Generic Data types

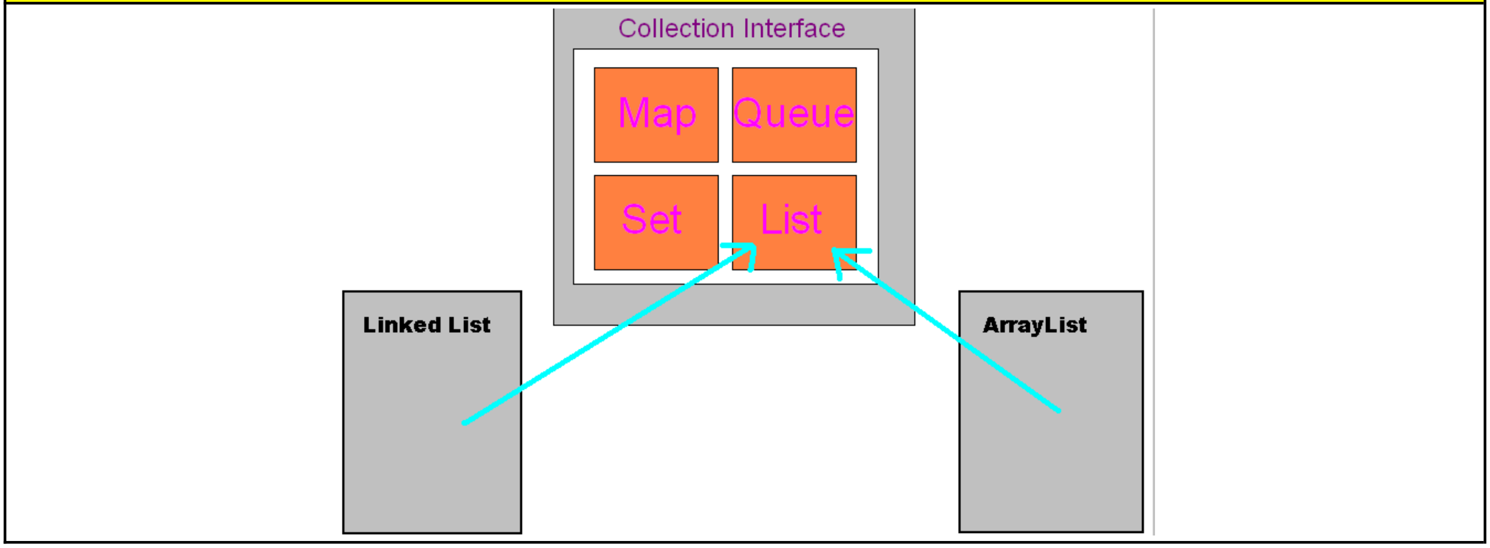
Collections Class Review

- prebuilt data structure(s) that handle ANY custom object we (create) give it
  - why write the same data structure that other people use over and over
- data structures covered today
  - Linked Lists
  - Arrays of Objects (not simple data type arrays)
- data structures covered later
  - Queues
  - Sets
  - Maps
- each data structure has it's pros and cons
- import java.util.LinkedList

Collections Class details

- The class is a huge help to experienced programmer that know what some data structures are.
  - why we cover AFTER Linked Lists and Stacks/Queues
- the Collections class is a SUPER class, so it itself can do many options to the lower data structures it creates
- all functions and sub-classes (as of 1.5) ARE NOW GENERIC
  - does not matter the object, will work with it

The Collection Class and SubClasses



Using Generics in Collections

- remember, only works with NON-simple data types
    - Integer // int != Integer
    - Double // double != Double
- ***ANY*** CREATED DATA TYPES (like NODE)
    - ***THAT'S WHY GENERIC!!! WORKS WITHOUT A LOT CHANGES!!***
- have to "downcast" to type cast when retrieving objects for the data structures
- have to redo (add) a NEW compareTo that works with general Objects

Using Generics with Collections

```
ArrayList<Employee> x = new ArrayList<Employee>();

Employee adjunct = new Employee("Dan", "Malesko", 30);
Employee dean = new Employee("Jack", "McLaughlin", 90);
Employee professor = new Employee("Peter", "Joyce", 60);
IndexCard lupoli = new IndexCard("Prof", "Lupoli", "1800SUPERMAN", 21117);
```

Java Generic's – History

- Pizza: 1996-97, extended Java with generics, function pointers, class cases and pattern matching
- GJ: 1998 derivative of Pizza; Generics the only extension
- Java 1.5: 2004. Modeled after GJ
- PolyJ: 1997, would have required changes to JVM
- NextGen: 1998, avoids **oddities of type erasure**, still compatible with JVM and existing binaries. Extension of GJ

The motivation for Generics

- typesafe polymorphic containers since casting becomes an issue
  - can still produce errors from a bad cast
    - which may only show up during run-time (too late!)

