Generics

GENERICS WITH COLLECTIONS CLASSES

Generic Programming

- Allows you to write classes (like collection classes and data structures) that can be used with objects of different types
- Allows *type* to be a parameter
- Puts a *compile-time check* on type
- Examples:
  - ArrayList<String>
  - ArrayList<Integer>
  - ArrayList<Button>
  - LinkedList<String>

Without Generics

```
ArrayList myWords = new ArrayList();
myWords.add(“hello”);
myWords.add(“bonjour”);
…
myWords.add(new Integer(4));
for(int i=0; i<myWords.size(); i++) {
      if(words.get(i) instanceof String) {
             String word = (String) myWords.get(i);
      {
}
```

- No type safety- I can add Integers to a list that is meant for Strings
- Because the list holds *any* Object, I have to check the type and cast when removing from the list

With Generics

```
ArrayList<String> myWords = new ArrayList<String>();
myWords.add(“hello”);
myWords.add(“bonjour”);
…
myWords.add(new Integer(4)); // compiler error!
for(int i=0; i<myWords.size(); i++) {
      String word = myWords.get(i);
}
for(String s : myWords) { … }
```

- Compile time safety checking
- No casting
- Can use the for-each loop

Starting in Java 7, you can omit the type from the second angle brackets, like this:
new ArrayList<>();

always use the <>

Using Generics

- As a client
  - We do this all the time when we use the collection classes (ArrayList, LinkedList, etc.)
- As a developer of generic code

WRITING A GENERIC CLASS

Writing Generic Classes

formal type parameter

```
public class Pair <T>{
      private T first, second;

      public Pair(T first, T second) {
              this.first = first;
              this.second = second;
      }
      public T getFirst()  {  return first;           }
      public T getSecond() {    return second;        }
      public void setFirst(T first)  {   this.first = first;     }
      public void setSecond(T second){   this.second = second;   }
}
```

Type Parameter

- The type parameter can be used anywhere else you would use a type
  - variable type
  - return type
  - parameter type
- Classes can have one or more type parameter
- By convention, type parameter names are single, uppercase letters
  - E for an element of a collection
  - K, V for key and value types
  - T, U, S for any other types
- To create an object of our generic class, we perform a *generic type invocation* to replace the type parameter (T) with some concrete value (Integer, String, etc.)

Practice

- Review the Pair example.

Type Erasure

- A generic class is compiled once.
  - There is one single bytecode file.
  - No new classes are created for parameterized types.
- The compiler:
  - Replaces all type parameters with Object (or a *bound*)
  - Inserts type casts to preserve type safety
    - Example: convert int num = numbePair.getFirst() to int num = (Integer) numberPair.getFirst();

Type Erasure

- The JVM (runtime engine) does not have objects of generic types.
- Generic types (like Pair<T>) have a corresponding *raw* type (like Pair)
  - In the raw type, the generic type parameters (T) have been removed or *erased* and replaced with their *bounded types* (Object or, for example, Comparable)
- The raw type is just like a normal class that you would have written before you knew about generics!

- Review the PairRaw class.

GENERIC METHODS

Generic Methods

- We can put generic methods inside of normal, non-generic classes.
  - Generic methods can be static or non-static.
- The type parameters are local to that method only.

```
public static <T> void doSomething(T item)
public static <T> T doSomething(T[] things)
```

generic type

return type

Generic Methods

- Invoke as:

```
MyClass.<String>doSomething(myStringArray)

MyClass.<Integer>doSomething(myNumbersArray)
```

- Or you can omit the actual types in most cases (*type inference*)

```
MyClass.doSomething(myStringArray)
```

Practice

- Write a collection of array utility methods.
  - Write a method to create a list of duplicate numbers in an array.
  - Write a method to create a list of duplicate Strings in an array.
  - Convert to using generics.

Bounds

- We often want to restrict a generic type to only *some* types.
- We do this by specifying `<T extends SomeClassOrInterface>`
  - SomeClassOrInterface is the *upper bound*
  - This means T either “extends SomeClass” or “implements SomeInterface”
- Commonly, we see

```
            T extends Comparable<T>
```

- Note that it is always **extends**, even if the bounding type is an interface.

