CS 2100: Data Structures & Algorithms 1

Java Generics

Dr. Nada Basit // basit@virginia.edu

Spring 2022

Friendly Reminders

- Masks are **required** at all times during class (University Policy)
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

Generics in Java

In the context of Vectors

Generics

- An abstraction of data representation

- The idea is to allow type (Integer, String, …etc and user defined types) to be **a parameter to methods**, classes and interfaces.
- For example, classes like ArrayList, Vector, HashSet, HashMap, etc. use generics very well. We can use them for any type.

Generics – An Abstraction (Ensuring type-safety)

E - Element (used extensively by the Java Collections Framework)
K - Key
N - Number
T - Type
V - Value
S,U,W etc. - 2nd, 3rd, 4th types

- There exists a **Generic type**
  e.g. `ArrayList<T>` or `Map<K, V>`
  It specifies what kind of objects can be "**used**" by the class, interface, method, etc.
- List, Set and Map interfaces all accept **Generic** type specifiers
- For instance, ArrayList will accept **any** object, but this line specifies Strings:
  - `ArrayList<String> myList = new ArrayList<String>();`
- This allows the compiler to check for **type safety**
- *All code you will write that deals with generics will be collection-related code!*
- `https://docs.oracle.com/javase/tutorial/java/generics/types.html`

Generics

- A **generic type** is a generic class or interface that is parameterized over types
  - Defined using angle brackets: `ArrayList<T>`
  - Specifies what kind of objects can be "used" by the class
    - Ex: `ArrayList<T>` can store elements of type T
    - Ex: `Map<K, V>` has keys of type K and values of type V

- `E=Element; K=Key; V=Value; N=Number; T=Type; S,U,… = additional`

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

Motivation

- We can make the ***Vector*** from previous slides better...
  - Now, can only store doubles, but what about int, String, etc.
- Would be nice if there was a way to **write one class** that could store ANY type of object
- **Wildcards** and **Generics** allows us to do this.

In a nutshell, generics enable *types* (classes and interfaces) to be **parameters** when defining classes, interfaces and methods. Much like the more familiar *formal parameters* used in method declarations, **type parameters** provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

https://docs.oracle.com/javase/tutorial/java/generics/why.html

Wildcards

- **Wildcards** allow programmers to specify a generic type for an object or parameter.
- Use wildcards when there are no dependencies with other variables or parameters.
  - I.e., this method will accept any type
  - However, the type that is eventually used does not have to match any other variable or parameters (it's standalone)

```
//the ? specifies that the type is a wildcard
//the method will accept any type of List
private static void printlist(List< ? > list)
{
  for(int i=0; i < list.size(); i++) {
    System.out.print(list.get(i) + " ");
  }
}
```

Generics

- **Generics** are essentially the same, but should be used when there is a dependency across variables.
- Below, means there will be a generic and is referred to as **T**
- Later, `params T[] list` and `T value` can be any types
  - But, `T[]` in **list** and `T` in value MUST **match**

```
private static < T > int countOccurences(T[] list, T value) {
    int count = 0;
    for(T next : list ) {
      if(next.equals(value)) count++;
    }
    return count;
  }
```

Example of Generic Class & Using A Generic Class

- You don't see actual types
- Instead, you see **Generics** in the class

```
public class ListNode< T > {

  /* Data being stored in this node */
  private T data;

  public ListNode(T data) {
    this.data = data;
  }

  /* Getters */
  public T getData() { return this.data; }
}
```

- When you actually use the Generic Class you can pick the type.
- Can use many types ====>

```
ListNode< String > n1 = new ListNode< String >("Hello");
System.out.println(n1.getData());

ListNode< Integer > n2 = new ListNode< Integer >(5);

/* This last one is NOT valid. WHY? Only object types allowed */
ListNode< double > n3 = new ListNode< double >(3.45);
```

Some Advantages of Generics

- Useful when we want a data structure to **store any type of object**
- Useful when we have multiple variables whose types **need to be general but match each other**

Some Limitations of Generics

- Cannot set a generic type to a primitive
  - But java provides **object versions** (Integer, Char, etc.) for all primitives.