Without Generics, we would have to

Error!

```
List l = new LinkedList();

l.add(new Integer(0));

Integer x = l.iterator().next();
// What happens without type cast?
```

Fixed… but… still stinks we have to do it

```
List l = new LinkedList();
l.add(new Integer(0));
Integer x = (Integer) l.iterator().next(); // need type cast
String s = (String) l.iterator().next(); // bad cast exception
```

But with Generics (no casting!)

```
List<Integer> l = new LinkedList<Integer>();

l.add(new Integer(0));

Integer x = l.iterator().next(); // no need for type cast

String x = l.iterator().next(); // compile-time error
```

Creating Generic Class/Type

- In C++ this was called a Template
- The class you are about to create will take a undetermined TYPE <T>
- This call will interact with the T the SAME WAY no matter the T
    - If you want something to have the SAME behaviors no matter the type, a Generic Class is perfect
- Can
    - return T
    - accept T as a parameter
        - add(T p)
        - p is the actual instance

Generic Class Example

```
import java.util.ArrayList;


public class Hold< T >
{
    ArrayList <T> holdBlock = new ArrayList<T>();

    public T getFirst()
    {
        return holdBlock.get(0);
    }

    public int getLength() { return holdBlock.size(); }

    public void add(T p) { holdBlock.add(p); }

    public boolean isEmpty()
    {
        if(holdBlock.isEmpty())
        { return true; }

        return false;
    }
}
```

// Why an ArrayList is used is very important, it allows Object as a type
// a regular array would not (Type Erasure, covered later)

The General Plan using a Generic Class

```
Hold<Animal> vet = new Hold<Animal>();
```

 Eric	 Amy	 Porche	 Victoria	

```
Hold <Representative> cellblock = new Hold<Representative>();
```

 Politian1	 Politian2	 Politian3	

What would below look like?

```
Hold <Object> tank = new Hold<Object>();
```

Interaction with a Generic Class – Part 1

```
public class Driver {
	public static void main(String[] args)
	{
		Hold <Object> tank = new Hold<Object>();
		tank.holdBlock.add(new String("Lupoli"));
		tank.holdBlock.add(new String("Hyland"));
		System.out.println(tank.getFirst());
	}
}
```

```
Lupoli
```

Interaction with a Generic Class – Part 2

```
public static void main(String[] args)
{
		Dog Amy = new Dog("Amy");
		Animal Eric = new Dog("Eric");
		Cat Porche = new Cat("Porche");
		Animal Victoria = new Cat("Victoria");

		Hold<Animal> vet = new Hold<Animal>(); // NOTICE A SUPER CLASS!!!
		vet.add(Eric);  // added a Dog
		vet.add(Amy);  // added a Dog
		vet.add(Porche);  // added a Cat!!
		vet.add(Victoria); // added a Cat!!

		System.out.println(vet.getFirst());

		Employee adjunct = new Employee("Prof. L", "Lupoli", 30);
		Employee dean = new Employee("Jack", "McLaughlin", 90);
		Employee professor = new Employee("Peter", "Joyce", 60);

		Hold<Employee> cubicle = new Hold<Employee>();
		cubicle.add(professor);
		cubicle.add(dean);
		cubicle.add(adjunct);

		System.out.println(cubicle.getFirst());
}
```

Create a Generic TAMUStack (use “Hold” above as an example) that will accept any Object. It should also contain a **private** ArrayList of GENERIC objects, and size variable. The stack should have a constructor, pop (return and delete), push, peek (return last item entered), constructor, toString (print entire Stack using a loop) and size(using the ArrayList size) function. Within a driver create two different types of

TAMUStacks (use Object you have already coded, like Employee, MPG, etc…)

Answer$_b$:

