Generic Types

saacsos

Non-Generic Class

```
class Box {
   private Object object;
   public void set(Object object) {
      this.object = object;
   }
   public Object get() { return object; }
}
```

Generic Types

A generic class or interface that is parameterized over types

Generic Class

```
class Box<T> {
    private T t;
    public void set(T t) {
        this.t = t;
    }
    public T get() { return t; }
}
```

Can be any non-primitive type

Type Parameter Naming Conventions

- E - Element (used by the Java Collection Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S, U, V, etc. - 2nd, 3rd, 4th types

Invoking and Instantiating a Generic Type

```java
Box<Integer> intBox = new Box<Integer>();

Box<Weapon> weapon = new Box<Weapon>();
```

An invocation of a generic type is generally known as a parameterized type.

The Diamond

You can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context

```
Box<Integer> intBox = new Box<>();
Box<Weapon> weapon = new Box<>();
```

Multiple Type Parameters

```
public interface Pair<K, V> {

    public K getKey();

    public V getValue();

}
```

Multiple Type Parameters

```
public class OrderedPair<K, V>
             implements Pair<K, V> {
   private K key;
   private V value;
   public OrderedPair(K key, V value) {
      this.key = key;
      this.value = value;
   }
   public K getKey() { return key; }
   public V getValue() { return value; }
}
```

Multiple Type Parameters

```
Pair<String, Integer> p1 =
                new OrderedPair<>("Even", 8);

Pair<String, String>  p2 =
     new OrderedPair<>("hello", "world");
```

Parameterized Type

```
OrderedPair<Integer, Box<Student>> p3
   = new OrderedPair<>(10,
        new Box<>(new Student("Anna")));

System.out.println(p3.getKey());
System.out.println(p3.getValue().get());
```

Raw Types

If the actual type argument is omitted,

you create a raw type of Box<T>

```
Box rawBox = new Box();
```

Raw Types

If the actual type argument is omitted,

you create a raw type of Box<T>

```
Box rawBox = new Box();
```

Raw Types

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;      // OK
rawBox.set(8);
 // warning: unchecked invocation to set(T)


Box rawBox2 = new Box();
Box<Integer> intBox = rawBox2;
     // warning: unchecked conversion
```

Generic Methods

```
public class Util {
    public static <K, V> boolean equals(
                Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey())
            && p1.getValue().equals(p2.getValue());
    }
}
```

Generic Methods

```
Pair<Integer, String> p1
             = new OrderedPair<>(1, "apple");
Pair<Integer, String> p2
             = new OrderedPair<>(2, "pear");
boolean same
       = Util.<Integer, String>equals(p1, p2);
System.out.println(same);
```

Bounded Type Parameters

```
class Box<T> {
    // ...
    public <U extends Number> void inspect(U u){
      System.out.println(
              "T: " + t.getClass().getName());
      System.out.println(
              "U: " + u.getClass().getName());
     }
}
```

Bounded Type Parameters

```
Box<Student> studentBox
          = new Box<>(new Student("ABC"));
studentBox.inspect(8.5);
studentBox.inspect("Text"); // error
```

Bounded Type Parameters

```
public class GradeBook<T extends Student> {
    private T t;