Bounds

- Setting a bound limits the types that can be used to instantiate the generic type
- It also allows you to invoke methods from the bounded type
- Example:
  - T extends Comparable<T>
  - we can now invoke .compareTo on any T object!
  - (More to come on this later!)

Practice

- Write a collection of array utility methods.
  - Write a method to find the min and max of an int[] and return a Pair<Integer> with the data.
  - Convert to using generics with a bound.

Multiple Bounds

- You can specify T to have multiple bounds

```
T extends Comparable<T> & Serializable
```

- T can only extend one class and the class must come first
- T can extend multiple interfaces

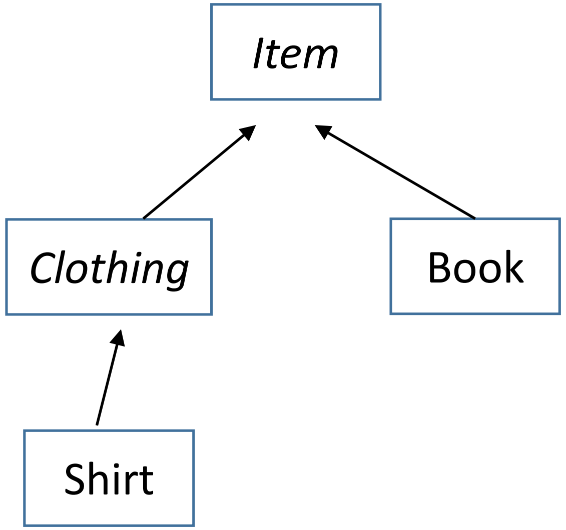
```
T extends MyClass & MyInterface1 & MyInterface2
```

Bounds and Type Erasure

- For a bounded generic parameter type, when the code is compiled, the parameter type is replaced by the bound.
- Examples:
  - public class Pair<T>: T is replaced with Object
  - public static <T> T processArray (T[] array): T is replaced with Object
  - public static <T extends Comparable> T findMin(T[] array): T is replaced with Comparable

Practice

- Review the Item classes.
- Review the FeatureSpot class and examples.
- Modify the FeatureSpot class so it can only hold Item objects.



GENERICS RULES

Rules for Generics

1. Type parameters cannot be instantiated with primitives.
   - Because of type erasure
2. `instanceof` (and `getClass()`) only check the raw type.
   - `if(obj instanceof Pair<Integer>) //` not allowed
   - `if(obj instanceof Pair<T>) //` not allowed
   - `if(obj instanceof Pair)` // allowed
   - `if(obj instanceof Pair<?>)` // allowed
3. You cannot create arrays of parameterized types.
   - `Pair<String>[] arrayOfPairs = new Pair<String>[10];` // not allowed
   - If you need to do this, create an ArrayList, which *is* allowed:

   -
     ```
     ArrayList<Pair<String>> arrayListOfPairs =
               new ArrayList<Pair<String>>();
     ```

Rules for Generics

4. You cannot instantiate type variables.
   - `T first = new T();` // not allowed
   - There are workarounds in Java 8 or with reflection, but it gets complex.
5. You cannot construct a generic array.
   - You **can** construct an Object[] and cast:

     ```
     Object[] myArray = new Object[10];
     T[] myTArray = (T[]) myArray;
     ```
6. In a generic class, you cannot create static generic variables or use the generic type in static methods.
7. Beware of clashes after erasure.
   - `public boolean equals(T value)` // will clash with equals inherited from Object

Rules for Generics

8. You cannot catch or throw generic objects.
9. You cannot overload a method where the formal parameter types erase to the same raw type.
   - `public void print(Pair<String> pair) {…}`
   - `public void print(Pair<Integer> pair) {…}`

GENERICS AND INHERITANCE

Generics and Inheritance

- The parent/child relationship does not translate into generic classes instantiated with a parent/child.
- Example:

```
    public class Documentary extends Movie {…}
    Movie movie = new Movie(…);
    Movie doc = new Documentary(…);
```

- Allowed because Documentary is a child class of Movie. A Documentary *is a* Movie.

Generics and Inheritance