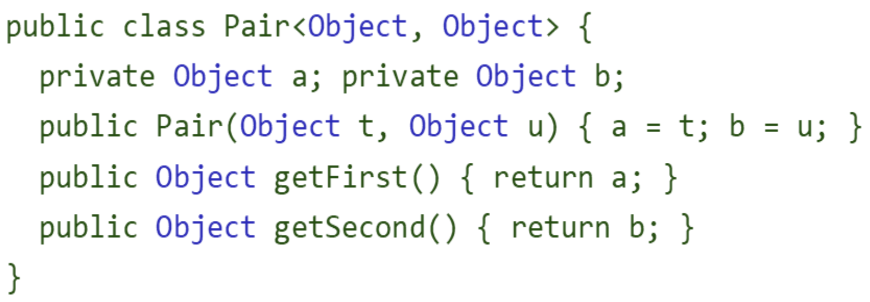
Multiple Object Generic Class

- While the first generic class example accepted one generic object <T>, generic class can accept unlimited number of generic objects
    - gets weird fast, and could have type issues later
        - which we can fix
- Stand alone class meaning no direct instance is made
    - a ***static*** method will use the class/new type
- Remember, the example given would be a new data type, BUNDLED from 2 separate data types!!
    - T and S

Generic Class stores (& returns) pairs of objects

Class Setup

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

    public Pair(T first, S second)
    {
        this.first = first;
        this.second = second;
    }

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

    @Override
    public String toString() {
        return "Pair [first=" + first + ", second=" + second + "]";
    }
```



```
public class Pair<Object, Object> {
  private Object a; private Object b;
  public Pair(Object t, Object u) { a = t; b = u; }
  public Object getFirst() { return a; }
  public Object getSecond() { return b; }
}
```

}
(driver below)

Driver and Call Setup

```
public class PairDriver
{

	public static void main(String[] args)
	{
		String[] names = { "Walter Hyland", "Kris Darlington", "John Phillips",
"John Styles", "Greg Reardon" };

		Pair<String, Integer> result1 = findFirstOccurence(names, "John");
		Pair<String, Integer> result2 = findFirstOccurence(names, "Shawn");
	}

	// this function should return the value looked for and the index found
	public static Pair<String, Integer> findFirstOccurence(... )
	{
		// you finish!!!


			// what should it return if it does NOT find a match?
	}
}
```

Download from the link below and finish the findFirstOccurrence function ***and*** function header. If NOT found, have it return <null, -1> respectfully. You might have to look up how to get the size of the simple String array. Answer$_b$:
http://faculty.cse.tamu.edu/slupoli/notes/Java/code/Generics/Pair.zip

Generic (Parametrized) Methods

- sounds like you think but
  - method is static
    - don’t need to create an instance
  - uses public static <T> as part of function header
    - then the return type, then function name and parameters and so on

Basic Parametrized Methods Example

Class and Function Setup

```
public class ArrayUtil
{
        // much more here

        public static <E> void print(E[] a) // generic method
        {
                for (E e : a) System.out.print(e.toString() + " ");

                System.out.println();
        }
}
```

Driver - Version 1   (no instantiation of ArrayUtil!)

```
                Rectangle[] rects = . . . ;  String[] strs = . . . ;
                ArrayUtil.print(rects); // uses Rectangle’s toString method to work!
                ArrayUtil.print(strs);
```

Driver – Version 2 (using Explicit Instantiation)

```
                ArrayUtil.<Rectangle>print(rects);
                ArrayUtil.<String>print(strs);
```

Using Generics Methods to sort

- sorting requires the pre-mentioned comparable ***within*** the current object
    - did not inherit in any way, or at least the comparable
    - if not built in the BASE object (super class)
        - then it must be in sub-class
- make sure you have a game plan on WHAT you are sorting
    - the compareTo can be different of each object you want to use
        - the generic class will treat them the same, BUT will use a different measure on what to sort them by!!

Generic Sort with a Non-Inherited Object

Sort Class

```
public class Sort
{
      // since this is public STATIC, no need to create an instance of Sort
      // but use Sort.bubbleSort(z) to use

