Java Generics

- Added in Java in 2004 (1.5)
- Created to allow "a type or method to operate on objects of various types while providing compile-time type safety"
- Generics are used in:
  - Classes
  - Interfaces
  - Methods
  - and Constructors

Why Generics?

- For many and important reasons!
- Including an example:

```
List list = new ArrayList();
list.add(new Integer(2));
list.add("a String");
//then we need casting
Integer integer = (Integer) list.get(0);
String string   = (String) list.get(1);
```

Previous example using generics

```
List<String> strings = new ArrayList<String>();
strings.add("a String");
String aString = strings.get(0);
```

Generics Main Advantages

- Type-safety: We can hold only a single type of objects in generics. It doesn’t allow to store other objects
- Type casting is not required: There is no need to typecast the object
- Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. Good programming strategy suggests it is far better to handle the problem at compile time than runtime

Generics and interfaces

- In contrast to regular interfaces, to define a generic interface it is sufficient to provide the type (or types) it should be parameterized with:

```
public interface GenericInterfaceOneType< T > {
   void performAction( final T action );
}
```

- The interface may be parameterized with more than one type:

```
public interface GenericInterfaceSeveralTypes< T, R > {
   R performAction( final T action );
}
```

Implementing a generic Interface

- Whenever any class wants to implement the interface, it has an option to provide the exact type substitutions:

```
public class ClassImplementingGenericInterface
    implements GenericInterfaceOneType< String > {
  @Override
  public void performAction( final String action ) {
    // Implementation here
  }
}
```

Java Generic Methods

- Generic methods enable programmers to specify, with a single method declaration, a set of related methods
- Rules:
  - Generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type
  - Each type parameter section contains one or more type parameters separated by commas
  - Type parameters can be used to declare the return type and also as placeholders for the types of the arguments that are passed to the generic method
  - Type parameters can represent <u>only reference types</u>, not primitive types

Example 1/2 Generic Method

```
public static <T> void printArray(T[] array) {
        System.out.println("Array with elements:");
        for(T element : array) {
            System.out.println(element);
        }
    }
```

Using the generic method

```
Integer[] intarray = {4,6,1,23,67,100};
printArray(intarray);
String[] stringarray = {"Helen","Paul","John","Peter"};
printArray(stringarray);
```

Example 2/2 Generic Method

```
public static <T> ArrayList<T> makeList(T[] arr){
        ArrayList<T> newlist = new ArrayList<T>();
        for(T t:arr){
            newlist.add(t);
        }
        return newlist;
    }
```

Using the generic method

```
Integer[] intarray = {4,6,1,23,67,100};
ArrayList<Integer> intarraylist = makeList(intarray);
intarraylist.add(15);
```

Important notes!

- If methods are declared (or defined) as part of generic interface or class, they may (or may not) use the generic types of their owner
- They may define own generic types or mix them with the ones from their class or interface declaration:

```
public class GenericMethods< T > {
    public< R > R performAction( final T action ) {
        final R result = ...;
        // Implementation here
        return result;
    }
}
```

- Class constructors are also considered to be kind of initialization methods, and as such, may use the generic types declared by their class, declare own generic types or just mix both

Commonly used type parameter names

- E – Element (used extensively by the Java Collections Framework, for example ArrayList, Set etc.)
- K – Key (Used in Map)
- N – Number
- T – Type
- V – Value (Used in Map)

Java Generic Classes

- The class name is followed by a type parameter section
- The type parameter section of a generic class can have one or more type parameters separated by commas
- Type parameters can be used everywhere inside the class
- Type parameters can represent <u>only reference types</u>, not primitive types

Example Generic Class

```
public class GenClass<T> {
    private T val;
    public T get(){ return val;}
    public void set (T t){
        if (t.toString().length()==5)
        val = t;
    }
    public void myprint (){
        if (val!=null){
            System.out.println("Val has value: "+val.toString());
        }else {
            System.out.println("Val has no value...");
        }
    }
}
```

Using the Generic Class

```
GenClass1<Integer> genc1 = new GenClass1<Integer>();
genc1.set(12345);
genc1.myprint();
```