    public GradeBook(T t)  { this.t = t; }

    public boolean isPassExam() {
        return t.getScore() >= 50;
    }
    // ...
}
```

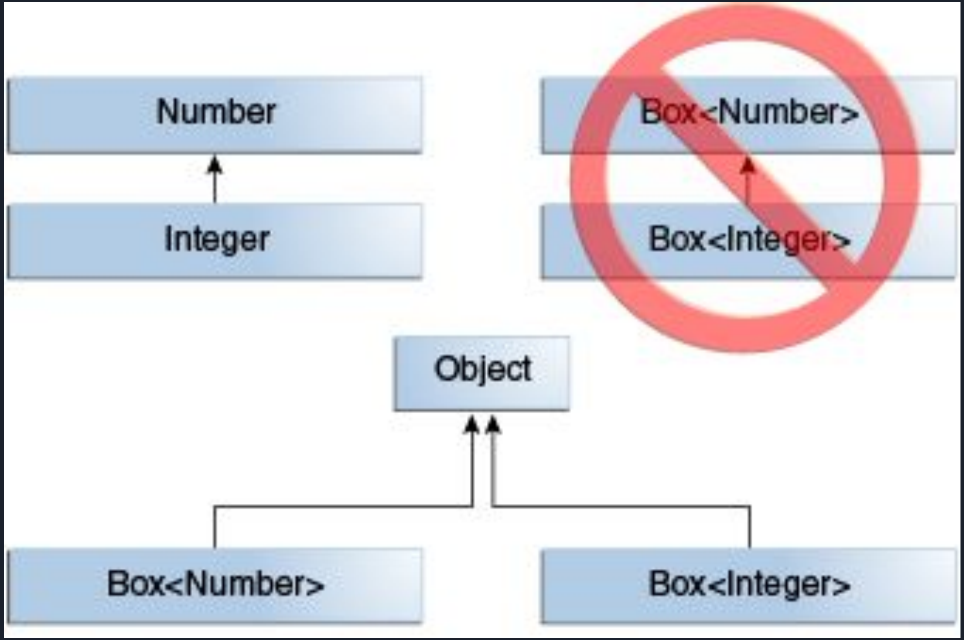
Bounded Type Parameters

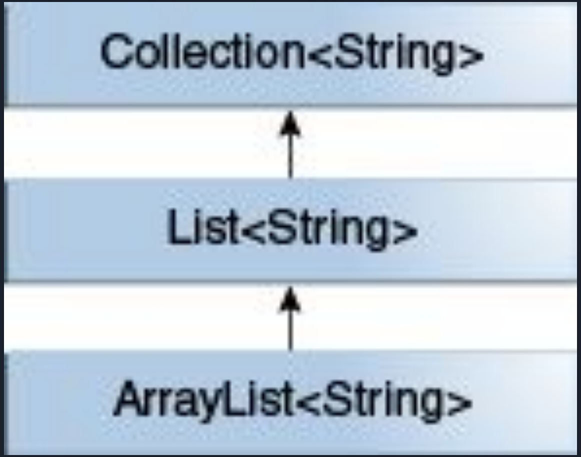
```
Class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }
// If bound A is not specified first,
// you get a compile-time error

class E <T extends B & A & C> { /* ... */ }
// compile-time error
```

Generic Methods and Bounded Type Parameters

```
public static <T extends Comparable<T>>
int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

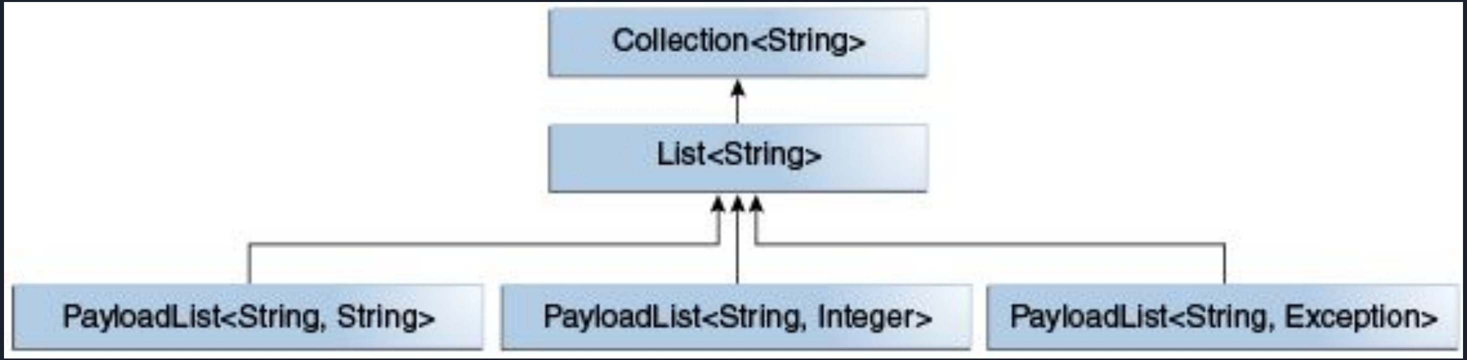
Generics, Inheritance, and Subtypes





Generics, Inheritance, and Subtypes

```
interface PayloadList<E, P> extends List<E> {
  void setPayload(int index, P val);
  ...
}
```


Collection<String>
List<String>
PayloadList<String, String>
PayloadList<String, Integer>
PayloadList<String, Exception>

