EECS 2030 Advanced Object-Oriented Programming

S2023, Section A

Generic Types

So far…

- How to store and organize data?
  - Primitive types
  - Reference types
    - Composition, Aggregation
    - Inheritance
  - Enumerated types
- Do we need anything else?

Implementing stack using composition

```
/**
 * Initializes an empty stack.
 */
public BetterStack() {
   this.elems = new ArrayList<Integer>();
}
```

Implementing stack using composition

```
    /**
     * Pushes a value onto the top of the stack.
     *
     * @param value the value to push onto the stack
     */
    public void push(int value) {
        this.elems.add(value);
    }

    /**
     * Pops a value from the top of the stack.
     *
     * @return the value that was popped from the stack
     */
    public int pop() {
        int last = this.elems.remove(this.elems.size() - 1);
        return last;
    }
}
```

Implementation with Array

- the **`ArrayList`** version of stack hints at how to implement a stack using a plain array
  - however, an array always holds a fixed number of elements
    - you cannot add to the end of the array without creating a new array
    - you cannot reduce the size of the array without creating a new array
- instead of adding and removing from the end of the array, we need to keep track of which element of the array represents the current top of the stack
  - we need a field for this index

```java
import java.util.Arrays;
import java.util.EmptyStackException;

public class IntStack {
  // the initial capacity of the stack
  private static final int DEFAULT_CAPACITY = 16;

  // the array that stores the stack
  private int[] stack;

  // the index of the top of the stack (equal to -1 for an empty stack)
  private int topIndex;
```

```java
/**
 * Create an empty stack.
 */
public IntStack() {
  this.stack = new int[IntStack.DEFAULT_CAPACITY];
  this.topIndex = -1;
}
```

Implementation with Array

```
IntStack t = new IntStack();
```

this.topIndex == -1

this.stack	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Implementation with Array

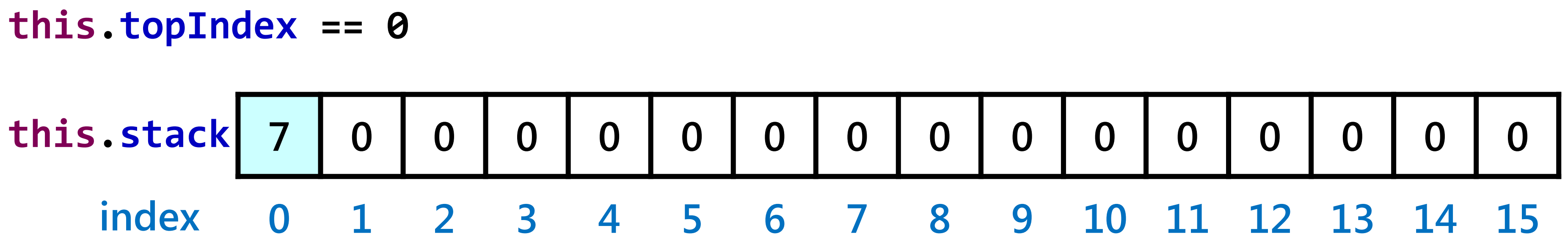
pushing a value onto the stack:

- increment `this.topIndex`
- set the value at `this.stack[this.topIndex]`

Implementation with Array

```
IntStack t = new IntStack();

t.push(7);
```



Implementation with Array

```
IntStack t = new IntStack();

t.push(7);

t.push(-5);
```

this.topIndex == 1

this.stack	7	-5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Implementation with Array

popping a value from the stack:

- get the value at `this.stack[this.topIndex]`
- decrement `this.topIndex`
- return the value

