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

- Subtypes (weakly construed) can be expressed in Java using inheritence

- So it's a natural combination to combine inheritence with generics

- A few examples follow

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> {
  …
  }
  ```