在定義泛型時，可以定義型態的邊界。例如：

```
class Animal {}
class Human extends Animal {}
class Toy {}
class Duck<T extends Animal> {}

public class BoundDemo {
    public static void main(String[] args) {
        Duck<Animal> ad = new Duck<Animal>();
        Duck<Human> hd = new Duck<Human>();
        Duck<Toy> hd = new Duck<Toy>();  // 編譯錯誤
    }
}
```

在上例中，使用extends限制指定T實際型態時，必須是Animal的子類別，你可以使用Animal與Human來指定T實際型態，但不可以使用Toy，因為Toy不是Animal的子類別。

一個實際應用可以用快速排序法的例子來說明：

```
package cc.openhome;

public class Util {
    public static <T extends Comparable> void sort(T[] array) {
        sort(array, 0, array.length-1);
    }

    private static <T extends Comparable> void sort(T[] array, int left, int right) {
        if(left < right) {
            int q = partition(array, left, right);
            sort(array, left, q-1);
            sort(array, q+1, right);
        }
    }

    private static <T extends Comparable> int partition(T[] array, int left, int right) {
        int i = left - 1;
        for(int j = left; j < right; j++) {
            if(array[j].compareTo(array[right]) <= 0) {
                i++;
                swap(array, i, j);
            }
        }
        swap(array, i+1, right);
        return i + 1;
    }

    private static <T> void swap(T[] array, int i, int j) {
        T t = array[i];
        array[i] = array[j];
        array[j] = t;
    }
}
```

關於快速排序法，可參考 常見程式演算 中的說明。

物件要能排序，基本上物件本身必須能比較大小，因此這個範例要求sort()方法傳入的陣列，當中每個元素必須是T型態，<T extends Comparable>語法限制了T必須實作java.lang.Comparable介面。可以如下使用sort()方法：

```
String[] words = {"B", "X", "A", "M", "F", "W", "O"};
Util.sort(words);
```

由於String實作了Comparable介面，因此可以使用Util的sort()方法進行排序。若extends之後指定了類別或介面後，想再指定其它介面，可以使用&連接。例如：

```
public class Some<T extends Iterable<T> & Comparable<T>> {
    ...
}
```

接著要來看看在泛型中的型態通配字元?。如果你定義了以下類別：

```
package cc.openhome;

public class Node<T> {
    public T value;
    public Node<T> next;

    public Node(T value, Node<T> next) {
        this.value = value;
        this.next = next;
    }
}
```

如果有個Fruit類別繼承體系如下：

```
package cc.openhome;

class Fruit {
    int price;
    int weight;
    Fruit() {}
    Fruit(int price, int weight) {
        this.price = price;
        this.weight = weight;
    }
}
class Apple extends Fruit {
    Apple() {}
    Apple(int price, int weight) {
         super(price, weight);
    }
    @Override
    public String toString() {
```

```
        return "Apple";
    }
}

class Banana extends Fruit {
    Banana() {}
    Banana(int price, int weight) {
         super(price, weight);
    }
    @Override
    public String toString() {
        return "Banana";
    }
}
```

如果有以下程式片段，則會發生編譯錯誤：

```
Node<Apple> apple = new Node<>(new Apple(), null);
Node<Fruit> fruit = apple;  // 編譯錯誤，incompatible types
```

在這個片段中，apple型態宣告為Node<Apple>，而fruit型態宣告為Node<Fruit>，那麼Node<Apple>是一種Node<Fruit>嗎？顯然地，編譯器認為不是，所以不允許通過編譯。

如果B是A的子類別，而Node<B>可視為一種Node<A>，則稱Node具有共變性（Covariance）或有彈性的（flexible）。從以上編譯結果可看出，Java的泛型並不具有共變性，不過可以使用型態通配字元?與extends來宣告變數，使其達到類似共變性。例如以下可以通過編譯：

```
Node<Apple> apple = new Node<>(new Apple(), null);
Node<? extends Fruit> fruit = apple; // 類似共變性效果
```

在上面片段中使用了<? extends Fruit>語法，?代表fruit參考的Node物件，不知道T實際宣告為何種型態，加上extends Fruit表示雖然不知道T宣告為何種型態，但一定宣告會為Fruit的子類別型態。由於apple被宣告為Node<Apple>，Apple是一種Fruit，所以可以通過編譯。

一個實際應用的例子是：

```
package cc.openhome;

public class CovarianceDemo {
    public static void main(String[] args) {
        Node<Apple> apple1 = new Node<>(new Apple(), null);
        Node<Apple> apple2 = new Node<>(new Apple(), apple1);
        Node<Apple> apple3 = new Node<>(new Apple(), apple2);

        Node<Banana> banana1 = new Node<>(new Banana(), null);
        Node<Banana> banana2 = new Node<>(new Banana(), banana1);

        printlnForEach(apple3);
        printlnForEach(banana2);
    }

    static void printlnForEach(Node<? extends Fruit> n) {
        Node<? extends Fruit> node = n;
        do {
            System.out.println(node.value);
            node = node.next;
        } while(node != null);
    }
}
```

printlnForEach()方法目的是可以顯示所有的水果節點，如果參數n僅宣告為Node<Fruit>型態，將只能接受Node<Fruit>實例。由於printlnForEach()方法使用型態通配字元?與extends宣告參數，使得n具備類似共變性的效果，因此show()方法就可以接受Node<Apple>實例，也可以接受Node<Banana>實例。執行結果如下：

```
Apple
Apple
Apple
Banana
Banana
```

若宣告?不搭配extends，則預設為? extends Object。例如：

```
Node<?> node = null; // 相當於Node<? extends Object>
```

以上的node可接受Node<Object>、Node<Fruit>、Node<Apple>等物件，也就是只要角括號中的物件是一種Object，都可以通過編譯。

注意！這與宣告為Node<Object>不同，如果node宣告為Node<Object>，那就真的只能參考至Node<Object>實例了，也就是以下會編譯錯誤：

```
Node<Object> node = new Node<Integer>(1, null);
```

但以下會編譯成功：

```
Node<?> node = new Node<Integer>(1, null);
```

一旦使用通配字元?與extends限制T的型態，就只能透過T宣告的名稱取得物件指定給Object，或將T宣告的名稱指定為null，除此之外不能進行其它指定動作。例如：

```
Node<? extends Fruit> node = new Node<>(new Apple(), null);
Object o = node.value;
node.value = null;
Apple apple = node.value;   // 編譯錯誤
node.value = new Apple();   // 編譯錯誤
```

以上程式片段，只知道value參考的物件型態會是繼承Fruit，但實際上會是Apple還是Banana呢？如果實際上node.value是Banana實例，那指定給Apple型態的apple當然不對，所以編譯錯誤。如果一開始建立Node時指定的T型態是Banana，那將Apple實例指定給node.value就不符合原先要求，所以編譯也是錯誤。

因為Java的泛型語法只用在編譯時期檢查，也就是泛型上的型態資訊僅提供編譯器進行型態檢查，不做為執行時期的確切型態資訊（又稱為型態抹除），由於無法在執行時期獲得確切型態資訊，編譯器就只能就編譯時期看到的型態來作檢查，因而造成以上談及的限制。

Java泛型在執行時期型態抹除，有時會讓人感到困惑。例如以下執行結果會是true或false呢？

```
List<Integer> list1 = new ArrayList<>();
List<String> list2 = new ArrayList<>();
System.out.println(list1.equals(list2));
```