      public static <T extends Comparable<T>> void bubbleSort(T[] a)
      {
            for(int i = 0; i < a.length - 1; i++)
            {
                  for(int j = 0; j < a.length - 1 - i; j++)
                  {
                        if(a[j+1].compareTo(a[j]) < 0)
                        {
                             T tmp = a[j];
                             a[j] = a[j+1];
                             a[j+1] = tmp;
                        }
                  }
            }
      }
}
```

- but ***why*** re-write a sort function??
    - Collections already has sort functions!!
    - but… what if we wanted the top 10 out of millions (or many)
        - cutting down the time could be a game changer
        - since we only need the first 10, leave the rest alone!!
-

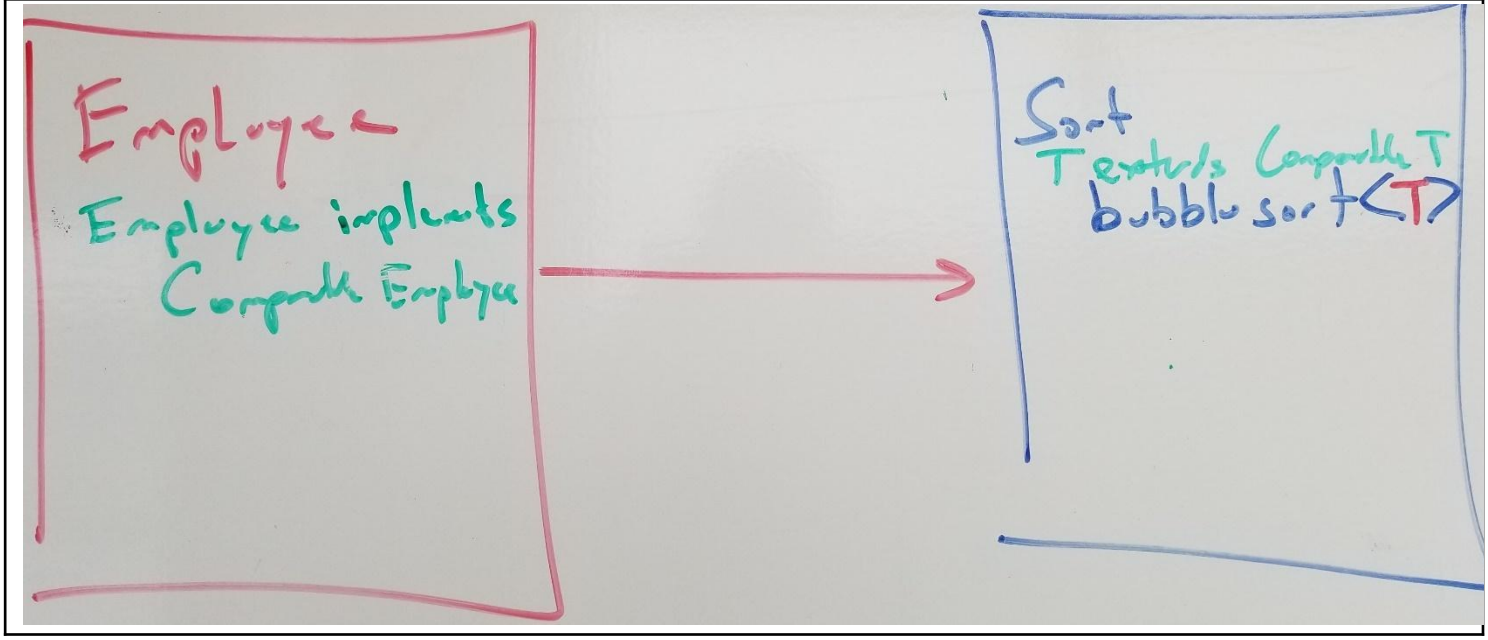
Employee Class

```
class Employee implements Comparable <Employee>
{
    private String firstname, lastname;
    private int age;

    Employee() {}

    Employee(String f, String l, int a)
    {
        firstname = f;
        lastname = l;
        age = a;
    }

    public int compareTo(Employee x)
    {
        if(this.getlastName().equals(x.getlastName()))
                { return this.getfirstName().compareTo(x.getfirstName()); }
        else { return this.getlastName().compareTo(x.getlastName()); }
    }// What values can CompareTo return??


    public String getfirstName() { return firstname; }
    public String getlastName() { return lastname; }
    public int getAge() { return age; }

    public String toString()
    {
        return "Employee [firstname=" + firstname + ", lastname=" + lastname
                + ", age=" + age + "]";
    }