```
Pair<Movie> moviePair = new Pair<>();
moviePair.setFirst(movie);
moviePair.setSecond(doc);
```

- Allowed because Documentary is a child class of Movie.
  - setSecond expects an object of type Movie.
  - A Documentary *is a* Movie.
- The Pair class expects Movie objects so I am allowed to send in a Movie object or *any compatible type*.

Generics and Inheritance

- Two generic types are **not** parent/child just because their parameters are parent/child. The relationship does not carry over.

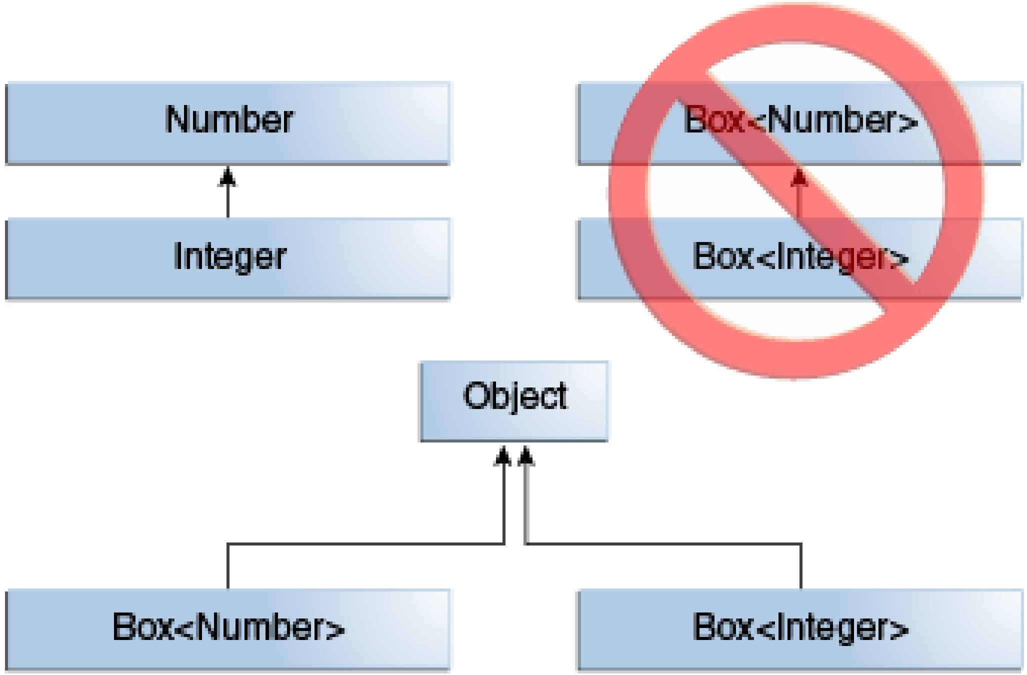
  ClassA extends ClassB

  List<ClassA> does not extend List<ClassB>

  Pair<Documentary> does not extend Pair<Movie>

  there is no relationship between these two classes!

Generics and Inheritance



Generics and Inheritance

```
public static void findWinner(Pair<Movie> moviePair) {…}

Pair<Movie> movies = new Pair<>(mov1, mov2);
Pair<Documentary> docs = new Pair<>doc1, doc2);

movies.setSecond(doc2); // allowed
findWinner(movies); // allowed
findWinner(docs); // not allowed
movies = new Pair<Documentary>(doc1, doc2); // not allowed
```

WILDCARDS ?

Upper Bounded Wildcard Types

- We cannot pass in an object such as `MyClass<Child>` when `MyClass<Parent>` is expected.
  - Expected: Pair<Movie>
  - Not allowed: Pair<Documentary>
- Instead, we use a wildcard parameter, which allows type parameter to vary.

```
? extends Parent
```

```
public void method(MyClass<? extends Parent> myClass)
```

- The method can now accept `MyClass<Parent>` or `MyClass<Child>`

Upper Bounded Wildcard Types

- An upper bund wildcard allows us to specify that a type can be that type *or lower* on the inheritance chain.

