
CMSC 202H

Generics

Generalized Code

- One goal of OOP is to provide the ability to write reusable, generalized code.
- Polymorphic code using base classes is general, but restricted to a single class hierarchy
- Generics is a more powerful means of writing generalized code that can be used by any class in any hierarchy represented by the type parameter

Motivating Example: Containers

- Almost all programs require that objects be stored somewhere while they are being used
- A container is a class used to hold objects in some meaningful arrangement
- Generics provide the ability to write generalized containers that can hold any kind of object.
 - Yes, arrays can hold any kind of object, but a container is more flexible. Different types of containers can arrange the objects they hold in different ways.

Simple Container

The container class below models a prison cell used to hold a pickpocket.

```
public class Cell1
{
    private PickPocket prisoner;
    public Cell1( PickPocket p ) { prisoner = p; }
    public PickPocket getPrisoner( ) { return prisoner; }
}
```

This class is not very useful or reusable since it can only hold pick pockets.

A More General Cell

By using the Java Object class, we can use our cell to hold any kind of criminal (why?)

```
public class Cell2
{
    private Object prisoner;
    public Cell2( Object p )    { prisoner = p; }
    public Object getPrisoner() { return prisoner; }
}
```

But this approach can lead to some interesting code

Cell2 Example

```
public class Cell2 {  
    private Object prisoner;  
    public Cell2( Object p) { prisoner = p; }  
    public Object getPrisoner( ) { return prisoner; }  
}
```

```
public class Cell2Demo {  
    public static void main (String[ ] args)  
    {  
        // put a pickpocket into a new cell  
        Cell2 cell2 = new Cell2( new PickPocket( ));  
  
        // remove the prisoner, but now he's a Thief!?  
        Thief thief = (Thief)cell2.getPrisoner( );  
  
        // rest of main  
    }  
}
```

One Type per Container

- Using generics we specify the one type of object that our container holds, and use the compiler to enforce that specification.
- The type of object held in our container is specified by a **type_parameter**

Class Definition with a Type Parameter

- A class that is defined with a parameter for a type is called a **generic class** or a **parameterized class**
 - The type parameter is included in angular brackets after the class name in the class definition heading
 - Any non-keyword identifier can be used for the type parameter, but by convention, the parameter starts with an uppercase letter
 - The type parameter can be used like other types used in the definition of a class

Generic Cell

```
public class Cell3<T>
{
    private T prisoner;
    public Cell3(T p)          { prisoner = p; }
    public T getPrisoner( )   { return prisoner; }
}
```

- A class definition with a type parameter is stored in a file and compiled just like any other class
- Once a parameterized class is compiled, it can be used like any other class
 - However, the class type plugged in for the type parameter must be specified before it can be used in a program

Generic Cell Example

```
public class Cell3<T> {  
    private T prisoner;  
    public Cell3(T p)      { prisoner = p; }  
    public T getPrisoner( ) { return prisoner; }  
}
```

```
public class Cell3Demo {  
    public static void main (String[ ] args)  
    {  
        // define a cell for PickPockets  
        Cell3<PickPocket> ppCell =  
            new Cell3<PickPocket>(new PickPocket());  
  
        // define a cell for thieves  
        Cell3<Thief> tCell = new Cell3<Thief>(new Thief());  
  
        // compiler error if we remove a Thief from PickPocket Cell  
        Thief thief = (Thief)ppCell.getPrisoner( );  
    }  
}
```

Using the **ArrayList** Class

Originally, we said:

...

- An **ArrayList** is created and named in the same way as object of any class:

```
ArrayList aList = new ArrayList();
```

(Note that what we are teaching here is an obsolete, simplified form of ArrayList you can use *for now*; for documentation, see: <http://download.oracle.com/javase/1.4.2/docs/api/java/util/ArrayList.html>. Later, we will learn the proper form, after covering Generics.)

...