- Cannot instantiate a generic type
  - `new T()` is NEVER allowed.
  - Can cause heap pollution *(don't worry about what that is)*.

- If you need to instantiate a generic type:
  - Make the type **Object** instead
  - Manually **cast** as needed *(see next couple slide)*

Making our Vector Generic

- Turning Vector into a **Generic class**, we take out the types, put in a placeholder " **T** "

```
public class Vector{

  private double theList[];
  private int size;

  public double find(double value){
      for(int i = 0; i < this.size; i++){
        if(theList[i] == value) return i;
      }
      return -1; //didn't find it
  }

  //More code down here...
}
```

Class attributes/fields
&
find() method:

```
public class Vector < T >{

  private T theList[]; //Problem will arise here in a minute
  private int size;

  public double find(T value){
      for(int i = 0; i < this.size; i++){
        if(theList[i].equals(value)) return i;
      }
      return -1; //didn't find it
  }

  //More code down here...
}
```

But … Problems!!

```
public class Vector < T >{

  private T theList[]; //Problem will arise here in a minute
  private int size;
```

- When converting types to Generics, how do we handle the **constructor**??

```
public class Vector < T >{
  /*...*/
  public Vector(){
    this.theList = new T[100]; //BOOM...NOT ALLOWED
  }
}
```

- Unfortunately, this causes a problem in Java. It is NOT allowed. So, what is the fix? **Casting**!

The solution:

```
public class Vector < T >{

  private T theList[];
  private int size;

  public Vector(){
    this.theList = (T[])new Object[100]; //<--Cast here
  }
```

Solution Strategy:

1. Make the type **Object** instead
2. Manually **cast**

Vector get() method

getAt() method:

```
public class Vector < T >{

  /* From previous slides... */

  /* returns the item at specified index */
  public T getAt(int index){
    if(index >= 0 && index < this.size)
      return theList[index];
    else
      ; //Uh Oh, we went off the bounds of the vector
  }
}
```

Summary

- Most data structures should be **generic**, because that is more flexible.
- From here on out, **ALL** of our **data structures** will be **generic**.
  - *Though we may need to do this Object array trick sometimes.*

Additional Information about Generics

Some references:

** https://docs.oracle.com/javase/tutorial/java/generics/why.html

** GeeksforGeeks - Generics in Java

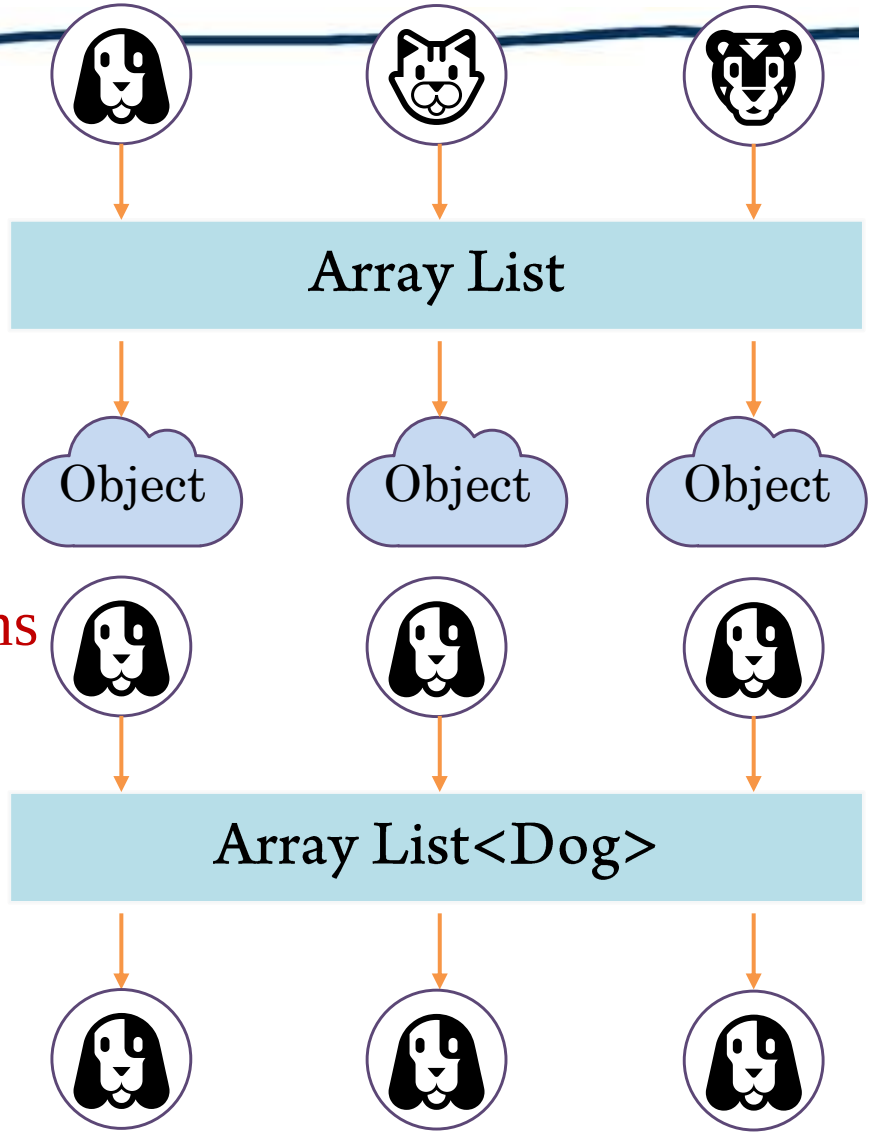
Generics Ensure Type-safety

- **Without generics**, the compiler would happily allow you to put a **Cat** into an ArrayList that was supposed to hold only **Dog** objects (`ArrayList<Object> dogs…`)

```
dogs.add(aCat);//will be allowed!!
```

- **With generics**, you can create type-safe collections to catch problems at *compile-time* instead of run-time (`ArrayList<Dog> dogs…`)

```
dogs.add(aCat);//compile-time error!
```



Generics – examples and why they matter

Use	Example
Creating instances of generified **classes**	• When you create an ArrayList, you have to tell it the type of objects allowed in the list • E.g., … `new ArrayList<Dog>()`
Declaring and assigning **variables** of generic types	• Assigning object instances to variables of generic types (polymorphism with generic types) • E.g., `List<Dog> dogs = new ArrayList<Dog>()`
Declaring (and invoking) **methods** that take generic types	• Passing arguments to methods that are declared to accept generic parameter types • E.g., `void foo(List<Dog> list) {. . .}` `x.foo(dogs)`

Using Generic Classes : Understanding the ArrayList class declaration

The "E" is a placeholder for the REAL type you use when you declare and create an ArrayList

ArrayList is a subclass of AbstractList, so whatever type you specify for the ArrayList is automatically used for the type of the AbstractList

```
public class ArrayList<E> extends AbstractList<E> implements List<E> {

   // Method declaration for adding elements
   public boolean add(E o) {…}

   // more code
}
```

The type (the value of "E") becomes the type of the list interface as well

Here's the important part! Whatever "E" is determines what kind of things you're allowed to add to the ArrayList

Think of "E" as a stand-in for "the type of element you want this collection to hold and return" (E is for Element)

Using the generic parameter with ArrayList

- This code:
  - `ArrayList<String> thisList = new ArrayList<String>();`
- Is treated by the compiler as:

We now have an ArrayList of **Strings**:

```
public class ArrayList<String> extends AbstractList<String>
… {

        // Method declaration for adding elements
        public boolean add(String o) {…}

        // more code
}
```

"**T**" is the convention for a generic type, unless it is used in a collection class where we use "**E**" for the type of element

