Java Generics

**Overview**

The Java collections framework makes ample use of generics to provide flexibility to its data structures. In a strongly typed language, such as Java, the values you assign to a variable are first checked to see whether they match the type of a variable (otherwise your code will not compile).

**Reasoning (Why Generics?)**

As an example, the two (very simple) data structures below are limited to what types of values we can store inside:

public class IntegerStorage   
{  
    private int data;  
  
    public IntegerStorage(int data)  
    {  
        this.data = data;  
    }  
  
    public int getData()  
    {  
        return data;  
    }  
}  
  
public class StringStorage   
{  
    private String data;  
  
    public StringStorage(String data)  
    {  
        this.data = data;  
    }  
  
    public String getData()  
    {  
        return data;  
    }  
}

The following code segment will create a few of these objects successfully:

IntegerStorage intStorage = new IntegerStorage(10);  
StringStorage stringStorage = new StringStorage("Hello!");

This code segment will not:

IntegerStorage intStorage = new IntegerStorage(19.99);  
StringStorage stringStorage = new StringStorage(new Car("Lamborghini"));

It would be ideal to have a storage class here that can store lots of different data types. This can be accomplished using upcasting, where we store a object from a child class in variable whose type is a parent (or ancestor) type. The variable type is typically the Object class, which allows any type of object to be stored in the variable. Using this reasoning, we might design a class like the following:

public class ObjectStorage   
{  
    private Object data;  
  
    public ObjectStorage(Object data)  
    {  
        this.data = data;  
    }  
  
    public Object getData()  
    {  
        return data;  
    }  
}

This class solves the first problem we have discovered. We can avoid having to create a new class for each type we would like to store in a class. Instead we can use a single class to store any type of object (we'll worry about primitive values later...). For example, we reduce our redundant code by using a single class now:

ObjectStorage objStorage1 = new ObjectStorage(new Car("Lamborghini"));  
ObjectStorage objStorage2 = new ObjectStorage("Hello!");

There is still a  problem in our design though. If we retrieve our data value from the objects above using getData() our return value is Object. This means that we can only assign these to an Object variable, which reduces the methods we can call on the variable.

For example, the following compiles:

System.out.println(objStorage1.getData().toString());  
System.out.println(objStorage1.equals(objStorage2.getData()));

*Note: Both toString() and equals() are defined in the Object class.*

But this will not:

System.out.println(objStorage2.getData().toUpperCase());

*Note: toUpperCase() is defined in the String class.*

Because our return type is Object we are limited (greatly) in how we can interact with the objects stored in the storage class. This limitation is in place, even though we have interesting objects (strings and cars) stored in those variables. To work around this limitation, developers have historically used casting to convert the data type back to the underlying object type. For example:

String value = (String)objStorage2.getData();  
System.out.println(value.toUpperCase());

There are several drawbacks to this approach:

* Casting is error prone.
  + There is a ClassCastException class that is thrown when you mistakenly cast an object to an incorrect type.
* Casting clutters your code.

*Wouldn't it be nice* if we could have a class that is not redundant (ie. we only define the class once for all objects) and we are able to retrieve data values using the same type we stored inside the class?

**Creating a Generic Class**

Generic types are used to satisfy both of the goals above. To create a generic class, we first declare a generic type identifier at the top of a class.

public DataStorage<T>  
{  
    //more here...  
}

T represents the type used in the class. You can replace T with any valid identifier, so you could alter the class above:

public DataStorage<MyType>  
{  
    //more here...  
}

There are naming conventions associated with the type and it is suggested that you always use a single letter to identify your types. For example, the letters: T, K, V, E are all popular choices.

After declaring your generic type, you can then replace a type in your class with the generic type:

public class DataStorage<T>  
{  
    private T data;  
  
    public DataStorage(T data)  
    {  
        this.data = data;  
    }  
  
    public T getData()  
    {  
        return data;  
    }  
}

**Instantiating Generic Classes**

To use the generic class above, you must specify what type of data value you want to store inside a DataStorage<T> object. For example:

DataStorage<Integer> data1 = new DataStorage<Integer>(20);  
DataStorage<String> data2 = new DataStorage<String>("Hi!");

Notice how I'm storing a primitive type above. Each primitive type has an associated wrapper class that represents the primitive type when an object is required. When using generics, your declared generic type must always be an object. ie. the following will not work:

DataStorage<int> data1 = new DataStorage<int>(20);

Here are the eight wrapper classes associated with each primitive type:

|  |  |
| --- | --- |
| **Primitive** | **Wrapper Class** |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| boolean | Boolean |
| char | Character |

**Type Erasure**

When the compiler evaluates the two DataStorage declarations above, it will convert the generic DataStorage classes into two non-generic classes (with meaningless names), by replaced the generic type T in each class with the specified type given during instantiation. This process is called **Type Erasure** and is denoted below:

|  |  |
| --- | --- |
| **Declared Type** | **Hidden Class Created By Compiler** |
| DataStorage<Integer> | public class ClassOne {     private Integer data;      public ClassOne(Integer data)     {         this.data = data;     }      public Integer getData()     {         return data;     } } |
| DataStorage<String> | public class ClassTwo {     private String data;      public ClassTwo(String data)     {         this.data = data;     }      public String getData()     {         return data;     } } |

*Note: the return types given by each getData() method match the type T!*  
*Note: we only wrote the code for one class (DataStorage). The compiler generates other classes based on how you instantiate your generic class.*

This solves both our problems declared above! Therefore, using this generic class we can write code segments like the following:

DataStorage<Integer> data1 = new DataStorage<Integer>(20);  
DataStorage<String> data2 = new DataStorage<String>("Hi!");  
  
int storedInt = data1.getData();  
System.out.println(storedInt);  
System.out.println(data2.getData().toUpperCase()); //no casting needed!