Using the Generic **ArrayList** Class

Actually:

- An **ArrayList** is created and named in the same way as object of any class, *except that it is actually a generic class, for which you specify a base type, as follows:*

```
ArrayList<BaseType> aList =  
    new ArrayList<BaseType>();
```

Creating an **ArrayList**

- So, to instantiate a generic **ArrayList** to store objects of the base type **String** with an initial capacity of 20 items:

```
ArrayList<String> list = new ArrayList<String>(20);
```

- Note that the base type of an ArrayList is specified as a *type parameter*

ArrayList code Example

```
// Note the use of Integer, rather than int
public static void main( String[ ] args)
{
    ArrayList<Integer> myInts = new ArrayList<Integer>(25);
    System.out.println( "Size of myInts = " + myInts.size());
    for (int k = 0; k < 10; k++)
        myInts.add(3 * k);
    myInts.set(6, 44);
    System.out.println("Size of myInts = " + myInts.size());
    for (int k = 0; k < myInts.size(); k++)
        System.out.print(myInts.get(k) + ", " );
}
// output
Size of myInts = 0
Size of myInts = 10
0, 3, 6, 9, 12, 15, 44, 21, 24, 27,
```

Pitfall: A Generic Constructor Name Has No Type Parameter

- Although the class name in a parameterized class definition has a type parameter attached, the type parameter is not used in the heading of the constructor definition

```
public Cell3( )
```
- A constructor can use the type parameter as the type for a parameter of the constructor, but in this case, the angular brackets are not used

```
public Cell3(T prisoner );
```
- However, when a generic class is instantiated, the angular brackets are used

```
Cell3<Thief> tCell = new Cell3<Thief>(new Thief());
```

Pitfall: A Primitive Type Cannot be Plugged in for a Type Parameter

- The type plugged in for a type parameter **must always be a reference type**
 - It cannot be a primitive type such as `int`, `double`, or `char`
 - However, now that Java has automatic boxing, for wrapper classes this is not a big restriction
 - Note: reference types can include arrays

Pitfall: A Type Parameter Cannot Be Used Everywhere a Type Name Can Be Used

- Within the definition of a parameterized class, there are places in the class's methods where an ordinary class name would be allowed, but a type parameter is not allowed
- In particular, the type parameter cannot be used in simple expressions using **new** to create a new object
 - For instance, the type parameter cannot be used as a constructor name or like a constructor:

```
T object = new T ();  
T[] a = new T[10];
```

Pitfall: An Instantiation of a Generic Class Cannot be an Array Base Type

- Arrays such as the following are illegal:

```
Cell3<Thief>[] a = new Cell3<Thief>[10];
```

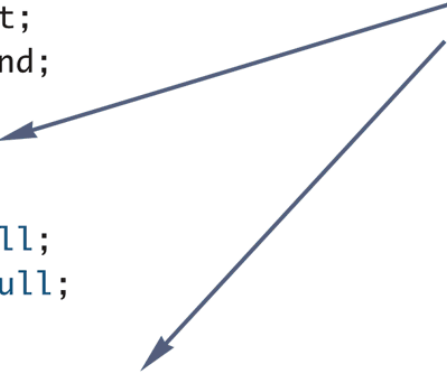
- Although this is a reasonable thing to want to do, it is not allowed given the way that Java implements generic classes
- Use an ArrayList instead

The Commonly Used Generic Ordered Pair Class (1 of 4)

Display 14.5 A Generic Ordered Pair Class

```
1 public class Pair<T>
2 {
3     private T first;
4     private T second;
5     public Pair()
6     {
7         first = null;
8         second = null;
9     }
10    public Pair(T firstItem, T secondItem)
11    {
12        first = firstItem;
13        second = secondItem;
14    }
```

Constructor headings do not include the type parameter in angular brackets.