Example

```
// A Simple Java program to show working of user defined
// Generic classes

// We use < > to specify Parameter type
class Test<T>
{
    // An object of type T is declared
    T obj;
    Test(T obj) {  this.obj = obj;  }  // constructor
    public T getObject()  { return this.obj; }
}
```

```
// Driver class to test above
class Main
{
    public static void main (String[] args)
    {
        // instance of Integer type
        Test <Integer> iObj = new Test<Integer>(15);
        System.out.println(iObj.getObject());

        // instance of String type
        Test <String> sObj =
                            new Test<String>("GeeksForGeeks");
        System.out.println(sObj.getObject());
    }
}
```

When does the substitutability property / Polymorphism work with Generics? [mixing types]

- `ArrayList<Animal> animals = new ArrayList<Animal>();`
  - Since the reference and object types are exactly the same (ArrayList<Animal>), **this will compile!**

- `ArrayList<Animal> dogLst = new ArrayList<Dog>();`
  - Even through Dog extends Animal, substitutability/ polymorphism does NOT apply on the generic type inside the <>. The reference (ArrayList<Animal>) is different type than the object's type (ArrayList<Dog>) **(This will not compile!)**

- **`List<Cat> kitties = new ArrayList<Cat>();`**
  - The reference type (List<Cat>) is a **superclass** of the object's type (ArrayList<Cat>). This is an application of polymorphism on the **container types**. **(This will compile!)**

```
abstract class Animal {
    public String makeNoise() {
       return "…"
    }
}

class Dog extends Animal {
   public String makeNoise() {
       return "Woof!"
    }
}

class Cat extends Animal {
   public String makeNoise() {
       return "Meow!"
    }
}
```

When does the substitutability property / Polymorphism work with Generics? [mixing types]

- `ArrayList<Cat> catdog = new ArrayList<Dog>();`
  - For obvious reasons this does not work, since Cat and Dog are not related in any way. (**This will not compile!**)

- `ArrayList<Cat> catLst = new ArrayList<Cat>();`
  `ArrayList<Animal> animals = catLst;`
  - **This will not compile**, since the new reference animals type (ArrayList<Animal>) is not the same as the type of the object that variable catLst holds (ArrayList<Cat>); once more, polymorphism does not apply on the generic type of the container type ArrayList.

- `ArrayList<Object> myObjs = new ArrayList<Animal>();`
  - Since the reference (ArrayList<Object>) is different type than the object's type (ArrayList<Animal>); polymorphism does not apply on the generic type inside the <>. (**This will not compile!**)

```
abstract class Animal {
    public String makeNoise() {
      return "…"
    }
}

class Dog extends Animal {
   public String makeNoise() {
      return "Woof!"
   }
}

class Cat extends Animal {
   public String makeNoise() {
      return "Meow!"
   }
}
```

How does Generics work with method parameters?

- `public static void takeAnimals(ArrayList<Animal> animals) {…}`

- Method parameters:
  - If a method takes in an ArrayList of a certain type, that is the ONLY type that will be accepted!
  - Polymorphism and substitutability will not work in this case (using the syntax given above)
  - If `Cat` extends `Animal`, and we pass to method `takeAnimals` an ArrayList of `Cat`, it will **NOT** compile since it accepts an ArrayList of `Animal`.

Generics Example in method parameter (1)

```
public static void main(String[] args) {
    ArrayList<Animal> animals = new ArrayList<Animal>();
    animals.add(new Dog("Cleo"));
    animals.add(new Cat("Ginger"));
    animals.add(new Dog("Sandy"));
    takeAnimals(animals);
}
```

Does this work? Yes!

```
public static void takeAnimals(ArrayList<Animal> animals) {
    for (Animal a : animals) {
        Vet.giveShot(a);    }
}
```

```
abstract class Animal {
    public String makeNoise() {
        return "…"
    }
}

class Dog extends Animal {
    public String makeNoise() {
        return "Woof!"
    }
}

class Cat extends Animal {
    public String makeNoise() {
        return "Meow!"
    }
}
```

Generics Example in method parameter (2)

