Java Generics
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 class 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

Generic type symbols are arbitrary, but we tend to use a few:

• 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
  
  ```java
  E[] elements = (E[]) new Object[capacity];  // Legal
  ```
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
Administrivia

• Lab 5 grades: coming
• Project 1 grades: posted
• Exam 1 grades & returned exams: posted and emailed
• Lab 7 coming soon
Generics

A type becomes a parameter of another type definition

For example: GenericStack<Person>

• Code reuse
• Standard interfaces
• Type checking at compile time

• Type erasure: generic types are lost at run time
Class Hierarchies

Object

Number

Integer
Double
Class Hierarchies

```java
Integer i = new Integer(42);
Number n = new Integer(1138);
```
Class Hierarchies

Object

Number

Integer  Double

ArrayList<Number>

ArrayList<Integer>
Class Hierarchies

```
ArrayList<Number>
```

```
ArrayList<Integer>
```
Class Hierarchies

The only common (specific) ancestor is Object...
GenericTest example
Wildcards

• We still want a way of saying that we will accept any type as input to a generic
• Or – we want to put constraints on the type
Wildcards

There is a class hierarchy that we can use...

```
Arraylist<? extends Number>

Arraylist<Integer>
Arraylist<Double>
Arraylist<Number>
```
Wildcards

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

“ArrayList of anything”
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
Example: sum a stack of Numbers
Binary Search

Search for a key in an array and return it’s index

• One possible implementation:
  ```java
  public static <T>
      binarySearch(T[] a, T key, Comparator<T> c)
  ```

• The Comparator allows us to compare the key against the elements of the array
• The generic implementation doesn’t require knowledge of the specific object types
Binary Search

Could we be more general about what Comparators are acceptable?

• Suppose T = Double
Binary Search

Could we be more general about what Comparators are acceptable?
• Suppose T = Double

• Could a Comparator<Number> work?
  • Yes! Number allows access to the doubleValue
    • public static int compare(Number d1, Number d2)
    • If(d1.doubleValue() < d2.doubleValue()) return -1;
    • ...

Andrew H. Fagg: CS 2334: Java Generics
Arrays in Java API (actual implementation):

```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
Wildcards

The complement...
Wildcards

The complement...

“ArrayList of anything that is a superclass of a Number”
Wildcards

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

Example: Copy from one GenericStack to another

```java
public static<T> void copy (GenericStack<? super T> dest,
                             GenericStack<? 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>>