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!

```java
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;
}
```

Attribution

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

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
- at first glance, the method is easy to implement…

```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<E> src) {
    for (E elem : src) {
      this.push(elem);
    }
  }

}
```

Bounded wildcards

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

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

- the previous example will not compile because **`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
- at first glance, the method is easy to implement…

```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>)
```

- 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>)
```

```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<? 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 an *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`**:

```java
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;
}
```

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**

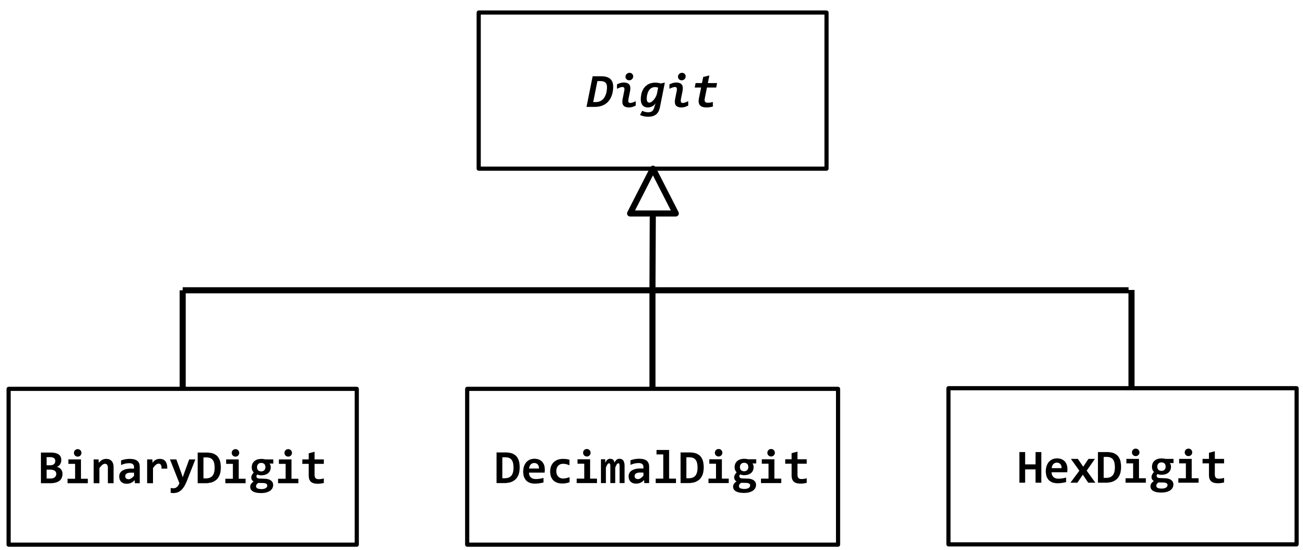
Comparables are consumers

- 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
- 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