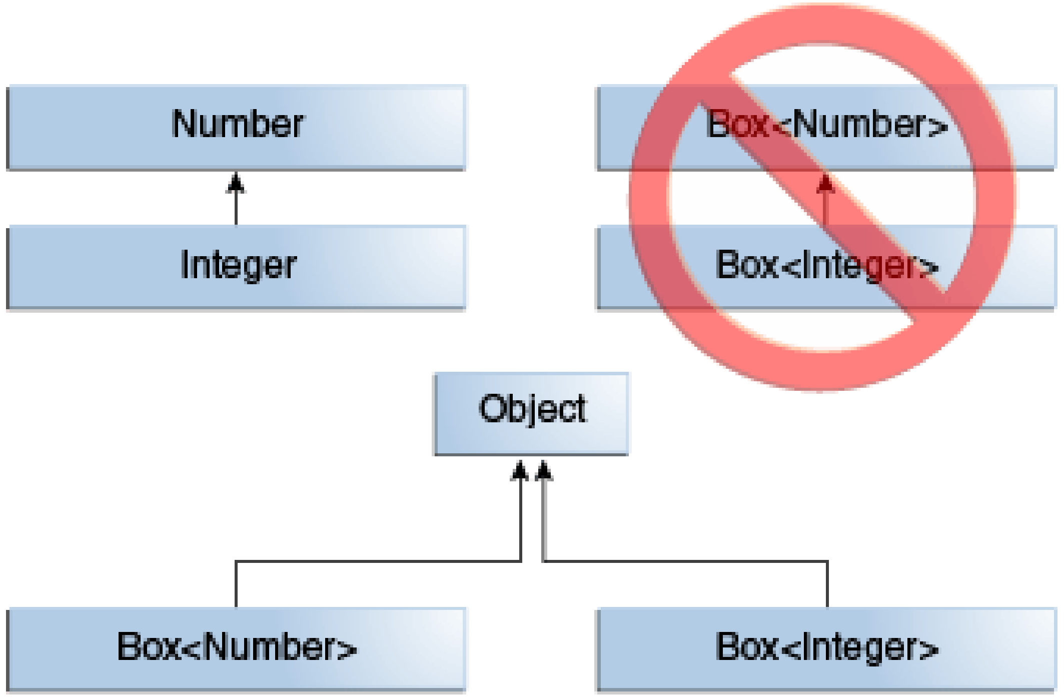
Interesting notes!

- A class may pass (or may not) its generic type (or types) down to the interfaces and parent classes, without providing the exact type instance:

```
public class GenericClassImplementingGenericInterface< T >
        implements GenericInterfaceOneType< T > {
    @Override
    public void performAction( final T action ) {
        // Implementation here
    }
}
```