(continued)

A Generic Ordered Pair Class (2 of 4)

Display 14.5 A Generic Ordered Pair Class

```
15     public void setFirst(T newFirst)
16     {
17         first = newFirst;
18     }

19     public void setSecond(T newSecond)
20     {
21         second = newSecond;
22     }

23     public T getFirst()
24     {
25         return first;
26     }
```

(continued)

A Generic Ordered Pair Class (3 of 4)

Display 14.5 A Generic Ordered Pair Class

```
27     public T getSecond()
28     {
29         return second;
30     }

31     public String toString()
32     {
33         return ( "first: " + first.toString() + "\n"
34                 + "second: " + second.toString() );
35     }
36
```

(continued)

A Generic Ordered Pair Class (4 of 4)

Display 14.5 A Generic Ordered Pair Class

```
37     public boolean equals(Object otherObject)
38     {
39         if (otherObject == null)
40             return false;
41         else if (getClass() != otherObject.getClass())
42             return false;
43         else
44         {
45             Pair<T> otherPair = (Pair<T>)otherObject;
46             return (first.equals(otherPair.first)
47                 && second.equals(otherPair.second));
48         }
49     }
50 }
```

Using Our Ordered Pair Class

Display 14.6 Using Our Ordered Pair Class

```
1  import java.util.Scanner;

2  public class GenericPairDemo
3  {
4      public static void main(String[] args)
5      {
6          Pair<String> secretPair =
7              new Pair<String>("Happy", "Day");
8
9          Scanner keyboard = new Scanner(System.in);
10         System.out.println("Enter two words:");
11         String word1 = keyboard.next();
12         String word2 = keyboard.next();
13         Pair<String> inputPair =
14             new Pair<String>(word1, word2);
```

(continued)

A Class Definition Can Have More Than One Type Parameter

- A generic class definition can have any number of type parameters
 - Multiple type parameters are listed in angular brackets just as in the single type parameter case, but are separated by commas

Multiple Type Parameters (1 of 4)

Display 14.8 Multiple Type Parameters

```
1  public class TwoTypePair<T1, T2>
2  {
3      private T1 first;
4      private T2 second;
5
6      public TwoTypePair()
7      {
8          first = null;
9          second = null;
10
11         public TwoTypePair(T1 firstItem, T2 secondItem)
12         {
13             first = firstItem;
14             second = secondItem;
15         }
16     }
17 }
```

(continued)

Multiple Type Parameters (2 of 4)

Display 14.8 Multiple Type Parameters

```
15     public void setFirst(T1 newFirst)
16     {
17         first = newFirst;
18     }

19     public void setSecond(T2 newSecond)
20     {
21         second = newSecond;
22     }

23     public T1 getFirst()
24     {
25         return first;
26     }
```

(continued)

Multiple Type Parameters (3 of 4)

Display 14.8 Multiple Type Parameters

```
27     public T2 getSecond()
28     {
29         return second;
30     }

31     public String toString()
32     {
33         return ( "first: " + first.toString() + "\n"
34                 + "second: " + second.toString() );
35     }
36
```

(continued)

Multiple Type Parameters (4 of 4)

Display 14.8 Multiple Type Parameters

```
37     public boolean equals(Object otherObject)
38     {
39         if (otherObject == null)
40             return false;
41         else if (getClass() != otherObject.getClass())
42             return false;
43         else
44         {
45             TwoTypePair<T1, T2> otherPair =
46                 (TwoTypePair<T1, T2>)otherObject;
47             return (first.equals(otherPair.first)
48                 && second.equals(otherPair.second));
49         }
50     }
51 }
```

The first equals is the equals of the type T1. The second equals is the equals of the type T2.

Using TwoTypePair (1 of 2)

Display 14.9 Using a Generic Class with Two Type Parameters

```
1  import java.util.Scanner;

2  public class TwoTypePairDemo
3  {
4      public static void main(String[] args)
5      {
6          TwoTypePair<String, Integer> rating =
7              new TwoTypePair<String, Integer>("The Car Guys", 8);

8          Scanner keyboard = new Scanner(System.in);
9          System.out.println(
10             "Our current rating for " + rating.getFirst());
11         System.out.println(" is " + rating.getSecond());

12         System.out.println("How would you rate them?");
13         int score = keyboard.nextInt();
14         rating.setSecond(score);
```

(continued)

Using TwoTypePair (2 of 2)

Display 14.9 Using a Generic Class with Two Type Parameters

```
15      System.out.println(  
16          "Our new rating for " + rating.getFirst());  
17      System.out.println(" is " + rating.getSecond());  
18  }  
19 }
```

SAMPLE DIALOGUE

Our current rating for The Car Guys

is 8

How would you rate them?

