Object Oriented Programming

Generics

Why generics?

- Person[] people = new Person[25]; // you must say what's in the array
  people[0] = "Sally"; // syntax error

- ArrayList people = new ArrayList(); // but *anything* could go in the ArrayList!
  people.add("Sally");
  // sometime later...
  Person p = (Person)people.get(0); // runtime error

- ArrayList<Person> people = new ArrayList<Person>(); // say what's in it
  people.add("Sally"); // syntax error

- Since Java 5, collections should be used only with generics

Generics

- A generic is a method that is recompiled with different types as the need arises
- The bad news:
  - Instead of saying: `List words = new ArrayList();`
  - You'll have to say:
    ```
    List<String> words = new ArrayList<String>();
    ```
- The good news:
  - Replaces runtime type checks with compile-time checks
  - No casting; instead of
    ```
    String title = (String) words.get(i);
    ```
    you use
    ```
    String title = words.get(i);
    ```
- Some classes and interfaces that have been "genericized" are: Vector, ArrayList, LinkedList, Hashtable, HashMap, Stack, Queue, PriorityQueue, Dictionary, TreeMap and TreeSet

Genericized types are still types

- `ArrayList myList = new ArrayList();`
- `ArrayList<String> myList = new ArrayList<String>();`

↑ this is the type ↑ this is the type again

- You can use generic types as method parameters:
  `String findLongest(ArrayList<String> myList) { ... }`
  - But you ***don't*** mention types when you call a method:
    `String longestString = findLongest(myList);`
- You can return a generic type from a method:
  `ArrayList<String> readList() { ... }`

Generic Iterators

- To iterate over generic collections, it's a good idea to use a generic iterator
  - List<String> listOfStrings = new LinkedList<String>();

```
...
for (Iterator<String> i =
listOfStrings.iterator(); i.hasNext(); ) {
    String s = i.next();
    System.out.println(s);
}
```

Type wildcards

- Here's a simple (no generics) method to print out any list:
  - ```
    private void printList(List list) {
        for (Iterator i = list.iterator(); i.hasNext(); ) {
            System.out.println(i.next());
        }
    }
    ```
- The above still works in Java, but now it generates warning messages
- You should eliminate *all* errors and warnings in your final code, so you need to *tell* Java that any type is acceptable:
  - ```
    private void printListOfStrings(List<?> list) {
        for (Iterator<?> i = list.iterator(); i.hasNext(); ) {
            System.out.println(i.next());
        }
    }
    ```

Creating a ArrayList the new way

- Specify, in angle brackets after the name, the type of object that the class will hold
- Examples:
  - `ArrayList<String> vec1 = new ArrayList<String>();`
  - `ArrayList<String> vec2 = new ArrayList<String>(10);`
- To get the old behavior, but without the warning messages, use the <?> wildcard
  - Example: `ArrayList<?> vec1 = new ArrayList<?>();`

Accessing with and without generics

- Object get(int ***index***)
  - Returns the component at position ***index***

- Using get the old way:
  - ```
    ArrayList myList = new ArrayList();
    myList.add("Some string");
    String s = (String)myList.get(0);
    ```

- Using get the new way:
  - ```
    ArrayList<String> myList = new ArrayList<String>();
    myList.add("Some string");
    String s = myList.get(0);
    ```

- Notice that casting is no longer necessary when we retrieve an element from a “genericized” ArrayList

Generics and Inheritence

- Suppose you want to restrict the type parameter to express some restriction on the type parameter
- This can be done with a notion of subtypes
- expressed in Java using inheritance
- So it’s a natural combination to combine inheritance with generics
- A few examples follow

Parameterized Classes in Methods

- A parameterized class is a type just like any other class.
- It can be used in method input types and return types.

Parameterized Classes in Methods

- If a class is parameterized, that type parameter can be used for any type declaration in that class, e.g:

```
public class Box<E>
{E data;
public Box(E data) {this.data = data;}

public E getData() {return data;}
public void copyFrom(Box<E>  b)
        {this.data = b.getData();}
```

Bounded Parameterized Types

- Sometimes we want restricted parameterization of classes.
- We want a box, called MathBox that holds only Number objects.
- We can’t use Box<E>because Ecould be anything.
- We want E to be a subclass of Number.

Bounded Parameterized Types

```
public class MathBox<E extends Number> extends
                        Box<Number>
{public MathBox(E data)
    {super(data);
     }
     public double sqrt()
    {return Math.sqrt(getData().doubleValue())
     }
}
```

Bounded Parameterized Types

- The <E extends Number> syntax means that the type parameter of MathBox must be a subclass of the Number class
  - We say that the type parameter is bounded

```
new MathBox<Integer>(5); //Legal
new MathBox<Double>(32.1); //Legal
new MathBox<String>("No good!");//Illegal
```

Bounded Parameterized Types

- Java allows multiple inheritance in the form of implementing multiple interfaces, so multiple bounds may be necessary to specify a type parameter. The following syntax is used then:

  <T extends A & B & C & …>

- **Example**

```
interface A {…}
interface B {…}

class MultiBounds<T extends A & B> {
…
}
```