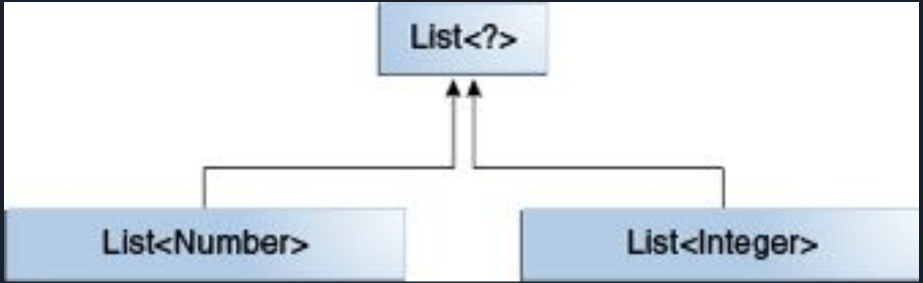
Upper Bounded Wildcards

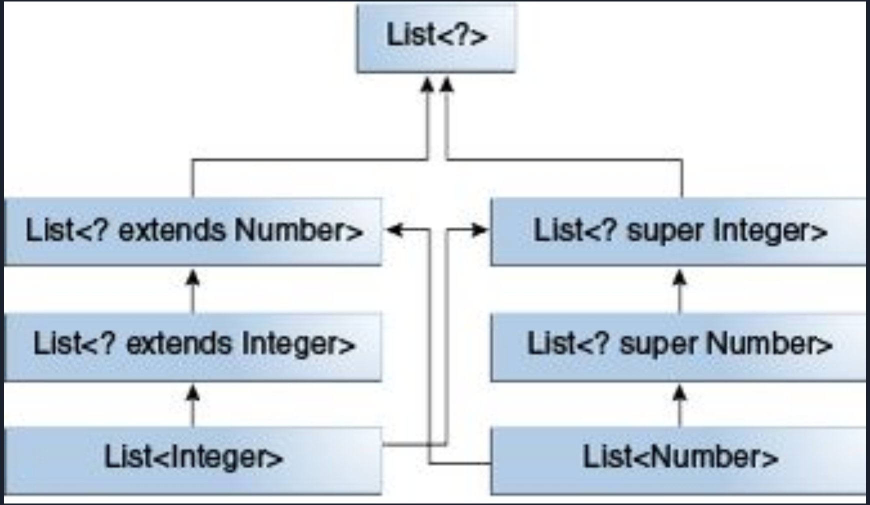
```
public static double
    sumOfList(List<? extends Number> list) {
      double s = 0.0;
      for (Number n : list)
          s += n.doubleValue();
      return s;
}
```

Upper Bounded Wildcards

```
List<Integer> li = Arrays.asList(1, 2, 3);
System.out.println("sum = " + sumOfList(li));

List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);
System.out.println("sum = " + sumOfList(ld));
```

Wildcards and Subtyping





หาค่าที่มากที่สุด โดยใช้ Comparable

```
class Data {
  public static Comparable max(Comparable a, Comparable b) {
     if (a.compareTo(b) > 0) return a;
     return b;
  }

  public static Comparable max(Comparable[] comparables) {
     if (comparables.length == 0) return null;
     Comparable m = comparables[0];
     for (int i = 1; i < comparables.length; i++ )
         m = max(m, comparables[i]);
     return m;
  }
}
```

หาค่าที่มากที่สุด

```
GameCharacter[] players = new GameCharacter[3];
players[0] = new GameCharacter("Player 1", 10);
players[1] = new GameCharacter("Player 2", 20);
players[2] = new GameCharacter("Player 3", 5);

System.out.println(Data.max(players));
```

หาค่าที่มากที่สุด โดยใช้ Comparator

```
class Data {
  public static Object max(Object a, Object b, Comparator c) {
     if (c.compare(a, b) > 0) return a;
     return b;
  }

  public static Object max(Object[] objects, Comparator c) {
     if (objects.length == 0) return null;
     Object m = objects[0];
     for (int i = 1; i < objects.length; i++ )
         m = max(m, objects[i], c);
     return m;
  }
}
```

หาค่า HP ที่มากที่สุด โดยใช้ Comparator

```
GameCharacter[] players = new GameCharacter[3];
players[0] = new GameCharacter("Player 1", 10, 30);
players[1] = new GameCharacter("Player 2", 20, 10);
players[2] = new GameCharacter("Player 3", 5, 25);

System.out.println(Data.max(players,  new Comparator() {
   @Override
   public int compare(Object o1, Object o2) {
       GameCharacter p1 = (GameCharacter) o1;
       GameCharacter p2 = (GameCharacter) o2;
       if (p1.getHp() > p2.getHp()) return 1;
       if (p1.getHp() < p2.getHp()) return -1;
       return 0;
   }
}));
```

หาค่าที่มากที่สุด โดยใช้ Generic และ Comparator

```
public class Data<T> {
    public static <T> T max(T a, T b, Comparator<T> c) {
        if (c.compare(a, b) > 0) return a;
        return b;
    }
    public static <T> T max(List<T> elements, Comparator<T> c) {
        if (elements.size() == 0) return null;
        ListIterator<T> iterator = elements.listIterator();
        T m = iterator.next();
        while (iterator.hasNext()) {
            T t = iterator.next();
            m = max(m, t, c);
        }
        return m;
    }
}
```