notice that we do not need to modify the value stored in the array

Implementation with Array

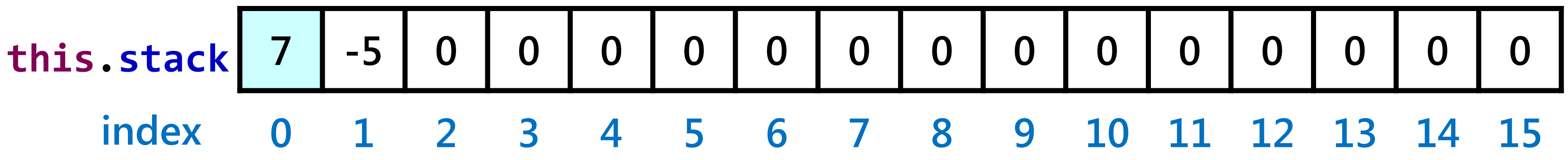
```
IntStack t = new IntStack();

t.push(7);

t.push(-5);

int value = t.pop();    // value == -5
```

```
this.topIndex == 0
```



```
  /**
   * Pop and return the top element of the stack.
   *
   * @return the top element of the stack
   * @throws EmptyStackException if the stack is empty
   */
  public int pop() {
    // is the stack empty?
    if (this.topIndex == -1) {
      throw new EmptyStackException();
    }
    // get the element at the top of the stack
    int element = this.stack[this.topIndex];

    // adjust the top of stack index
    this.topIndex--;

    // return the element that was on the top of the stack
    return element;
}
```

Implementation with Array

// stack state when we can safely do one more push

```
this.topIndex == 14
```

`this.stack`	7	-5	6	3	2	1	0	0	9	-3	2	7	1	-2	1	0
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

```
/**
 * Push an element onto the top of the stack.
 *
 * @param element the element to push onto the stack
 */
public void push(int element) {
  // is there capacity for one more element?
  if (this.topIndex < this.stack.length - 1) {
    // increment the top of stack index and insert the element
    this.topIndex++;
    this.stack[this.topIndex] = element;
  }
  else {
```

Adding Capacity

if we run out of capacity in the current array we need to add capacity by doing the following:

- make a new array with greater capacity
  - how much more capacity?
- copy the old array into the new array
- set `this.stack` to refer to the new array
- push the element onto the stack

```
    else {
      // make a new array with double previous capacity
      int[] newStack = new int[this.stack.length * 2];

      // copy the old array into the new array
      for (int i = 0; i < this.stack.length; i++) {
        newStack[i] = this.stack[i];
      }

      // refer to the new array and push the element onto the stack
      this.stack = newStack;
      this.push(element);
    }
  }
```

Adding Capacity

When working with arrays, it is a common operation to have to create a new larger array when you run out of capacity in the existing array

Probably better to use **Arrays.*copyOf*** to create and copy an existing array into a new array

```
  else {
    // make a new array with greater capacity
    int[] newStack = new int[this.stack.length * 2];

    int[] newStack = Arrays.copyOf(this.stack, this.stack.length * 2);
    // copy the old array into the new array
    for (int i = 0; i < this.stack.length; i++) {
      newStack[i] = this.stack[i];
    }

    // refer to the new array and push the element onto the stack
    this.stack = newStack;
    this.push(element);
  }
}
```

A Stack of String

suppose we wanted a stack of strings

before the introduction of generics into the Java language one solution would have been to create a new class

`StringStack`

```java
import java.util.Arrays;
import java.util.EmptyStackException;

public class StringStack {
  // the initial capacity of the stack
  private static final int DEFAULT_CAPACITY = 16;

  // the array that stores the stack
  private String[] stack;

  // the index of the top of the stack (equal to -1 for an empty stack)
  private int topIndex;

  public void push(String element) {  // implementation not shown  }

  public String pop() {  // implementation not shown }

}
```

A Stack of String

almost *nothing* changes in the implementation of **`StringStack`** compared to **`IntStack`**

consider the **pop** method

```java
  /**
   * Pop and return the top element of the stack.
   *
   * @return the top element of the stack
   * @throws EmptyStackException if the stack is empty
   */
  public String pop() {
    // is the stack empty?
    if (this.topIndex == -1) {
      throw new EmptyStackException();
    }
    // get the element at the top of the stack
    String element = this.stack[this.topIndex];

    // adjust the top of stack index
    this.topIndex--;

    // return the element that was on the top of the stack
    return element;
}
```

A Stack of String

the ***only*** things that change are related to **type declarations**

- we substitute **`String`** for **`int`** in a few places

Could the compiler perform this substitution for us?