10

Our new rating for The Car Guys

is 10

Bounds for Type Parameters

- Sometimes it makes sense to restrict the possible types that can be plugged in for a type parameter **T**:
 - This is called “placing bounds on a type parameter”
- A bound on a type may be a class name
- With class bounds, only the bounding class or any of its descendent classes may be plugged in for the type parameters

```
public class ExClass<T extends Class1>
```

Generics and Hierarchies

- What if we want a somewhat specialized container that assumes the objects it holds are part of a hierarchy so that the container code can assume the existence of a particular method? Let's look at Animals, Dogs, and Cats

```
class Animal { public void speak(){...} ... }  
class Dog extends Animal {...}  
class Cat extends Animal {...}
```

We would like to create a container named Zoo to hold some animals that speak.

Zoo<T>

- If we define the Zoo like this

```
public class Zoo< T >
```

we'll get a compiler error when we try to invoke the `speak()` method. Not all classes provide a method called `speak()`. Only `Animals` provide `speak()`.

The solution is to place a bounds on `T`. The Zoo can only contain `Animals` or any type that inherits from `Animal`.

```
public class Zoo<T extends Animal>
```

The phrase `T extends Animal` means "Animal or any subtype of Animal"

Interface Bounds for Type Parameters

- Other times, it makes sense to restrict the possible types that can be plugged in for a type parameter **T** to a particular interface
 - For instance, to ensure that only classes that implement the **Comparable** interface are plugged in for **T**, define a class as follows:

```
public class RClass<T extends Comparable<T>>
```
 - "**extends Comparable<T>**" serves as a *bound* on the type parameter **T**.
 - Any attempt to plug in a type for **T** which does not implement the **Comparable** interface will result in a compiler error message

Bounded Example

```
// Pair container, but only for those classes that  
// implement the Comparable interface  
public class Pair<T extends Comparable<T> >  
{  
    private T first;  
    private T second;  
  
    public T max( )  
    {  
        if (first.compareTo(second) <= 0)  
            return second;  
        else return first;  
    }  
    // remaining Pair<T> code  
}
```

Generic Sorting

We can now implement sorting functions that can be used for any class (that implements Comparable). The familiar bubble sort is shown below.

```
public class Sort
{
    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;
                }
    }
}
```

Syntax Rules on Type Bounds

- You may specify more than one bounds on a type parameter
- A bounds expression may contain multiple interfaces, but only one class
- If you have both class and interface bounds, the class *must* come first in the list

Generics and Hierarchy

Let's revisit the Animals hierarchy:

- Each animal has a weight and a name. Let's say two Dogs (or two Cats) are the equal if they have the same name and weight.

```
class Animal { private String name; private int weight; ... }  
class Dog extends Animal implements Comparable<Dog>{ ... }  
class Cat extends Animal implements Comparable<Cat>{ ... }
```

- Since Dog implements comparable<Dog> it's clear you can compare Dogs with Dogs, but not with Cats
- We can use our bubbleSort method with Dogs or with Cats

Sorting Dogs

```
public class DogSort
{
    public static void main( String[ ] args )
    {
        // create an array of Dogs
        Dog[ ] dogs = new Dog[ 42 ];

        // put some dogs in the array
        // .....

        // sort the dogs
        Sort.bubbleSort( dogs );

        // rest of main
    }
}
```

Generics and Hierarchies

What happens if we want to sort a class in an inheritance hierarchy, but some ancestor of the class implements comparable, and not the class itself?

But suppose we wanted compare all Animals using only their weight. The class definitions would look something like this

```
class Animal implements Comparable<Animal> { ... }  
class Dog extends Animal { ... }  
class Cat extends Animal { ... }
```

Since Animal implements comparable, any two Animals can be compared (albeit only by weight).