```
? extends Parent
```

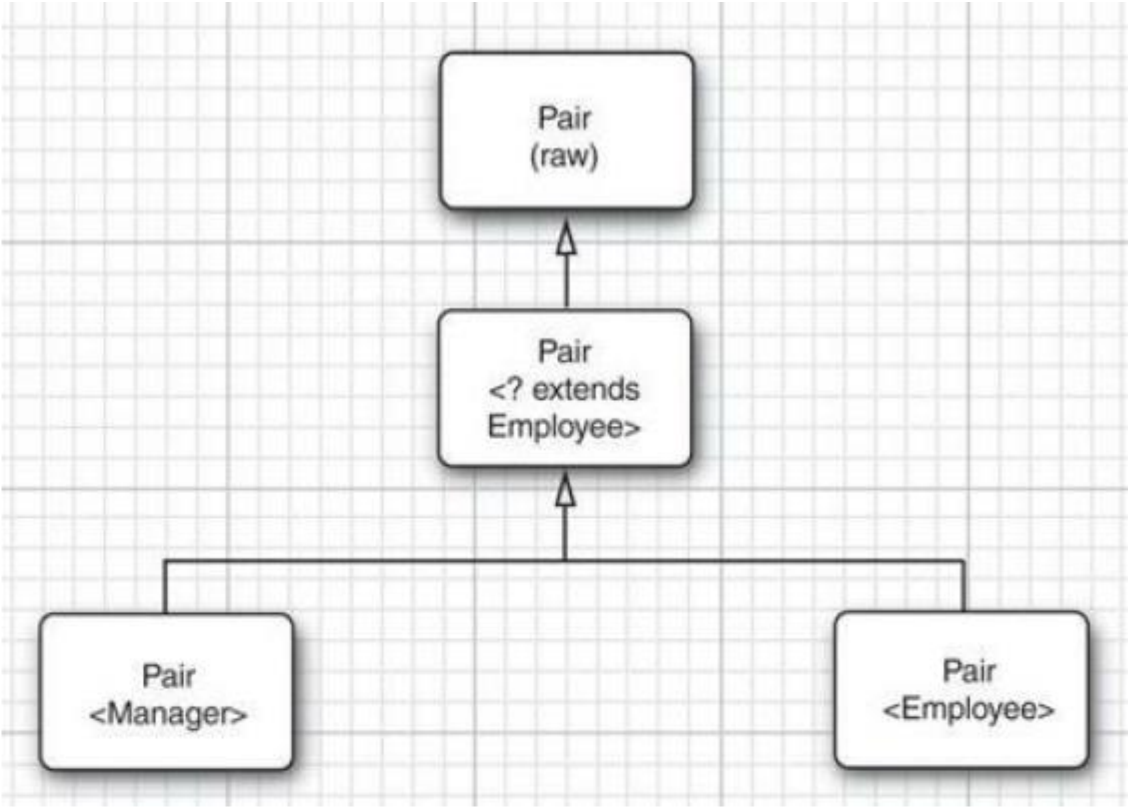
```
public void method(MyClass<? extends Parent> myClass)
```

- The method can now accept `MyClass<Parent>` or `MyClass<Child>`

```
public void findWinner(Pair<? extends Movie> moviePair)
```

- The method can now accept `Pair<Movie>` or `Pair<Documentary>`

Upper Bounded Wildcard Types



Practice

- Review the upper bounded methods.

Upper Bounded Wildcard Types

```
public void printItem(FeatureSpot<Item> spot) { … }
```

- Cannot invoke with a FeatureSpot<Book> or FeatureSpot<Clothing>

```
public void printItem(FeatureSpot<? extends Item> spot) {…}
```

- Can invoke with a FeatureSpot<Book> or FeatureSpot<Clothing> or FeatureSpot<Shrit>

Upper Bounded Wildcard Types

- In a method that uses a wildcard, you might **not** be able to access some methods of the generic type.
- Review the upperBoundRestrictions method.

Upper Bounded Wildcard Types