- this is exactly what generics in Java are for

Java Generics

generics in Java allow us to parameterize a class, interface, or method over types

instead of creating separate classes **`IntStack`**, **`StringStack`**, **`DateStack`**, etc. we create a single class and specify the type parameter:

- **`Stack<Integer>`** instead of **`IntStack`**
- **`Stack<String>`** instead of **`StringStack`**
- **`Stack<Date>`** instead of **`DateStack`**

Declaring a Generic Class

declare a generic class using the following syntax:

```
    class ClassName<T1, T2, ..., Tn>
```

the angled brackets denote the type parameter section that specifies the *type parameters* (also called *type variables*)

the type parameters can be any non-primitive type

Type Parameter Naming Conventions

type parameters are single, uppercase letters

common type parameter names:

- **E** – Element
- **K** – Key
- **N** – Number
- **T** – Type
- **V** – Value

A Generic Stack Class

```
import java.util.Arrays;
import java.util.EmptyStackException;

public class Stack<E> {
  // the initial capacity of the stack
  private static final int DEFAULT_CAPACITY = 16;

  // the array that stores the stack
  private E[] stack;

  // the index of the top of the stack
  private int topIndex;
```

```
/**
 * Create an empty stack.
 */
public Stack() {
  this.stack = new E[Stack.DEFAULT_CAPACITY];
  this.topIndex = -1;
}
```

No Arrays of Generic Type

the highlighted line on the previous slide causes a compilation error

- in Java you cannot create an array of generic type

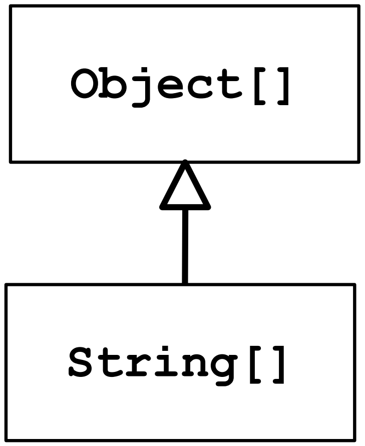
a solution is to make an array of **`Object`** and cast the array to the generic array type

```
/**
 * Create an empty stack.
 */
public Stack() {
  this.stack = (E[]) new Object[Stack.DEFAULT_CAPACITY];
  this.topIndex = -1;
}
```

Why?
Because Arrays are Covariant

arrays in Java are said to be *covariant*

- if **Sub** is a subtype of **Super** then the array type **Sub[]** is a subtype of the array type **Super[]**

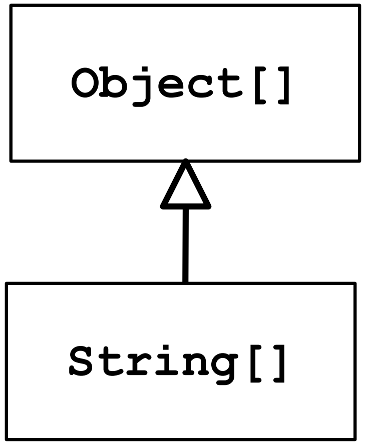


Why are Arrays Covariant?

So that functions like

```
void methodForArray(Object[] a);
```

can work with any array subtypes (we want polymorphism, right)



Simpler at the time

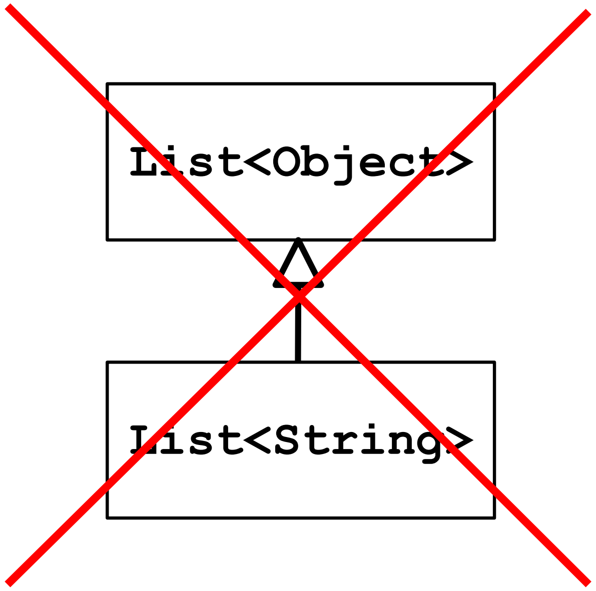
Drawback: array type must be known at compile time

```
this.stack = new E[Stack.DEFAULT_CAPACITY];
```