}
```

Driver

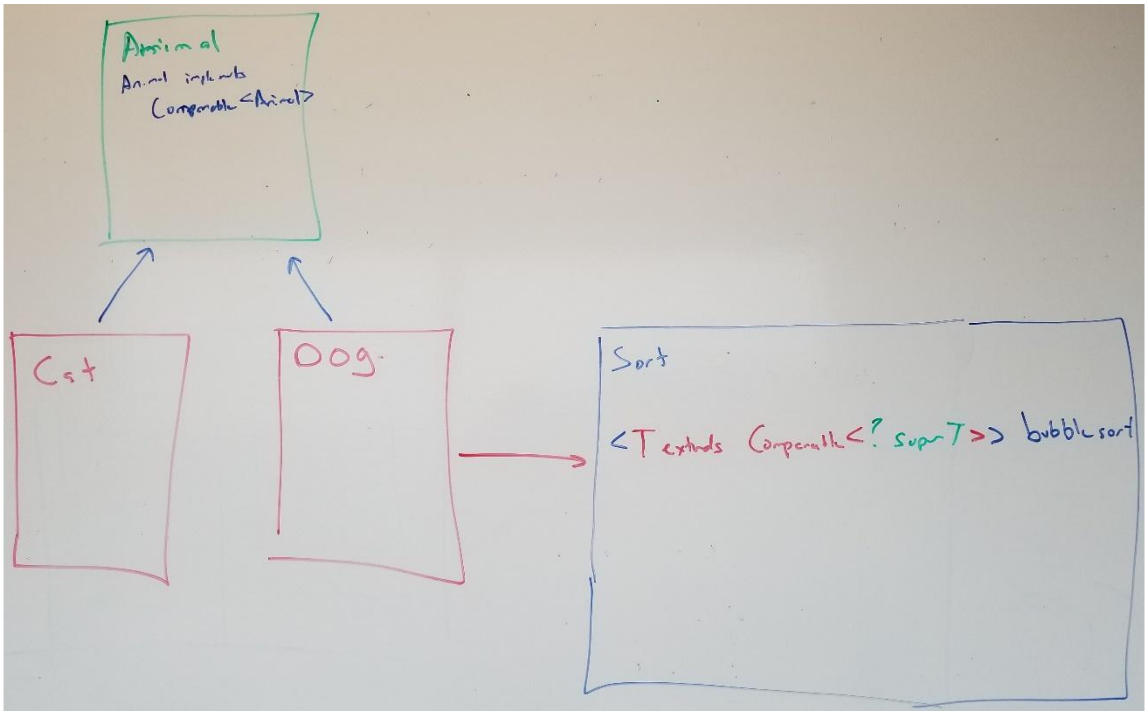
```
Employee [] CCBC = new Employee[3];
CCBC[0] = new Employee("Prof. L", "Lupoli", 30);
CCBC[1] = new Employee("Jack", "McLaughlin", 90);
CCBC[2] = new Employee("Peter", "Joyce", 60);

Sort.bubbleSort(CCBC);
// will sort by Lastname/Firstname (set by Employees compareTo)

// Sort EmSort = new Sort(); // NOT NEEDED since static
// EmSort.bubbleSort(CCBC); // NOT NEEDED since static

for(int i = 0; i < CCBC.length; i++)
{ System.out.println(CCBC[i]); }
```

Overall Logistical Setup – No Inheritance



- Because Dogs and Cats (together) will be sorted by weight, which has been placed in the BASE class Animal, there are some minor changes
  - In Sort class
    - SUPER, checks to see if the SUPER class of Dog/Cat has the comparable needed for the sorting algorithm

Overall Logistical Setup – Using Inheritance



Generic Sort using an Inherited Object Comparable

Sort Class

```java
public class Sort
{
      // since this is public STATIC, no need to create an instance of Sort
      // but use Sort.bubbleSort(z) to use

      public static <T extends Comparable<? super T>> void bubbleSort(T[] a)
      {
            for(int i = 0; i < a.length - 1; i++)
            {
                  for(int j = 0; j < a.length - 1 - i; j++)
                  {
                        if(a[j+1].compareTo(a[j]) < 0)
                        {
                              T tmp = a[j];
                              a[j] = a[j+1];
                              a[j+1] = tmp;
                        }
                  }
            }
      }
} // ? is a wildcard
```

Animal Class

```
public abstract class Animal implements Comparable <Animal>
{
        // set all to protected so subclasses have direct-ish access
        protected String name;
        protected String sound;
        protected String food;
        protected float weight;

        public Animal(String name)
        {
                this.name = name;
        }

        public Animal(String name, float weight)
        {
                this.name = name;
                this.weight = weight;
        }

        public String getName() {return name;}
        public String getSound() {return sound;}
        public String getFood() {return food;}
        public float getWeight() {return weight;}

        // set this to abstract since they may be different depending sub-class
        public abstract void setName();
        public abstract void setSound();
        public abstract void setFood();
        public abstract void setWeight();
        // using abstract here to set a standard
        public abstract String toString();

        public int compareTo(Animal x)
        {
                if(this.getWeight() == x.getWeight()) { return 0; }
                else if(this.getWeight() < x.getWeight()) { return -1; }
                else { return 1; }
        }// What values can CompareTo return??

