CMSC 202

Generics I

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

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 SpeficicBox used to hold a String.

```
public class SpecificBox {
   private String item;

   public SpecificBox(String s){
      item = s;
   }

   public String getItem(){
      return item;
   }
}
```

- This Box is limited to only holding String objects. It is very specific in its uses.

A More General Box

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

```
public class ObjectBox {
  private Object item;

  public ObjectBox(Object o){
    item = o;
  }

  public Object getObject(){
    return item;
  }
}
```

- But this approach can lead to some interesting code and run time exceptions.

Object Box Example

```
public static void main(String[] args){
   ObjectBox box1 = new ObjectBox(new String("HI"));

   // downcast a String to an Integer?
   Integer i = (Integer)box1.getObject();
}
```

- Object is the base class of all classes in Java.
  - Using an Object reference variable will lead to a number of run time exceptions.
  - Special case code would have to be made for every type of derived object that was put in the Box.

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String
cannot be cast to java.lang.Integer
at Generics.ObjectBox.main(ObjectBox.java:17)
```

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 parametrized 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 Box

```
public class GenericBox<Type> {
   private Type item;
   public GenericBox(Type item){
      this.item = item;
   }
   public Type getItem(){
      return item;
   }
   public void setItem(Type newItem){
      this.item = newItem;
   }
}
```

- A class definition with a type parameter is stored in a file and compiled just like any other class
- Once a parametrized 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 Box Example

```
public static void main(String[] args){
   GenericBox<String> box1 = new GenericBox<String>("Charlie Sheen");
   GenericBox box2 = new GenericBox<Integer>(new Integer(2));
   String thingInTheContainer = box1.getItem();

   // compile errors when we try to use Integers with a String Box
   Integer thingInTheContatiner2 = box1.getItem();
   box1.setItem(new Integer(2));
}
```

- Declaring an reference variable to a Generic Object requires you specify the Type

- The Type that is specified does provide syntax checking to make sure you are not trying to insert an Integer into a Box that was meant for Strings.

Pitfall: A Generic Constructor Name Has No Type Parameter

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

```
public GenericBox( )
```

- 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 GenericBox(T item );
```

- However, when a generic class is instantiated, the angular brackets are used

```
GenericBox<String> box1 = new GenericBox<String>("Charlie Sheen");`
```

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 parametrized class definition, there are places in the generic class' 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

```
GenericBox<Integer>[] array = new GenericBox<Integer>[5];
```

- Arrays such as the following are illegal:
  - 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

```
ArrayList<GenericBox<Integer>> arraylist;
arraylist = new ArrayList<GenericBox<Integer>>(5);
```

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

Multi-Type Generic Objects

```
public class MultiType<Type1, Type2> {
   private Type1 item1;
   private Type2 item2;

   public MultiType(Type1 i1, Type2 i2){
      item1 = i1;
      item2 = i2;
   }

   public static void main(String[] args){
      MultiType<String, Integer> container1 =
            new MultiType<String, Integer>("Johnny", 5);
      MultiType<String, String> container2 =
            new MultiType<String, String>("Johnny", "Five");
   }
}
```

- All the rules about parametrized types are still enforced with generic classes that have multiple parametrized types.

Invoking Methods of Typeless Variables

```
public class GenericBox<Type> {
  private T item;

  public void doSomething(){
    item.function();
    SomeType tmp = item.publicVariable;
    System.out.println("Item: " + item);
  }
```

- What interface does Type provide?
  - Java can not know what the interface of Type is during compile time. This means we can not invoke specific methods on variables of type Type.
  - Variables of Type are limited to the interface of Object because all classes are derived from object.
    - All objects can invoke the toString() method even though they did not define it in the class.

Ordering Generic Boxes

```
public class GenericBox<Type> implements Comparable<GenericBox<Type>> {
  private Type item;

  public  GenericBox(Type item) {
     this.item  = item;
  }

  public int compareTo(GenericBox<Type> other){
    return this.item.compareTo(other.item);
  }
  public static void main(String[] args){
    GenericBox<String> box1 = new GenericBox<String>("Derp");
    GenericBox<String> box2 = new GenericBox<String>("Herp");
    box1.compareTo(box2);
  }
}
```

- Suppose we want to implement compareTo() for GenericBox.
  - A syntax error will appear when we attempt to invoke the compareTo() method of an object of type Type.
    - Java can only assume that Type is an Object!!!

Bounds for Type Parameters

- Sometimes it makes sense to restrict the possible types that can be plugged in for a type parameter **`T`**
  - 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

Bounding GenericBox Example

```
public class GenericBox<Type extends Comparable<Type>>  implements
Comparable<GenericBox<Type>> {
   private Type item;
   public  GenericBox(Type item) {
      this.item  = item;
   }
   public int compareTo(GenericBox<Type> other){
      return this.item.compareTo(other.item);
   }
   public static void main(String[] args){
      GenericBox<String> box1 = new GenericBox<String>("Derp");
      GenericBox<String> box2 = new GenericBox<String> ("Herp");
      box1.compareTo(box2);
   }
}
```

- We have to bound Type to extends Comparable<Type> so that in GenericBox's compareTo() method we are able to invoke compareTo() on item.
  - Java will require an object in this container to be a descendant of Comparable. (Implementing comparable).

Bounds for Type Parameters

- A bound on a type may be a class name
- Then only descendent classes of the bounding class may be plugged in for the type parameters

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

- A bounds expression may contain multiple interfaces and up to one class