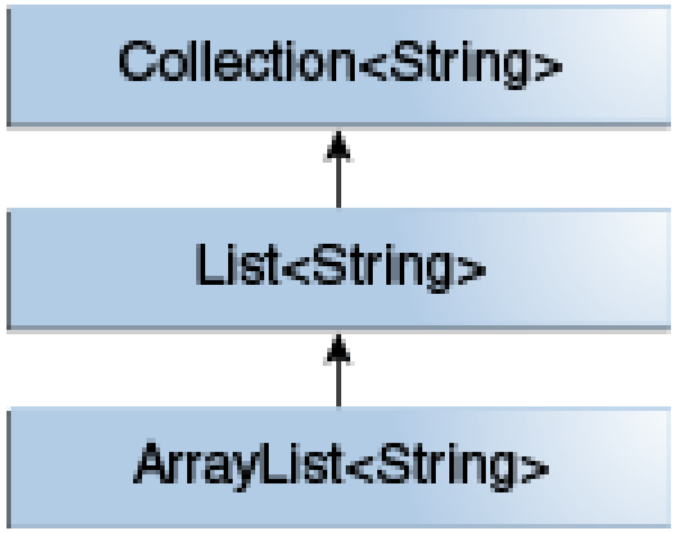
Advanced

Generics, Inheritance, and Subtypes



Advanced

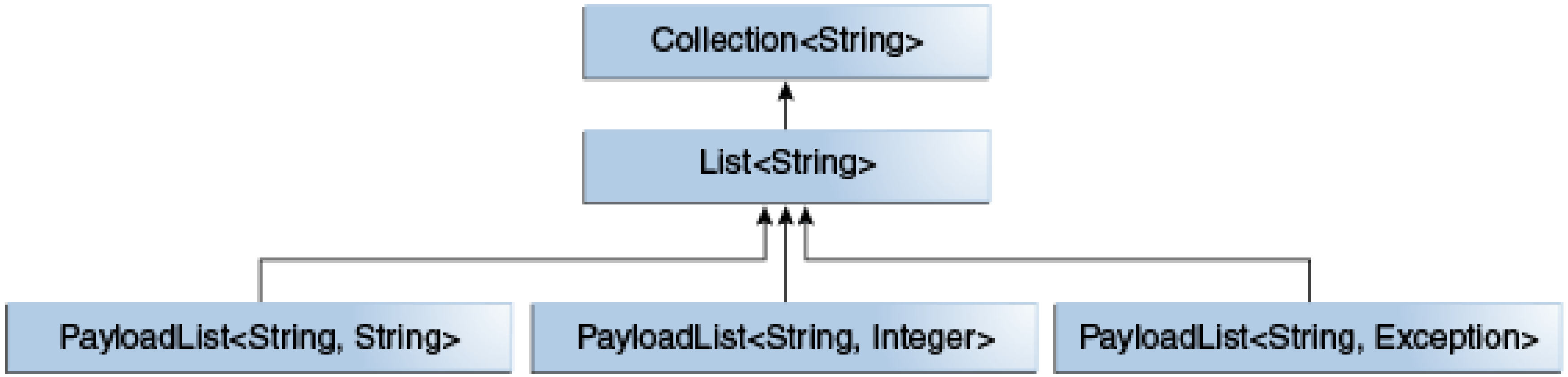
Correct subtyping



Advanced

Example

```
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>

Advanced

Bounded Type Parameters

- There may be times when you want to restrict the types that can be used as type arguments in a parameterized type.
- For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.
- To declare a bounded type parameter, list the type parameter's name, followed by the **extends** keyword, followed by its upper bound.
- Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).
- E.g. <U extends Number>

Advanced

Example

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem)  // compiler error
            ++count;
    return count;
}
```

```
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;
}
```

Advanced

Type Inference

- Type inference represents the Java compiler's ability to look at a method invocation and its corresponding declaration to check and determine the type argument(s).
- The inference algorithm checks the types of the arguments and, if available, assigned type is returned.
- The inference algorithm tries to find a specific type which can fulfill all type parameters
- Box<Integer> integerBox = new Box<>();

Advanced

Multiple Bounds

- A type variable with multiple bounds is a subtype of all the types listed in the bound
- If one of the bounds is a class, it must be specified first
- <T extends B1 & B2 & B3>

Advanced

Upper Bounded Wildcards

- You can use an upper bounded wildcard to relax the restrictions on a variable.
- For example, say you want to write a method that works on List<Integer>, List<Double>, and List<Number>; you can achieve this by using an upper bounded wildcard.
- To declare an upper-bounded wildcard, use the wildcard character ('?'), followed by the extends keyword, followed by its upper bound.
- To write the method that works on lists of Number and the subtypes of Number, such as Integer, Double, and Float, you would specify

  List<? extends Number>

Advanced

Lower Bounded Wildcards

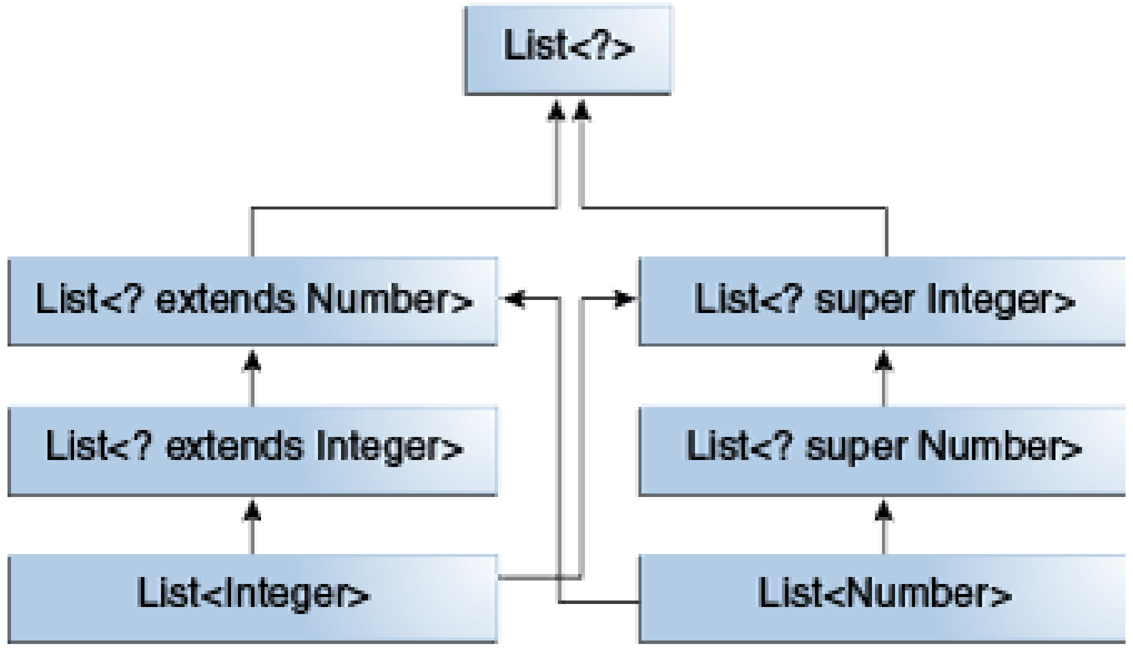
- In a similar way, a lower bounded wildcard restricts the unknown type to be a specific type or a super type of that type.
- A lower bounded wildcard is expressed using the wildcard character ('?'), following by the super keyword, followed by its lower bound:

  <? super A>

- <u>You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both</u>

Advanced

Wildcards and Subtyping



Advanced