}
```

Dog Class (Cat is the same)

```java
import java.util.Scanner;

public class Dog extends Animal
{
    private String bark;

    private static Scanner sc = new Scanner(System.in);

    // constructors
    public Dog(String name) { super(name);}
    public Dog(String name, float weight) { super(name, weight); }

    // setters
    public void setName()
    {
        System.out.println("Please enter your Dog's name:");
        this.name = sc.next();
        // access directly to name since protected in super class
    }

    public void setSound()
    {
        System.out.println("Please enter your Dog's barking sound:");
        this.sound = sc.next();
    }

    public void setFood()
    {
        System.out.println("Please enter your Dog's food:");
        this.food = sc.next();
    }

    public void setWeight()
    {
        System.out.println("Please enter your Dog's weight:");
        this.weight = sc.nextFloat();
    }

    public String toString()
    {
        return "Dog [bark=" + bark + ", name=" + name + ", sound=" + sound
                   + ", food=" + food + ", weight=" + weight + "]";
    }

}
```

Driver

```
        Animal [] pets = new Animal [4];
        pets[0] = new Dog("Amy", 110);
        pets[1] = new Dog("Eric", 225);
        pets[2] = new Cat("Porche", 23);
        pets[3] = new Cat("Victoria", 17);

        Sort.bubbleSort(pets); // will sort by weight (set by Animal compareTo)

        for(int i = 0; i < pets.length; i++)
        { System.out.println(pets[i]); }
```

1. In the Driver, create 11+ more instances of Dogs/Cats. (Use the 4 already there).
2. Inside the Sort class, create two more functions
    a. “bubblesortAscending” (which is basically a copy of the bubblesort function
    b. “bubblesortDescending***Top10***”.
        i. Remember, ONLY the top ten. I do not need it to sort everything
        ii. return a new list with only 10 values.
        iii. Hint: I do not care about the integrity of the original list
3. Call the two functions in the Driver to make sure they work. (It’s sorted by weight)

Collections and Generics

- hoping you will notice that Collections uses the same Generics we are creating
- Collections will accept any custom datatype!!
    - hence Generic datatype

WildCards (and intro to Constraints)

- Wildcards are both a convenience feature (more concise syntax), and to add support for co/contravariance for type parameters
- can be used to form ***constraints*** on type parameters
  - using extends and super, or none is also an option
  - ***without*** constraints, only operations that are supported for ***all types*** can be applied to values whose types are type parameters

Wildcard Quick Example

```java
public static void printAll (List<?> l)
{
        for (Object o : l) System.out.println(o);
}
```

Java Wildcard Usage

Name	Syntax	Meaning
WildCard with lower bound	? extends B	Any subtype of B
WildCard with upper bound	? super B	Any supertype of B
Unbounded	?	Any Type

Bounded Wildcard

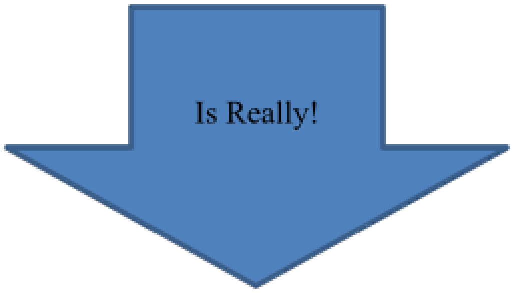
```java
public void addAll(LinkedList<? extends E> other)
{
    ListIterator<E> i = other.listIterator();
    while(i.hasNext())
    {
        add(i.next());
    }
}
// allows any SUBTYPE of “E”
```

Constraints and Bindings (sounds weird)

- Java, too, needs constraints to type parameters
- Without constraints, only operations that are supported for all types can be applied to values whose types are type parameters
- types
  - o extends Object (basic)
  - o extends Comparable
    - then requires the object passed in to have a comparable aspect
- found in class ***AND*** function headers

Everything has Constraints!

```
      public static <E> void print(E[] a) // generic method
      {
            for (E e : a) System.out.print(e.toString() + " ");

            System.out.println();
      }
```



Is Really!

```
      public static <E extends Object> void print(E[] a) // generic method
      {
            for (E e : a) System.out.print(e.toString() + " ");

            System.out.println();
      }

// “Object” justifies toString
```

Other constrained Examples

```
public static <E extends Comparable> void print(List<E> a, E threshold)
{
        for (E e : a)
        {
                if (e.compareTo(threshold) < 0)  // type error !!
                { System.out.print(e.toString() + " "); }
        }
        System.out.println();
}
```

```
public void addAll(LinkedList<? extends E> other)
{
        ListIterator<E> i = other.listIterator();
        while(i.hasNext())
        {
                add(i.next());
        }
}
```

```
public static <T extends Comparable<T>> void bubbleSort(T[] a)
{
```

```
public static <T extends Comparable<? super T>> void bubbleSort(T[] a)
{
```

```
 class SortedList<T extends Comparable & Serializable> // multiple bindings
 {  // . . .