The problem is now that we can't use bubbleSort to sort an array of Dogs because Dog doesn't explicitly implement Comparable (it's inherited from Animal)

New bubbleSort

The solution is to use a “wildcard” when defining bubbleSort

```
public class Sort
{
    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;
                }
    }
}
```

? **super T** is read as “**any supertype of T**”. Now, because Dog extends Animal which implements Comparable, bubbleSort can be used with an array of Dogs as before.

Pitfall: A Generic Class Cannot Be an Exception Class

- It is not permitted to create a generic class with **Exception**, **Error**, **Throwable**, or any descendent class of **Throwable**
 - A generic class cannot be created whose objects are throwable
`public class GEx<T> extends Exception`
 - The above example will generate a compiler error message

Tip: Generic Interfaces

- An interface can have one or more type parameters
- The details and notation are the same as they are for classes with type parameters

Generic Methods

- When a generic class is defined, the type parameter can be used in the definitions of the methods for that generic class
- In addition, a generic method can be defined that has its own type parameter that is not the type parameter of any class
 - A generic method can be a member of an ordinary class or a member of a generic class that has some other type parameter
 - The type parameter of a generic method is local to that method, not to the class

Generic Methods

- The type parameter must be placed (in angular brackets) after all the modifiers, and before the returned type

```
public class Utility {  
    ...  
    public static <T> T getMidPoint( T[ ] array)  
        { return array[array.length / 2]; }  
  
    public static <T> T getFirst( T[ ] a )  
        { return a[0]; }  
    ...  
}
```

Generic Methods

- When one of these generic methods is invoked, the method name is prefaced with the type to be plugged in, enclosed in angular brackets

```
String s =  
    Utility.<String>getMidPoint(arrayOfStrings) ;  
double first =  
    Utility.<Double>getFirst(arrayOfDoubles) ;
```

Inheritance with Generic Classes

- A generic class can be defined as a derived class of an ordinary class or of another generic class
 - As in ordinary classes, an object of the subclass type would also be of the superclass type
- Given two classes: **A** and **B**, and given **G**: a generic class, there is no relationship between **G<A>** and **G**
 - This is true regardless of the relationship between class **A** and **B**, e.g., if class **B** is a subclass of class **A**

A Derived Generic Class (1 of 2)

In this example `UnorderedPair` overrides `equals()` that was inherited from `Pair`

Display 14.11 A Derived Generic Class

```
1 public class UnorderedPair<T> extends Pair<T>
2 {
3     public UnorderedPair()
4     {
5         setFirst(null);
6         setSecond(null);
7     }
8
9     public UnorderedPair(T firstItem, T secondItem)
10    {
11        setFirst(firstItem);
12        setSecond(secondItem);
13    }
```

(continued)

A Derived Generic Class (Part 2 of 2)

Display 14.11 A Derived Generic Class

```
13     public boolean equals(Object otherObject)
14     {
15         if (otherObject == null)
16             return false;
17         else if (getClass() != otherObject.getClass())
18             return false;
19         else
20         {
21             UnorderedPair<T> otherPair =
22                 (UnorderedPair<T>)otherObject;
23             return (getFirst().equals(otherPair.getFirst())
24                 && getSecond().equals(otherPair.getSecond()))
25                 ||
26                 (getFirst().equals(otherPair.getSecond())
27                 && getSecond().equals(otherPair.getFirst()));
28         }
29     }
30 }
```

Using UnorderedPair (Part 1 of 2)

Display 14.12 Using UnorderedPair

```
1 public class UnorderedPairDemo
2 {
3     public static void main(String[] args)
4     {
5         UnorderedPair<String> p1 =
6             new UnorderedPair<String>("peanuts", "beer");
7         UnorderedPair<String> p2 =
8             new UnorderedPair<String>("beer", "peanuts");
```

(continued)

Using UnorderedPair (Part 2 of 2)

Display 14.12 Using UnorderedPair

```
9         if (p1.equals(p2))
10        {
11            System.out.println(p1.getFirst() + " and " +
12                               p1.getSecond() + " is the same as");
13            System.out.println(p2.getFirst() + " and "
14                               + p2.getSecond());
15        }
16    }
17 }
```

SAMPLE DIALOGUE²

```
peanuts and beer is the same as
beer and peanuts
```