CSE 331

Generics (Parametric Polymorphism)

slides created by Marty Stepp
based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer, Wikipedia
http://www.cs.washington.edu/331/

Parametric polymorphism

- **parametric polymorphism**: Ability for a function or type to be written in such a way that it handles values identically without depending on knowledge of their types.
  - Such a function or type is called a *generic* function or data type.
  - first introduced in ML language in 1976
  - now part of many other languages (Haskell, Java C#, Delphi)
  - C++ templates are similar but lack various features/flexibility
- Motivation: Parametric polymorphism allows you to write flexible, general code without sacrificing type safety.
  - Most commonly used in Java with collections.
  - Also used in reflection (*seen later*).

Java collections ≤ v1.4

- The initial Java collections stored values of type `Object`.
  - They could store any type, since all types are subclasses of Object.
  - But you had to cast the results, which was tedious and error-prone.
    - Examining the elements of a collection was not type-safe.

```
// in Java 1.4:
ArrayList names = new ArrayList();
names.add("Marty Stepp");
names.add("Stuart Reges");
String teacher = (String) names.get(0);

// this will compile but crash at runtime; bad
Point oops = (Point) names.get(1);
```

Type Parameters (Generics)

```
List<Type> name = new ArrayList<Type>();
```

- Since Java 1.5, when constructing a `java.util.ArrayList`, we specify the type of elements it will contain between < and >.
  - We say that the `ArrayList` class accepts a **type parameter**, or that it is a **generic** class.
  - Use of the "raw type" `ArrayList` without <> leads to warnings.

```
List<String> names = new ArrayList<String>();
names.add("Marty Stepp");
names.add("Stuart Reges");
String teacher = names.get(0);          // no cast
Point oops = (Point) names.get(1);  // error
```

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 (15.4)

```
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

    @SuppressWarnings("unchecked")
    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.

Comparing generic objects

```
public class ArrayList<E> {
    ...
    public int indexOf(E value) {
        for (int i = 0; i < size; i++) {
        // if (elementData[i] == value) {
            if (elementData[i].equals(value)) {
                return i;
            }
        }
        return -1;
    }
}
```

- When testing objects of type `E` for equality, must use `equals`

A generic interface

```
// Represents a list of values.
public interface List<E> {
    public void add(E value);
    public void add(int index, E value);
    public E get(int index);
    public int indexOf(E value);
    public boolean isEmpty();
    public void remove(int index);
    public void set(int index, E value);
    public int size();
}

public class ArrayList<E> implements List<E> { …
public class LinkedList<E> implements List<E> { …
```

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 = set2;              // 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

Type erasure

- All generics types become type `Object` once compiled.
  - One reason: Backward compatibility with old byte code.
  - At runtime, all generic instantiations have the same type.

```
List<String>  lst1 = new ArrayList<String>();
List<Integer> lst2 = new ArrayList<Integer>();
lst1.getClass() == lst2.getClass()  // true
```

- You cannot use `instanceof` to discover a type parameter.

```
Collection<?> cs = new ArrayList<String>();
if (cs instanceof Collection<String>) {
    // illegal
}
```

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
}
```