 }
```

```
 public static <E extends Comparable<E> & Measureable> E min(ArrayList …)
 {  // . . .

 }
```

Update your Stack class to constrain the type used to have a Comparable. Create an simple object without a Comparable and see if Java catches the error.

Type Erasure

- Java's JVM (Java Virtual Machine) handles generics rather oddly
- type parameters are actually replaced with ordinary but defined (custom too) Java types
- each type parameter is replaced with its bound (or Object if not bounded)
  - converted into compile-time checks and execution-time casts
  - compiler retains that is was using a Generic class <String>, <Custom>, etc

Type Erasure Example 1

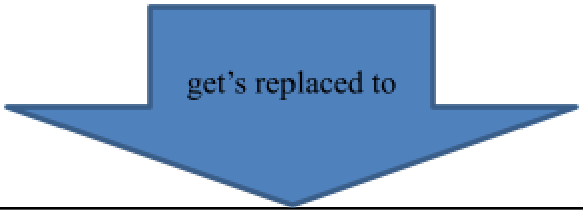
```
List<String> list = new ArrayList<String>();
list.add("Hi");
String x = list.get(0);
```

gets replaced to

```
Listlist = new ArrayList ();
list.add("Hi");
String x = (String) list.get(0);
```

Type Erasure Example 2

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

        public Pair(T first, S second)
        {
                this.first = first;
                this.second = second;
        }

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

        @Override
        public String toString() {
                return "Pair [first=" + first + ", second=" + second + "]";
        }
}
```



get's replaced to

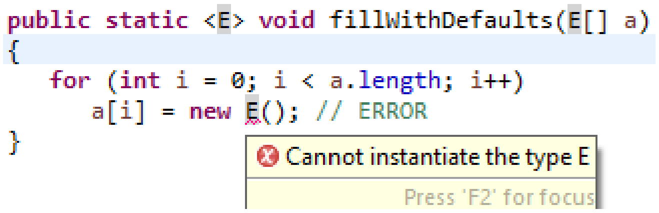
```
public class Pair
{
        private Object first;
        private Object second;

        public Pair(Object first, Object second)
        {
                this.first = first;
                this.second = second;
        }

        public Object getFirst() { return first; }
        public Object getSecond() { return second; }
        public void setFirst(Object first) { this.first = first; }
        public void setSecond(Object second) { this.second = second; }

        @Override
        public String toString() {
                return "Pair [first=" + first + ", second=" + second + "]";
        }
}
```

Issues with Type Erasure

- this identifies the limitation generics has
- biggest issue is the ability to NOT be able to create objects of a generic type
    - because of the type erasure, Object being the lowest form of a generic object

Type Erasure and Instantiating Generic Objects

Doesn’t like it during Compile-Time



```
public static <E> void fillWithDefaults(E[] a)
{
   for (int i = 0; i < a.length; i++)
      a[i] = new E(); // ERROR
}
```

Cannot instantiate the type E

Press 'F2' for focus

And during Type Erasure, no error, but not ***useful***

```
public static void fillWithDefaults(Object[] a)
{
   for (int i = 0; i < a.length; i++)
      a[i] = new Object(); // Not useful
}
```

But you could use default values (passed in)

```
public static <E> void fillWithDefaults(E[] a, E defaultValue)
{
   for (int i = 0; i < a.length; i++)
      a[i] = defaultValue;
}
```

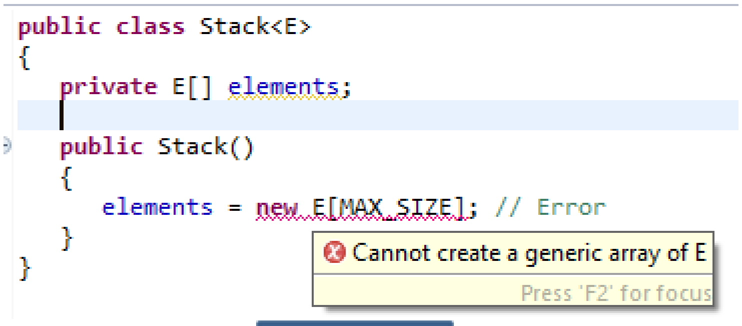
Type Erasure – So What, it didn't effect me!