Implementing generics

```
// a parameterized (generic) class
public class name<Type> {
```

or

```
public class name<Type, Type, ..., Type> {
```

- By putting the **Type** in < >, you are demanding that any client that constructs your object must supply a type parameter.
  - You can require multiple type parameters separated by commas.
- The rest of your class's code can refer to that type by name.
  - The convention is to use a 1-letter name such as:
    T for Type, E for Element, N for Number, K for Key, or V for Value.
- The type parameter is *instantiated* by the client. (e.g. `E` → `String`)

Generics and arrays

```
public class Foo<T> {
    private T myField;                  // ok
    private T[] myArray;                // ok


    public Foo(T param) {
        myField = new T();              // error
        myArray = new T[10];            // error
    }
}
```

– You cannot create objects or arrays of a parameterized type.

Generics/arrays, fixed

```
public class Foo<T> {
    private T myField;                          // ok
    private T[] myArray;                        // ok

public Foo(T param) {
        myField = param;                        // ok
        T[] a2 = (T[]) (new Object[10]); // ok
    }
}
```

– But you can create variables of that type, accept them as parameters, return them, or create arrays by casting `Object[]`.

- Casting to generic types is not type-safe, so it generates a warning.

Generic methods

```
public static <Type> returnType
name(params) {
```

- When you want to make just a single (often static) method generic in a class, precede its return type by type parameter(s).

```
public class Collections {
    ...
    public static <T> void copy(List<T> dst,
  List<T> src) {
        for (T t : src) {
            dst.add(t);
        }
    }
}
```

Bounded type parameters

<**Type** extends **SuperType**>

- An upper bound; accepts the given supertype or any of its subtypes.
- Works for multiple superclass/interfaces with & :

  <**Type** extends **ClassA** & **InterfaceB** & **InterfaceC** & **...**>

<**Type** super **SuperType**>

- A lower bound; accepts the given supertype or any of its supertypes.

- Example:

```
// tree set works for any comparable type
public class TreeSet<T extends Comparable<T>>
  {
     ...
}
```

Complex bounded types

- `public static <T extends Comparable<T>> T max(Collection<T> c)`
  - Find max value in any collection, if the elements can be compared.
- `public static <T> void copy( List<T2 super T> dst, List<T3 extends T> src)`
  - Copy all elements from src to dst. For this to be reasonable, dst must be able to safely store anything that could be in src. This means that all elements of src must be of dst's element type or a subtype.
- `public static <T extends Comparable<T2 super T>> void sort(List<T> list)`
  - Sort any list whose elements can be compared to the same type or a broader type.

Generics and subtyping

- Is `List<String>` a subtype of `List<Object>`?
- Is `Set<Giraffe>` a subtype of `Collection<Animal>`?

- **No**. That would violate the Liskov Substitutability Principle.
  - If we could pass a `Set<Giraffe>` to a method expecting a `Collection<Animal>`, that method could add other animals.

```
Set<Giraffe> set1 = new HashSet<Giraffe>();
Set<Animal>  set2 = set1;                  // illegal
...
set2.add(new Zebra());
Giraffe geoffrey = set1.get(0);    // error
```

Wildcards

- `?` indicates a *wild-card* type parameter, one that can be any type.
  - `List<?> list = new List<?>();  // anything`

- Difference between `List<?>` and `List<Object>` :
  - `?` can become any particular type; `Object` is just one such type.
  - `List<Object>` is restrictive; wouldn't take a `List<String>`

- Difference btwn. `List<Foo>` and `List<? extends Foo>`:
  - The latter binds to a particular `Foo` subtype and allows ONLY that.
    - e.g. List<? extends Animal> might store only Giraffes but not Zebras
  - The former allows anything that is a subtype of `Foo` in the same list.
    - e.g. List<Animal> could store both Giraffes and Zebras

Generics with subclass

```
public double areaOfCollection (Collection<? extends Shape> c)
{
  double sum = 0.0;
  for (Shape s : c)
    sum += s.getArea();
}
```

Generics with Comparator

Comparator interface is also generic

```
public interface Comparator<T> {
  int compare(T o1, T o2);
  boolean equals(Object o);
}
```

Create a comparator CompareByLength to sort Strings by length in x

Generics with Comparator

```
public class CompareByLength implements
Comparator<String> {
  int compare(String o1, String o2)
  {return o1.length() - o2.length();


}
```

Generics with Comparator

– Method that takes an array of objects and a collection and puts all objects in the array into the collection

```
static <T> void fromArrayToCollection(T[] a,
Collection<T> c) {
    for (T o : a) {
        c.add(o);
    }
}
```

Generics and casting

- Casting to generic type results in a warning.

```
List<?> l = new ArrayList<String>();
// ok
List<String> ls = (List<String>) l;
// warn
```

  - The compiler gives an unchecked warning, since this isn't something the runtime system is going to check for you.
  - Usually, if you think you need to do this, you're doing it wrong.

- The same is true of type variables:

```
public static <T> T badCast(T t,
Object o) {
    return (T) o;    // unchecked warning
}
```