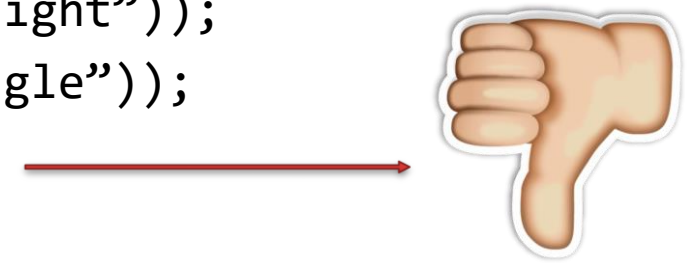
```
public static void main(String[] args) {
    ArrayList<Animal> animals = new ArrayList<Animal>();
    animals.add(new Dog("Cleo"));
    animals.add(new Cat("Ginger"));
    animals.add(new Dog("Sandy"));
    takeAnimals(animals);
    ArrayList<Cat> cats = new ArrayList<Cat>();
    cats.add(new Cat("Midnight"));
    cats.add(new Cat("Pringle"));
    takeAnimals(cats);
}

public static void takeAnimals(ArrayList<Animal> animals) {
    for (Animal a : animals) {
        Vet.giveShot(a);    }
}
```

```
abstract class Animal {
    public String makeNoise() {
        return "…"
    }
}

class Dog extends Animal {
    public String makeNoise() {
        return "Woof!"
    }
}

class Cat extends Animal {
    public String makeNoise() {
        return "Meow!"
    }
}
```

Does this work? No!

Generics Example in method parameter (2)

```
public static void main(String[] args) {
    ArrayList<Animal> animals = new ArrayList<Animal>();
    animals.add(new Dog("Cleo"));
    animals.add(new Cat("Ginger"));
    animals.add(new Dog("Sandy"));
    takeAnimals(animals);
    ArrayList<Cat> cats = new ArrayList<Cat>();
    cats.add(new Cat("Midnight"));
    cats.add(new Cat("Pringle"));
    takeAnimals(cats);
}

public static void takeAnimals(ArrayList<Animal> animals) {
    for (Animal a : animals) {
        Vet.giveShot(a);   }
}
```



The method "takeAnimals" accepts ArrayList<Animal>
Generic types are specific: it is NOT applicable for the argument ArrayList<Cat>!
(Must match!)

Does this work? No!

Why? Type Safety!

Generics Example in method parameter (3)

How do you fix this?

```
public static void main(String[] args) {
    ArrayList<Animal> animals = new ArrayList<Animal>();
    animals.add(new Dog("Cleo"));
    animals.add(new Cat("Ginger"));
    animals.add(new Dog("Sandy"));
    takeAnimals(animals);
    ArrayList<Cat> cats = new ArrayList<Cat>();
    cats.add(new Cat("Midnight"));
    cats.add(new Cat("Pringle"));
    takeAnimals(cats);
}

public static void takeAnimals(ArrayList<? extends Animal> animals) {
    for (Animal a : animals) {
        Vet.giveShot(a);   }
}
```

The wildcard "?" allows the method to accept an ArrayList of any subtype of Animal (such as Cats!)

Any type that **extends** Animal is now allowed!

Generics: Substitutability and Polymorphism

- **Generics are VERY SPECIFIC!**

```
public void takeAnimals(ArrayList<Animal> animals) { ... }
```

- Method only takes `ArrayList` typed with Animal
- Polymorphism and substitutability will not work for `ArrayLists` with other **Generics**
- Can not call with **cats**, such as:
  `takeAnimals(new ArrayList<Cat>()); //` Trying to pass ArrayList of type Cat

  (Given **takeAnimals** takes in an ArrayList of type Animal)

Generics: Substitutability and Polymorphism

- **Generic wildcard**: ?

```
public void takeAnimals(ArrayList<? Extends Animal> animals)
{ ... }
```

- Use the wildcard, `? Extends SomeClass`, to allow polymorphism in generics
- This WILL accept any ArrayList that is parameterized with any subclass of Animal