**Bounds on Generic Types**

So how are we able to interact with a generic type T in a generic class? Should the following compile?

public class DataStorage<T>  
{  
    private T data;  
  
    //constructors, methods...  
  
    public void doSomething()  
    {  
        System.out.println(data.toString());  
        System.out.println(data.foo()); //compiler error!  
    }  
}

What we must ask ourselves is "what do we know about the type T?" The way our class is declared, we only know that type T is an reference type (a class). Therefore, every class has Object as an ancestor class and we know we can call toString() on data. We do not know whether the type T has a foo() method, so the second method invocation will fail to compile.

We can put bounds on our generic type, by declaring that the type uses an interface or has a certain parent or child class. Both of these techniques tell use something about the type T. For example:

public class DataStorage<T **extends Comparable**> //T implements an interface   
{  
    private T data;  
  
    //constructors, methods...  
  
    public void doSomething()  
    {  
        System.out.println(data.toString());  
        System.out.println(data.**compareTo("hello")**); //no compiler error!  
    }  
}

This class header says that T uses the Comparable interface. And we know that the interface includes the compareTo() method, so we now can invoke compareTo() on any variable of type T in our generic class. Similarly:

public class Car  
{  
    //fields  
  
    public void drive()  
    {  
        //drives the car...  
    }  
}

public class DataStorage<T **extends Car**> //T extends from a class (parent or ancestor)   
{  
    private T data;  
  
    //constructors, methods...  
  
    public void doSomething()  
    {  
        System.out.println(data.toString());  
        System.out.println(data.**drive()**); //no compiler error!  
    }  
}

We can declare a parent or ancestor class for type T. This means that type T must have Car as a parent or ancestor class. Which means the type T has inherited a drive() method from Car, so we can call this method above.

Lastly, you can declare that a generic type T is a child class or an ancestor class of another class using wild card notation. An example is given below and in your book, but I would suggest avoiding this advanced notation until you have more experience working with generics:

//MoveableObject is an ancestor of Car  
DataStorage<? super Car> car1 = new DataStorage<MoveableObject>(new MoveableObject());   
//Honda is a descendant of Car  
DataStorage<? extends Car> car2 = new DataStorage<Honda>(new Honda());

**Multiple Generic Types**

You can declare more than one generic type for a class. For example:

public class Pair<K, V>  
{  
    private K first;  
    private V second;  
  
    public Pair(K first, V second)  
    {  
        this.first = first;  
        this.second = second;  
    }  
  
    public K getFirst()  
    {  
        return first;  
    }  
  
    public V getSecond()  
    {  
        return second;  
    }  
}

Below you can see an example of how you might use this class:

public class TestPair  
{  
    public static void main(String[] args)  
    {  
        Pair<String, Integer> nameToAge = new Pair<String, Integer>("Stacey", 25);  
  
        System.out.println(nameToAge.getFirst() + " is " + nameToAge.getSecond() + " years old!";  
    }  
}

Note: You can put bounds on any of the types of a generic class. So you might have a class that looks like the following:

public class Pair<K extends Comparable, V extends MyClass>  
{  
    //do something  
}

**Restricted Use of Generics**

There are certain restrictions on the use of a generic type. Most of these restrictions are due to how type erasure works. For example, is there anything wrong with the following class definition?

public class DataStorage<T>  
{  
    private T data;  
  
    //constructors, methods...  
  
    public void doSomething()  
    {  
        data = new T(); //instantiate our generic type? compiler error!  
    }  
}

This code will not compile because we do not know what the constructor looks like for type T. Recall that a type T can represent a whole range of classes, and a constructor is typically unique to the class. In other words, we cannot use an interface to guarantee that a certain constructor exists in a class. Therefore this technique will not work!

Another restriction is that Java does not allow you to (easily) instantiate generic arrays. This is a restriction you will probably run into when building data structures. For example:

public class DataStorage<T>  
{  
    private T[] data;  
  
    //constructors, methods...  
  
    public void doSomething()  
    {  
        data = new T[10]; //instantiate a generic array? compiler error!  
    }  
}

**Diamond Notation**

In Java 7, diamond notation was released. This notation allows the often verbose lines of code to instantiate generic types to be shortened. For example, the following code works fine with Java 7+.

DataStorage<Integer> data1 = new DataStorage<>(20); //type on the right is inferred from the left!