```
public void featureItem(FeatureSpot<? extends Item> spot, Item item) {
      featureSpot.featureItem(item); // not allowed
      Item i = featureSpot.getItem(); // allowed
}
```

- The FeatureSpot spot can hold *any* object that extends Item.
- The Item item can hold *any* object that extends Item.
- And the two *real* types might not be the same!
  - Example: spot could hold Clothing and item could be a Book
- So this is not allowed!
- Essentially, getters are allowed, setters are not.

Lower Bounded Wildcards

- Lower bounded wildcards allow us to specify that a type must be at least that type *or higher* on the inheritance chain.

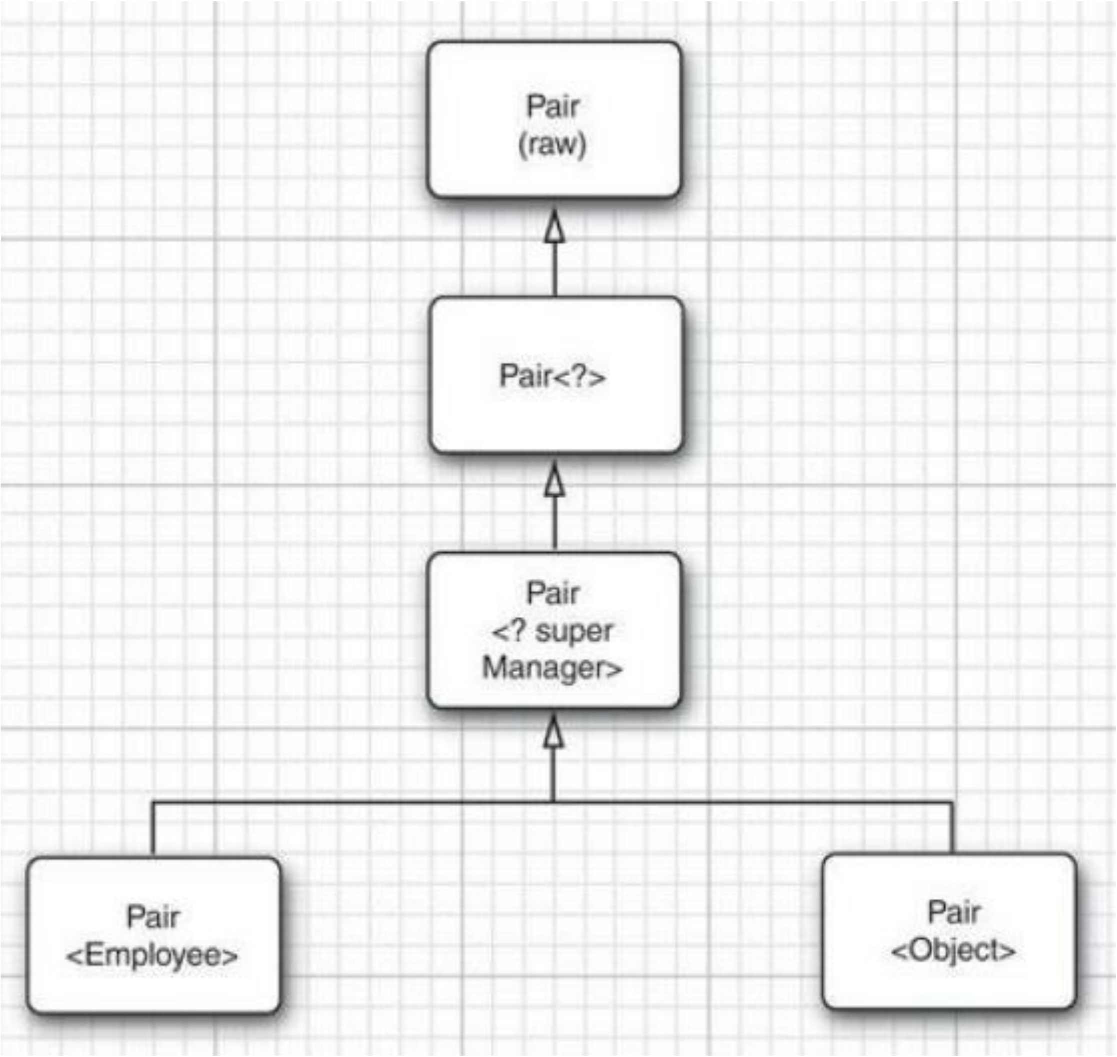
```
? super Child
```

- Essentially the opposite of `? extends Parent`

```
public void method(MyClass<? super Child> myClass)
```

- The method can accept `MyClass<Parent>` or `MyClass<Child>`

Lower Bound for Wildcards



Practice

- Review the lower bounded methods.

Lower Bounded Wildcard Types

```
public void featureItem(FeatureSpot<Item> spot) { … }
```

- Cannot invoke with a FeatureSpot<Book> or FeatureSpot<Clothing>