Generics are not Covariant

generics are *invariant*

for any two distinct types **T1** and **T2**, **List<T1>** is neither a subtype nor a supertype of **List<T2>**



Some illegal code:

```
List<Apple> apples = new List<Apple>();
List<Fruit> fruits = apples; // this should not be possible
fruits.add(new Orange());
Apple apple = apples.get(0); // list of apples now
                             // contains an orange
```

Q. But we want polymorphism, right?

A. *Wildcards* can deal with that if needed

```
void methodForList(List<?> list);
```

Back to Generic Stack Class

the methods **push** and **pop** can be implemented with only very minor changes compared to **`IntStack`**

```
public void push(E element) {
  // is there capacity for one more element?
  if (this.topIndex < this.stack.length - 1) {
    // increment the top of stack index and insert the element
    this.topIndex++;
    this.stack[this.topIndex] = element;
  } else {
    E[] newStack = Arrays.copyOf(this.stack,
                                 this.stack.length * 2);
    // refer to the new array and push the element onto the stack
    this.stack = newStack;
    this.push(element);
  }
}
```

```
public E pop() {
  // is the stack empty?
  if (this.topIndex == -1) {
    throw new EmptyStackException();
  }
  // get the element at the top of the stack
  E element = this.stack[this.topIndex];

  // adjust the top of stack index
  this.topIndex--;

  // return the element that was on the top of the stack
  return element;
}
```

```java
/**
  * Returns true if this stack is empty, false otherwise.
  * @return true if this stack is empty, false otherwise
  */
 public boolean isEmpty() {
   return this.topIndex == -1;
 }
```

Generic *Methods*

- a generic methods are methods that introduce their own type parameters
  - similar to declaring a generic type, but the type parameter scope is limited to the method
- a generic method header includes a list of type parameters inside angled brackets before the method's return type

```java
/**
 * Utility methods for stacks.
 */
public class Stacks {
  private Stacks() {
    // empty by design
  }

  /**
   * Remove all elements from a stack.
   * @param t a stack
   */
  public static <E> void clear(Stack<E> t) {
    while (!t.isEmpty()) {
      t.pop();
    }
  }
}
```

Bounded Type Parameters

sometimes you will want to *restrict* the types that can be used as type arguments in a parameterized type

for example, suppose we want to write a method that finds the maximum value in a stack

- because we are looking for a maximum value, we want to restrict the type of element in the stack to some type that extends **`Number`**
  - **`Number`** is the superclass for the numeric wrapper classes

Bounded Type Parameters

a bounded type parameter is a type parameter that has some sort of restriction on the allowable types

to restrict the allowable types to subclasses of Number you use the keyword **`extends`** after the parameter name followed by its *upper bound* (the upper-most class in the inheritance hierarchy that you want to allow)

```java
public static <T extends Number> T max(Stack<T> t) {
    double maxValue = Double.NEGATIVE_INFINITY;
    T result = null;
    Stack<T> u = new Stack<>();
    while (!t.isEmpty()) {
        T elem = t.pop();
        double val = elem.doubleValue();
        if (val > maxValue) {
            maxValue = val;
            result = elem;
        }
        u.push(elem);
    }
    while (!u.isEmpty()) {
        t.push(u.pop());
    }
    return result;
}
```

Bounded Type Parameters

a better way to compare values for reference types is to use the **`Comparable`** interface

can we restrict the type to all types that implement **`Comparable`**?