- but it did, in your TAMU Stack!!
- we used a ***defined Collection*** that ***can hold any*** type

<table>
<tr><th colspan="2">How type erasure has already affected your work</th></tr>
<tr><th>Declared Generic data Hold</th><th>Generic data hold</th></tr>
<tr><td><pre>
import java.util.ArrayList;

public class Stack<E>
{
   private ArrayList<E> elements;

   public Stack()
   {
      elements = new ArrayList<E>(); // Ok
   }

}
</pre></td><td>
<pre>
public class Stack<E>
{
   private E[] elements;

   public Stack()
   {
      elements = new E[MAX_SIZE]; // Error
   }
}
</pre>
Cannot create a generic array of E
Press 'F2' for focus

gets replaced to</td></tr>
<tr><td></td><td></td></tr>
<tr><td></td><td><pre>
public class Stack<E>
{
   private Object[] elements;

   public Stack()
   {
      elements = new Object[MAX_SIZE];
      // Again, Not useful
   }
}
</pre></td></tr>
</table>

Answers

TAMUStack

```java
import java.util.*;

public class TAMUStack<T> {
        ArrayList<T> stackList = new ArrayList<T>();
        private int size = 0;

        public T pop(){
                T temp = stackList.get(size - 1);
                stackList.remove(size-1);
                size = size - 1;
                return temp;
        }
        public void push(T e) {
                stackList.add(e);
                size = size + 1;
        }
        public T peek() {return stackList.get(size - 1);}
        public int getSize() {return stackList.size();}

        public String toString(){
                String temp  = "";
                for(int i = 0; i < getSize(); ++i) {
                        temp = temp + stackList.get(i) + " ";
                }
                return temp;
        }
}
```

```java
public class Cube {
        private double side;
        Cube(double in){side = in;}
        public void setSide(double x) {side = x; }
        public double getSide() { return side; }
        public String toString() { return "This cube's sides are " + getSide(); }

}
```

```java
public class Driver {

        public static void main(String[] args) {
                Cube c1 = new Cube(12);
                Cube c2 = new Cube(13);
                Cube c3 = new Cube(14);
                TAMUStack<Cube> newStack = new TAMUStack<Cube>();
                newStack.push(c1);
                newStack.push(c2);
                newStack.push(c3);
                System.out.println(newStack.pop());
                System.out.println(newStack.pop());
                System.out.println(newStack.pop());
                System.out.println("");

                Circle c4 = new Circle(2);
                Circle c5 = new Circle(3);
                TAMUStack<Circle> newStack2 = new TAMUStack<Circle>();
                newStack2.push(c4);
                newStack2.push(c5);
                System.out.println(newStack2.pop());
                System.out.println(newStack2.pop());

        }

}
```

findFirstOccurrence function

```
public class PairDriver
{

         public static void main(String[] args)
         {
                  String[] names = { "Walter Hyland", "Kris Darlington", "John Phillips", "John Styles",
"Greg Reardon" };

                  Pair<String, Integer> result = findFirstOccurence(names, "John");
                  Pair<String, Integer> nullResult = findFirstOccurence(names, "Bill");
                  System.out.println(result);
                  System.out.println(nullResult);
         }

     // this function should return the value looked for and the index found
         public static Pair<String, Integer> findFirstOccurence(String[] nameList, String name)
         {
                  for(int x = 0; x < nameList.length; x++) {
                           String currentName = nameList[x];
                           String firstName = currentName.substring(0, currentName.indexOf(" "));
                           if(firstName.compareTo(name) == 0) {
                                    return new Pair<String, Integer>(nameList[x], x);
                           }
                  }
                  return new Pair<String, Integer>(null, -1);
         }
}
```

Sources

Dr. Dylan Shell 314 Notes

Parameterized Generic Classes

http://javahowto.blogspot.com/2008/06/java-generics-examples-parameterized.html

http://docs.oracle.com/javase/tutorial/java/generics/types.html

Cay Horstman, “Big Java Late Objects”

Common Problems with wildcards

If we have an abstract Car class, and two subclasses, Toyota and Ford, we can not add a Ford to an
Arraylist<? Extends Car>, since that Arraylist could be an Arraylist of Toyotas.

findFirstOccurrence function

```
import java.util.ArrayList;
public class main {

	public static void main(String[] args) {
		ArrayList<? extends Car> Car_extends_array = new ArrayList();
		ArrayList<? super Car> Car_super_array = new ArrayList();
		ArrayList<Ford> Ford_array = new ArrayList();

		Ford a = new Ford();
		Car b = new Ford();

		//Does not work because Car_extends_array could be an array of Toyotas
		//Car_extends_array.add(a);
		//Car_extends_array.add(b);

		//Works because Car_super_array can be any super type of Car
		Car_super_array.add(a);
		Car_super_array.add(b);

		//Works because it's an array of Fords
		Ford_array.add(a);

		//Does not work because it is a Car not a Ford
		//Ford_array.add(b);
	}

}
```