```
public void featureItem(FeatureSpot<? super Clothing> spot) {…}
```

- Can invoke with a FeatureSpot<Item> or FeatureSpot<Clothing>
- Cannot invoke with a FeatureSpot<Shrit>

Lower Bounded Wildcard Types

- In a method that uses a wildcard, you might **not** be able to access some methods of the generic type.
- Review the lowerBoundRestrictions method.

Lower Bounded Wildcard Types

```
public Item featureItem(FeatureSpot<? super Clothing> spot, Clothing
cItem) {
      spot.featureItem(cItem); // allowed
      Clothing c = spot.getItem(); // not allowed
}
```

- The spot can hold Clothes or something higher- Clothes, Items, or Objects.
  - So setting a Clothing item will be acceptable for any of those types.
- The spot might not hold clothes- it might hold Items.
  - So we can’t assume we can retrieve a Clothing object.
- Essentially, setters are allowed, getters are not.

Upper and Lower Bounded Wildcards

- Wildcards with upper bounds (<? extends Parent>) let you read and save information from a generic object.
  - Use these for “in parameters” (“getters”)
- Wildcards with lower bounds (<? super Child>) let you write information to a generic object.
  - Use these for “out” variables (“setters”)
- If you need both “in” and “out” functionality, don’t use a wildcard.

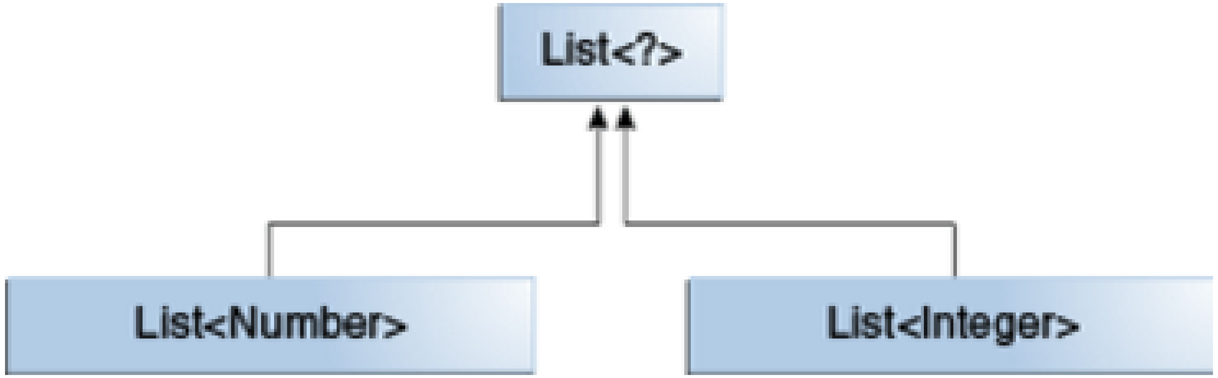
Unbounded Wildcards

- Use an unbounded wildcard when essentially you don’t need to access anything about the actual type- that don’t depend on the type parameter.

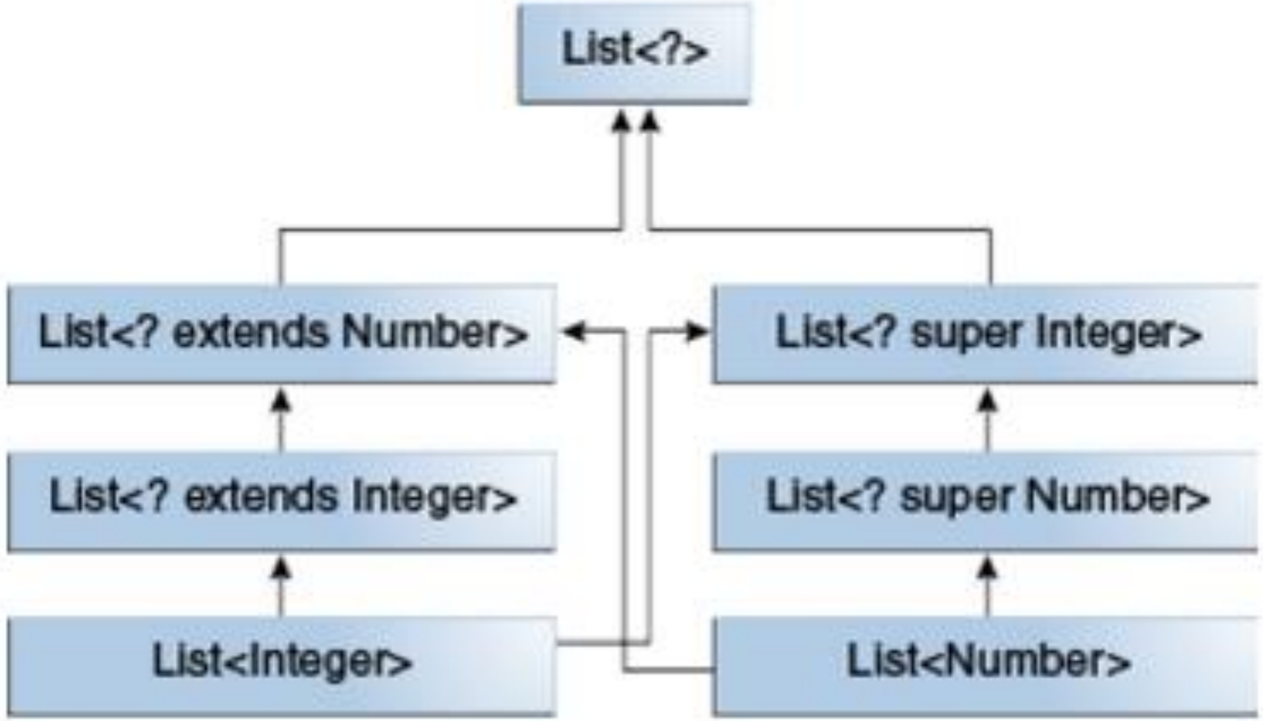
```
<?>
```

- Unbounded wildcards let you access any of the methods inherited from Object.

Unbounded Wildcards



Unbounded Wildcards



Unbounded Wildcards

- Review the unbounded wildcard method.

WILDCARDS AND EQUALS/COMPARABLE

Practice

- Review the Box class.

Equals Methods for Generic Classes

- You cannot test if an object is an instanceof GenericClass<T>
  - Only GenericClass<?>
- For a method like equals, that’s okay!
- You’ll rely on the underlying class to compare itself to the other object.
- Review the Box example.

Implementing Comparable in Child Classes

- We use generics to implement Comparable:

```
public class MyClass implements Comparable<MyClass>
public class Item implements Comparable<Item>
```

- But it gets tricky with child classes:

```
public class Clothing implements Comparable<Clothing> // not allowed!
public class Clothing implements Comparable<Item> // allowed
```

Then, we can override compareTo(Item) and do type checking to make sure the object is type Clothing.

Practice

- Review the Item/Clothing compareTo methods

Creating Generic Classes that Use Comparable Types

- We could restrict our collection to hold generic types like this:

```
public Box<T extends Comparable>
```

- But this doesn’t use Comparable appropriately.
    - It uses the raw version of Comparable.
    - We don’t want to do this.
- So we should do this:

```
public Box<T extends Comparable<T>>
```

- This now uses Comparable correctly.
- But…

Practice

- Review the Box class example and try to create a Clothing box.
    - Problem! Clothing does not implement Comparable<Clothing>!

Creating Generic Classes that Use Comparable Types

- Not all classes implement Comparable<T>!
- Some classes implement Comparable<ParentOfT>
  - Example: Clothing doesn’t implement Comparable<Clothing>, it implements Comparable<Item>
- And remember:
  - Child extends Parent but
  - Comparable<Child> does NOT extend Comparable<Parent>
  - Inheritance does not “translate” in this way!
- So when the Box class header is written this way, we cannot create a Box of Clothing!
  - It would require Clothing to implement Comparable<Clothing>, which it can’t do!

Creating Generic Classes that Use Comparable Types

- Instead, use this:

```
<T extends Comparable<? super T>>
```

```
public Box<T extends Comparable<? super T>>
```

- This has now covered all bases because
  - Item implements Comparable<Item>
  - Clothing implements Comparable<Item>

Creating Generic Classes that Use Comparable Types

- Bottom line, use this as the type parameter whenever writing a generic class that should be restricted to comparable/sortable objects.

```
<T extends Comparable<? super T>>
```