- yes!

```
public static <T extends Comparable<T>> T max(Stack<T> t) {
    T maxVal = null;
    Stack<T> u = new Stack<>();
    while (!t.isEmpty()) {
        T elem = t.pop();
        if (maxVal == null) {
            maxVal = elem;
        }
        else if (elem.compareTo(maxVal) > 0) {
            maxVal = elem;
        }
        u.push(elem);
    }
    while (!u.isEmpty()) {
        t.push(u.pop());
    }
    return maxVal;
}
```

Bounded Wildcards

Suppose that we want our **`Stack`** class to provide a method that pushes all of the elements of a collection onto the stack

Seems easy…

Attribution
the content of these slides is based on **Item 31** of Effective Java 3rd Edition by Joshua Bloch

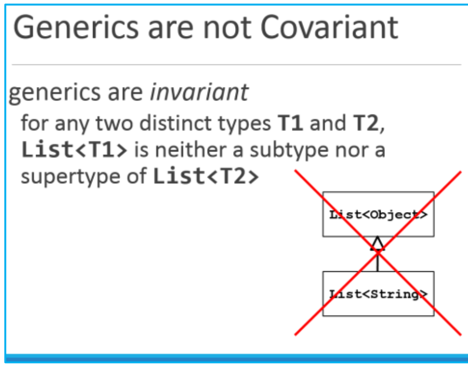
```
public class Stack<E> {
  // fields, constructors, other methods not shown

  /**
   * Push all elements in the specified collection onto this stack.
   *
   * @param src the source collection
   */
  public void pushAll(Collection<E> src) {
    for (E elem : src) {
      this.push(elem);
    }
  }

}
```

Bounded wildcards

consider the signature of the method **`pushAll`** for a **`Stack<Number>`**



Generics are not Covariant

generics are *invariant*

for any two distinct types **T1** and **T2**, **`List<T1>`** is neither a subtype nor a supertype of **`List<T2>`**

```
pushAll(Collection<Number>)
```

Wouldn't be able to call it with **`Collection<Integer>`**

- **`Collection<Integer>`** is not substitutable for **`Collection<Number>`**

This is inconvenient because we can certainly push an **`Integer`** onto a **`Stack<Number>`**

Bounded wildcards

we want some way to say

```
pushAll(Collection<some subtype of Number>)
```

where the exact type doesn’t actually matter

Java provides a special kind of parameterized type called a *bounded wildcard type* for this sort of situation:

```
pushAll(Collection<? extends Number>)
```

```java
public class Stack<E> {
  // fields, constructors, other methods not shown

  /**
   * Push all elements in the specified collection onto this stack.
   *
   * @param src the source collection
   */
  public void pushAll(Collection<? extends E> src) {
    for (E elem : src) {
      this.push(elem);
    }
  }

}
```

Bounded wildcards

after using the bounded wildcard the following now compiles and runs

```
Stack<Number> t = new Stack<>();
Collection<Integer> src = new ArrayList<>();
src.add(2);
src.add(0);
src.add(3);
src.add(0);
t.pushAll(src);
```

Bounded wildcards

the example

```
pushAll(Collection<? extends Number>)
```

uses an *upper bounded* wildcard

- the wildcard `<? extends Number>` matches **`Number`** and any subtype of **`Number`**
  - i.e., **`Number`** is the uppermost class in its inheritance hierarchy that matches the wildcard

Bounded wildcards

Suppose that we want our **`Stack`** class to provide a method that pops all of the elements of the stack into a collection

Seems easy…

```java
public class Stack<E> {
  // fields, constructors, other methods not shown

  /**
   * Pops all elements of this stack adding them to the specified
   * collection.
   *
   * @param dst the destination collection
   */
  public void popAll(Collection<E> dst) {
    while (!this.isEmpty()) {
      dst.add(this.pop());
    }
  }

}
```

Bounded wildcards

consider the signature of the method **`popAll`** for a **`Stack<Integer>`**

```
popAll(Collection<Integer>)
```

Wouldn't be able to call it with **`Collection<Number>`**

the previous example will not compile because **`Collection<Number>`** is not substitutable for **`Collection<Integer>`**

this is inconvenient because we can certainly add an **`Integer`** into a **`Collection<Number>`**

