# 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 E could 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

**<T extends A & B & C & …>**

```
public class TreeSet<T extends Comparable<T>>
  {
      . . .
}
```

# 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[]`.

# Generic methods

```
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** super **SuperType**>

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

```
public static <T> void copy(
    List<T2 super T> dst,
    List<T3 extends T> src  )
```

# 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();}


}
```