Generics and Collections

COMP 2210 – Dr. Hendrix

AUBURN

UNIVERSITY

SAMUEL GINN
COLLEGE OF ENGINEERING

Generic types

Generics allow a **type variable** to be used in place of a specific type name.

A type variable can be used to *parameterize* a class, interface, or method with respect to the types involved.

This allows classes, interfaces, and methods to deal with objects of different types at runtime while maintaining compile-time **type safety**.

```
public __<T>__ int search(___T___[] a, ___T___target)
{
   int i = 0;
   while ((i < a.length) && (_!a[i].equals(target)_))
      i++;
   if (i < a.length)
      return i;
   else
      return -1;
}
```

ORACLE **Tutorial** http://download.oracle.com/javase/tutorial/java/generics/index.html

Generic type search example

```
public static <T> int search(T[] a, T target) {
   int i = 0;
   while ((i < a.length) && (!a[i].equals(target))) {
           i++;
   }
   if (i < a.length)
      return i;
   else
      return -1;
}
```

Assume that this is a static method in a class named SearchLib.

When calling this method, clients will preface the method name with the class name.

In a client …

```
String[] sarray = {"2", "4", "6", "8", "10"};
Integer[] iarray = {2, 4, 6, 8, 10};
Number[] narray = {2, 4, 6, 8, 10};
```

Sample calls:

`SearchLib.<String>search(sarray, “8”)`	✓
`SearchLib.<String>search(sarray, 8)`	✗
`SearchLib.search(sarray, 8)`	✓
`SearchLib.<Integer>search(iarray, 8)`	✓
`SearchLib.<Integer>search(narray, 8)`	✗
`SearchLib.<Number>search(narray, 8)`	✓

Generic types and type safety

```
public class SearchLib {

   public static <T> int search(T[] a, T target) {
      int i = 0;
      while ((i < a.length) && (!a[i].equals(target)))
         i++;
      if (i < a.length)
         return i;
      else
         return -1;
   }
}
```

```
% javac -Xlint:unchecked SearchLib.java
%
```

No unchecked warnings, so this code is type-safe.

Generic types and type safety

```
public class SearchLib {

   public static int search(Object[] a, Object target, Comparator c) {
      int i = 0;
      while ((i < a.length) && (c.compare(a[i], target) != 0))
         i++;
      if (i < a.length)
         return i;
      else
         return -1;
   }
}
```

```
% javac -Xlint:unchecked SearchLib.java
SearchLib.java:5: warning: [unchecked] unchecked call to compare(T,T)
as a member of the raw type Comparator
      while ((i < a.length) && (c.compare(a[i], target) != 0))
                                         ^
  where T is a type-variable:
    T extends Object declared in interface Comparator
1 warning

%
```

An unchecked warning, so this code is not type-safe. *But where is the generic type variable?*

```
public interface Comparator<T> { . . . }
```

Generic types and type safety

```
public class SearchLib {

   public static <MyType> int search(MyType[] a, MyType target, Comparator<MyType> c) {
      int i = 0;
      while ((i < a.length) && (c.compare(a[i], target) != 0))
         i++;
      if (i < a.length)
         return i;
      else
         return -1;
   }
}
```

Binds the Comparator's type variable to MyType.

```
public interface Comparator<T> { . . . }
```

```
% javac -Xlint:unchecked SearchLib.java
%
```

No unchecked warnings, so this code is type-safe.

Note: We could have used T for the type variable of the search method instead of MyType. I just didn't want to make you think that we had to use T because Comparator used T. It's completely up to you what you name the type variables.

Generic types and type safety

Every interface and class in the Java Collections Framework (e.g., Comparator, ArrayList, etc.) have type variables. So, when you use these interfaces and classes, you should bind their type variables.

Example:

```
public static <T> T min(Collection<T> c, Comparator<T> comp) {
   . . .
}
```

This <T> *declares* a type variable named T for the min method.

These <T> notations *bind* the type variables of the Collection and Comparator interfaces to the T type used in the search method.

This is ensures that the min method accept only a Collection of a given type T and a Comparator designed to compare objects of type T. Type-safe!

Bounded type parameters

`<T>` No restrictions on what can be passed in for T.

A **bounded type parameter** allows you to restrict what types can be passed in as the generic type parameter.

`<T extends MyType>`	MyType is an “upper bound” on T.
`<T extends Book>`	*T can be Book or any **subclass** of Book.*
`<T extends Comparable>`	*T must implement Comparable.*

A *named* type parameter can’t be given a lower bound.

Bounded type parameters

```
public static <T extends Number> int search(T[] a, T target) {
   int i = 0;
   while ((i < a.length) && (!a[i].equals(target))) {
           i++;
   }
   if (i < a.length)
      return i;
   else
      return -1;
}
```

In a client …

```
String[] sarray = {"2", "4", "6", "8", "10"};
Integer[] iarray = {2, 4, 6, 8, 10};
Number[] narray = {2, 4, 6, 8, 10};
```

Sample calls:

`SearchLib.search(narray, 8)`	✓
`SearchLib.search(iarray, 8)`	✓
`SearchLib.<Number>search(narray, 8)`	✓
`SearchLib.<Integer>search(iarray, 8)`	✓
`SearchLib.<Integer>search(narray, 8)`	✗
`SearchLib.<String>search(sarray, “8”)`	✗

Wildcards

`?` The wildcard character represents an *unnamed* unknown type.

Wildcards can be given upper bounds **and** lower bounds.

`<? extends MyType>`	MyType is an upper bound on this unknown type
`<? extends Book>`	*An unknown type that is Book or any **subclass** of Book.*
`<? super MyType>`	MyType is a lower bound on this unknown type
`<? super Book>`	*An unknown type that is Book or any **superclass** of Book.*

Wildcards with bounds

Declaration of a named generic type variable for this method.

```
public static <T> boolean contains(
                  Collection <? extends T> collection,
```

collection must store things of type T or some subclass of T

```
                  Comparator <? super T> comp,
```

comp must be able to compare things of type T or any superclass of T

```
                T element)
```

element must be of type T

Generic types and type safety

```
public static int search(Comparable[] a, Comparable target) {
     int i = 0;
     while ((i < a.length) && (a[i].compareTo(target) != 0))
        i++;
     if (i < a.length)
        return i;
     else
        return -1;
  }


  public static <T extends Comparable> int searchAlmostSafe(T[] a, T target) {
     int i = 0;
     while ((i < a.length) && (a[i].compareTo(target) != 0))
        i++;
     if (i < a.length)
        return i;
     else
        return -1;
  }
```

Not type-safe

```
  public static <T extends Comparable<? super T>> int searchSafe(T[] a, T target) {
     int i = 0;
     while ((i < a.length) && (a[i].compareTo(target) != 0))
        i++;
     if (i < a.length)
        return i;
     else
        return -1;
  }
```

Type-safe

Collection v. Collections

`java.util.Collection` ≠ `java.util.Collections`

The root **interface** in the Java collection hierarchy.

A collection represents a group of objects, known as its *elements*

ORACLE **API**

http://bit.ly/onjH0N

ORACLE **Tutorial**

http://bit.ly/n2IZ3Z

A **class** of static methods that operate on type java.util.Collection.

Contains polymorphic algorithms for collections (e.g., searching, sorting, etc.).

ORACLE **API**

http://bit.ly/q4IC6S

ORACLE **Tutorial**

http://bit.ly/9Yrb2K