Bounded wildcards

we want some way to say

```
popAll(Collection<some supertype of Integer>)
```

where the exact type doesn't actually matter

Java provides a special kind of parameterized type called a *bounded wildcard type* for this sort of situation:

```
popAll(Collection<? super Number>)
```

```
public class Stack<E> {
  // fields, constructors, other methods not shown

  /**
   * Pops all elements of this stack adding them to the specified
   * collection.
   *
   * @param dst the destination collection
   */
  public void popAll(Collection<? super E> dst) {
    while (!this.isEmpty()) {
      dst.add(this.pop());
    }
  }

}
```

Bounded wildcards

after using the bounded wildcard the following now compiles and runs

```
Stack<Integer> t = new Stack<>();
t.push(2);
t.push(0);
t.push(3);
t.push(0);
Collection<Number> dst = new ArrayList<>();
t.popAll(dst);
```

Bounded wildcards

the example

```
popAll(Collection<? super Integer>)
```

uses a *lower bounded* wildcard

- the wildcard `<? super Integer>` matches **`Integer`** and any supertype of **`Integer`**
  - i.e., **`Integer`** is the lowermost class in its inheritance hierarchy that matches the wildcard

Bounded wildcards

when implementing a generic class or method, you should consider using wildcard types on input parameters that represent producers or consumers

Producer-extends

a producer is an input parameter that produces references for use by the method

in the method

```
pushAll(Collection<? extends E> src)
```

`src` is an input parameter that produces **E** instances that are pushed onto the stack

generic producers should use an upper bounded wildcard

- producer-extends

Consumer-super

a consumer is an input parameter that consumes references inside the method

in the method

```
popAll(Collection<? super E> dest)
```

`dest` is an input parameter that consumes **E** instances that are popped from the stack

generic consumers should use a lower bounded wildcard

- consumer-super

Producer **and** consumer

an input parameter can be **both** a producer and a consumer

recall our **`Stacks`** method **`max`**:

```
public static <T extends Comparable<T>> T max(Stack<T> t) {
    T maxVal = null;
    Stack<T> u = new Stack<>();
    while (!t.isEmpty()) {
        T elem = t.pop();          ←  producer
        if (maxVal == null) {
            maxVal = elem;
        }
        else if (elem.compareTo(maxVal) > 0) {
            maxVal = elem;
        }
        u.push(elem);
    }
    while (!u.isEmpty()) {
        t.push(u.pop());           ←  consumer
    }
    return maxVal;
}
```

Producer **and** Consumer

- if a parameter is **both** a producer and consumer then you cannot use a wildcard type
  - there is *no* type that is simultaneously a subclass of **T** and a superclass of **T**

Comparables are consumers

the **`Stacks.max`** example is very tricky

not only is the input stack both a producer and a consumer, the type **`T`** must implement the **`Comparable`** interface

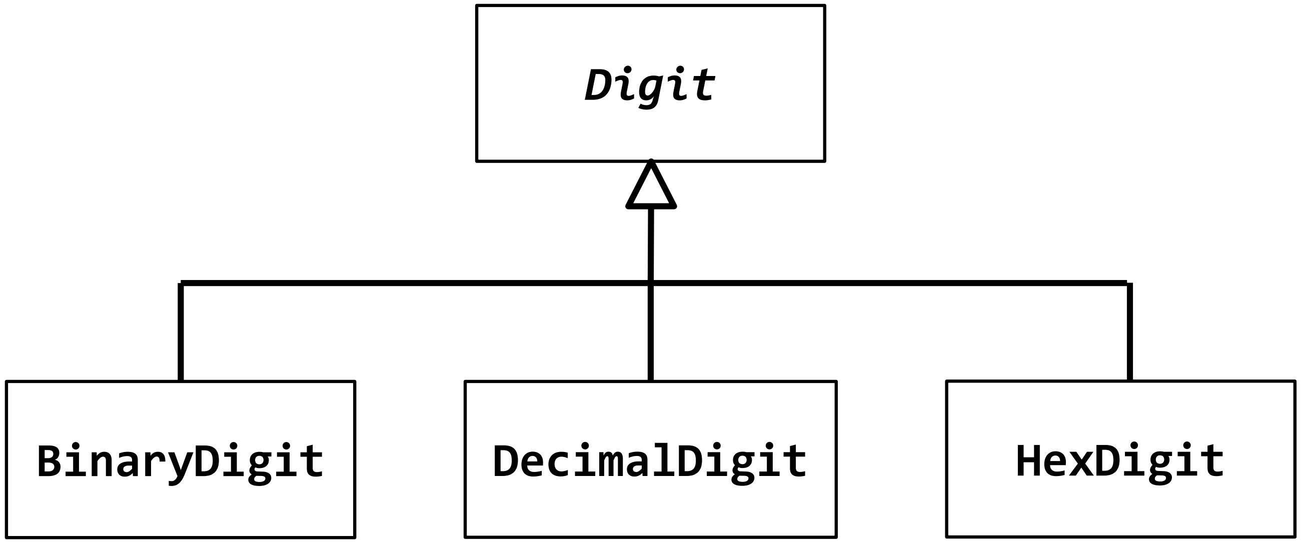
- the **`compareTo`** method of the **`Comparable`** interface is a consumer
  - therefore, we should use a lower-bounded wildcard type for the type parameter of **`Comparable`**

```
public static
<T extends Comparable<? super T>> T max(Stack<T> t)
```

