implements the `Comparator` interface for type `T` or for a superclass of type `T`

Declaring a Generic Method (cont.)

Sample declarations:

```
public static <T> void sort(T[] items, Comparator<? super T> comp)
```

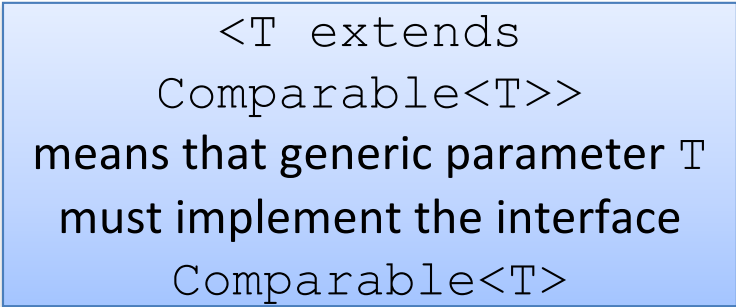


For example, you can define a class that implements `Comparator<Number>` and use it to sort an array of `Integer` objects or an array of `Double` objects

Declaring a Generic Method (cont.)

Sample declarations:

```
public static <T extends Comparable<T>> void sort(List<T> list)
```



<T extends
Comparable<T>>
means that generic parameter T
must implement the interface
Comparable<T>

Declaring a Generic Method (cont.)

Sample declarations:

```
public static <T extends Comparable<T>> void sort(List<T> list)
```

The method parameter `list`
(the object being sorted) is of
type `List<T>`

