Java Generics

Slides derived from the work of
Dr. Amy McGovern and Dr. Deborah Trytten
Notes

• Grade copies into Canvas largely caught up
  • Missing Lab 6 and Exam 1
  • Some caveats...

• Lab 7 deadline is Monday
• Project 2: deadline is next Wednesday
• Don’t forget code reviews for project 2

• No office hours on Friday
  • We are still available for email
Arrays Class

Provides, among other things, static methods for sorting primitive arrays of different types (byte, char, int, double)
Arrays Class

Problems with this?
• Separate implementation for each type
• Each new type needs a new implementation

Solutions?
Arrays Class

Solutions?

• Could provide a static method that sorts an array of Objects
Arrays Class

Could provide a static method that sorts an array of Objects

• But - what does it mean to compare two arbitrary Objects so that we can establish an ordering between them?
  • For example a String and an Integer?

• We really need a way of talking generically about a homogeneous array of Objects
Java Generics

• A type becomes a parameter to a class and/or a method:

```java
public ClassName<T>{
    :
}
```

• T is the variable type that is assigned when we use the class

• Within the class definition, we can “pretend” that it is a real type (parameters, variable declarations and return types)
GenericStack example ...
Standard Generic Type Names

• E - Element (used extensively by the Java Collections Framework)
• K - Key
• N - Number
• T - Type
• V – Value
Advantages of Generics

• Code reuse
  • ArrayList, Java Collections Framework

• Specific types are checked at compile time (as opposed to everything having to be an Object)
  • Reduces runtime errors

• Easier to read and understand code when we can be very explicit about types
Notes

• Primitive types cannot be used as generic types
  • Must use the wrapper classes
• Type erasure: generics are checked at compile time, not at runtime
  • This decision was made to maintain backward compatibility
  • Not a serious issue most of the time
Implications of Type Erasure

• Cannot construct objects of type E
  
  ```java
  E myData = new E();  // illegal code
  ```

• Cannot construct arrays of type E
  
  ```java
  E[] elements = new E[capacity];  // illegal
  ```

• Solution to the latter: create an array of objects and then cast to array of E
Implications of Type Erasure

• `instanceof()` cannot distinguish same class with different generic type, because it is done at run time
  • `ArrayList<Integer>` and `ArrayList<String>` are the same type according to `instanceof`
• Exception classes cannot be generic
• Static data cannot be of a generic type
Inheritance and Generics

• In many situations, we might have more than one generic type as part of a class or method definition

• These could be arbitrary types or we might want them to have some specific relationship
  • For example: we might want T1 to be a superclass of T2
Class Hierarchies

Object

Number

Integer  Double
Class Hierarchies

Integer \( i = \text{new Integer}(42); \)
Number \( n = \text{new Integer}(1138); \)
Class Hierarchies

Object

Number

Integer    Double

ArrayList<Number>

???

ArrayList<Integer>
Class Hierarchies

Arraylist<Number>

Arraylist<Integer>
The only common (specific) ancestor is Object…
GenericTest example
Wildcards

But, there is a hierarchy that we can use...

```
ArrayList<?>
ArrayList<?> extends Number
ArrayList<Integer>
ArrayList<Double>
ArrayList<Number>
```
Wildcards

But, there is a hierarchy that we can use...

"ArrayList of anything"

```
ArrayList<? extends Number>
```

```
ArrayList<Integer>
```

```
ArrayList<Double>
```

```
ArrayList<Number>
```

```
ArrayList<?>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Integer>
```

```
ArrayList<Double>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
```

```
ArrayList<Number>
Wildcards

But, there is a hierarchy that we can use...

"ArrayList of anything that is a subclass of a Number"
Wildcards

ArrayList<Integer> list1 = new ArrayList<Integer>();
ArrayList<? extends Number> list2 = list1;  // Legal
Wildcards

The complement...

ArrayList<? super Number>

ArrayList<? super Number> -> ArrayList<Object>

ArrayList<Object> -> ArrayList<? super Number>

ArrayList<? super Number> -> ArrayList<Number>
Wildcards

The complement...

```
ArrayList<? super Number>
```

"ArrayList of anything that is a superclass of a Number"
Wildcards

ArrayList<Object> list1 = new ArrayList<Object>();
ArrayList<? super Number> list2 = list1;    // Legal
Wildcard Example I

Return to Arrays in Java API....
Return to Arrays in Java API

```java
binarySearch(T[] a, T key, Comparator<? super T> c)
```

• The class that is passed as the third parameter must implement the Comparator interface for type T or a superclass of type T
Wildcard Example II

Examine Collections in Java API: copy list

```java
public static<T> void copy (List<? super T> dest, List<? extends T> src)
```

Andrew H. Fagg: CS 2334: Java Generics
Examine Collections in Java API: copy list

```java
public static<T> void
copy (List<? super T> dest, List<? extends T> src)
```

• The `<T>` before the method name determines the base type
• The source must be a class that is or extends `T`
• The destination must be a class that is or is a superclass of `T`
Wildcards and Generic Types

• Give us a tremendous amount of flexibility
• Wildcard types are defined and checked at compile time
  • Reduce runtime errors!

• Lab 7: we will define:
  • Generic notion of a Card<T>
  • Generic notion of a Deck<T, E extends Card<T>>