Comparables are consumers

Suppose you have an abstract class **Digit** that represents digits for base-n numbers

Subclasses represent specific base-n numbers



Comparables are consumers

`Digit` can implement **`Comparable<Digit>`**

- all subclasses inherit **`compareTo(Digit)`**

```java
public abstract class Digit implements Comparable<Digit> {

  private int val;

  public Digit(int val) {
    this.val = val;
  }

  @Override
  public int compareTo(Digit other) {
    return Integer.compare(this.val, other.val);
  }

}
```

Comparables are consumers

recall the original declaration of **`max`**:

```
public static
<T extends Comparable<T>> T max(Stack<T> t)
```

what happens if you use a **`Stack<BinaryDigit>`**?

`BinaryDigit` implements **`Comparable<Digit>`**
which doesn't match **`<T extends Comparable<T>>`**

Comparables are consumers

recall the wildcard bounded declaration of **max**:

```
public static
<T extends Comparable<? super T>> T max(Stack<T> t)
```

what happens if you use a **Stack<BinaryDigit>**?

BinaryDigit implements **Comparable<Digit>**

which does match **<T extends Comparable<? super T>>**

Unbounded wildcard

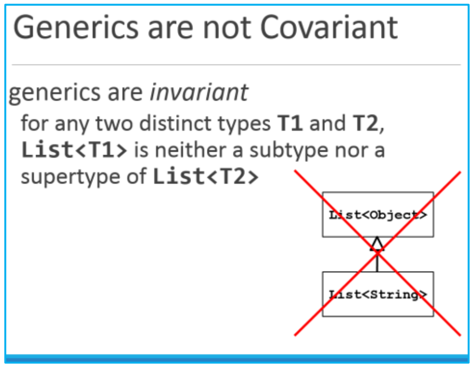
- there are times when you want to write a generic method where the generic type really doesn't matter
  - for example, you can implement the method using only methods from **Object**

```java
public static String fancyToString(Collection<Object> c) {
  StringBuilder b = new StringBuilder();
  for (Object obj : c) {
    b.append("{");
    b.append(obj.toString());
    b.append("}");
  }
  return b.toString();
}
```

Unbounded wildcard

the intent of **`fancyToString`** is to be able to print a collection of any type

- unfortunately this doesn't work
- **Collection<SomeType>** is not
  - a subtype of **Collection<Object>**



Generics are not Covariant

generics are *invariant*

- for any two distinct types **T1** and **T2**, **`List<T1>`** is neither a subtype nor a supertype of **`List<T2>`**

the solution is to use the unbounded wildcard type **?**

Unbounded wildcard

the type

```
Collection<?>
```

is the collection of unknown type

- because the type is unknown, you can't do anything with the collection or its elements that depend on the type
  - in particular, you cannot add anything to the collection except for **`null`**

```java
public static String fancyToString(Collection<?> c) {
  StringBuilder b = new StringBuilder();
  for (Object obj : c) {
    b.append("{");
    b.append(obj.toString());
    b.append("}");
  }
  return b.toString();
}
```

Unbounded wildcard

`fancyToString(Collection<?>)`
will return a string for a collection of any type