You're also free to put boundaries on the type you declare. For example, if you want to restrict the makeArrayList() method to only Number or its subtypes (Integer, Float, and so on), you would say.

public <T extends Number> void makeArrayList(T t)

public void makeList(T t) { }// legal method

public class Radio {

public <T> Radio(T t) { } // legal constructor

}

You can declare a class with a name that is the same as the type parameter placeholder:

class X { public <X> X(X x) { } }

One of the most common mistakes programmers make when creating generic classes or methods is to use a <?> in the wildcard syntax rather than a type variable <T>, <E>, and so on. This code might look right, but isn't:

public class NumberHolder<? extends Number> { }

While the question mark works when declaring a reference for a variable, it does NOT work for generic class and method declarations. This code is not legal:

public class NumberHolder<?> { ? aNum; } // NO!

But if you replace the <?> with a legal identifi er, you're good:

public class NumberHolder<T> { T aNum; } // Yes

* equals(), hashCode(), and toString() are public.
* If you don't override equals(), your objects won't be useful hashing keys.
* If you don't override equals(), different objects can't be considered equal.
* Strings and wrappers override equals() and make good hashing keys.
* When overriding equals(), use the instanceof operator to be sure you're evaluating an appropriate class.
* Highlights of the equals() contract:
* Reflexive: x.equals(x) is true.
* Symmetric: If x.equals(y) is true, then y.equals(x) must be true.
* Transitive: If x.equals(y) is true, and y.equals(z) is true, then z.equals(x) is true.
* Consistent: Multiple calls to x.equals(y) will return the same result.
* Null: If x is not null, then x.equals(null) is false.
* If x.equals(y) is true, then x.hashCode() == y.hashCode() is true.
* HashMap, HashSet, Hashtable, LinkedHashMap, and LinkedHashSet use hashing.
* Highlights of the hashCode() contract:
* Consistent: Multiple calls to x.hashCode() return the same integer.
* If x.equals(y) is true, x.hashCode() == y.hashCode() is true.
* If x.equals(y) is false, then x.hashCode() == y.hashCode() can be either true or false, but false will tend to create better efficiency.
* Transient variables aren't appropriate for equals() and hashCode().
* **ArrayList** Fast iteration and fast random access.
* **Vector** It's like a slower ArrayList, but it has synchronized methods.
* **LinkedList** Good for adding elements to the ends, i.e., stacks and queues.
* **HashSet** Fast access, assures no duplicates, provides no ordering.
* **LinkedHashSet** No duplicates; iterates by insertion order.
* **TreeSet** No duplicates; iterates in sorted order.
* **HashMap** Fastest updates (key/values); allows one null key, many null values.
* **Hashtable** Like a slower HashMap (as with Vector, due to its synchronized methods). No null values or null keys allowed.
* **LinkedHashMap** Faster iterations; iterates by insertion order or last accessed; allows one null key, many null values.
* **TreeMap** A sorted map.
* **PriorityQueue** A to-do list ordered by the elements' priority.
* Sorting can be in natural order or via a Comparable or many Comparators.
* To be searched, an array or List must first be sorted.
* A sort() method. Sort using a Comparator or sort using natural order.
* A binarySearch() method. Search a presorted array or List.
* Arrays.asList() creates a List from an array and links them together.
* Collections.reverse() reverses the order of elements in a List.
* Collections.reverseOrder() returns a Comparator that sorts in reverse.
* Lists and Sets have a toArray() method to create arrays.
* You can pass a generic collection into a method that takes a non-generic collection, but the results may be disastrous. The compiler can't stop the method from inserting the wrong type into the previously type-safe collection.
* Polymorphic assignments apply only to the base type, not the generic type parameter. You can say

List<Animal> aList = new ArrayList<Animal>(); // yes

You can't say

List<Animal> aList = new ArrayList<Dog>(); // no

* When using a wildcard List<? extends Dog>, the collection can be accessed but not modified.
* When using a wildcard List<?>, any generic type can be assigned to the reference, but for access only—no modifications.
* List<Object> refers only to a List<Object>, while List<?> or List<? extends Object> can hold any type of object, but for access only.
* Declaration conventions for generics use T for type and E for element.
* The generics type identifier can be used in class, method, and variable declarations:

class Foo<t> { } // a class

T anInstance; // an instance variable

Foo(T aRef) {} // a constructor argument

void bar(T aRef) {} // a method argument

T baz() {} // a return type

* You can declare a generic method using a type not defined in the class:

public <T> void makeList(T t) { }

This is NOT using T as the return type. This method has a void return type, but to use T within the argument, you must declare the <T>, which happens before the return type.