More on Java Generics

- Multiple Generic Types
- Bounded Generic Types
- Wild Cards
- Raw Types versus Parameterized Types
- Meta-data / Annotations
- Reading: **https://docs.oracle.com/javase/tutorial/java/generics/index.html**

Multiple Generic Types

- Multiple generic types in a class
- Example: **`TreeMap<K, V>`**
- The default **`TreeMap`** constructor assumes **`K implements Comparable`**, but **`K`** is not specified **`K extends Comparable`**
- The overloaded **`TreeMap`** constructor with a ***`Comparator`*** parameter does not assume it

Bounded Generic Types

- Bounded Generic Types

  <T extends *Class*> is a bounded generic type

  T must be Class or some subclass of Class

  <T extends *Interface*> is also a bounded type

  T must be an implementing class of Interface

- Note syntax is using **extends** for either:

  implements (interface) or

  extends (inheritance)

Bounded Generic Types

- Defining a class with a **bounded** type

```
public class Generic<T extends Comparable>
   { … }
```

- Using a class with a ***bounded generic*** type

```
Generic<Comparable> g1 =
                        new Generic<Comparable>();
Generic<String> g2 = new Generic<String>();
Generic<NotComparable> g3 =
        new Generic<NotComparable>(); // error
```

Bounded Generic Types

- Lower Bound Generic Types

  <T super *Class*> is a lower-bound generic type

  T must be Class or some superclass of Class

- Defining a class with a lower bound type

```
public class Generic<T super Stack>
   { … }
```

- Using a class with a lower bound type

```
Generic<Vector> g4 = new Generic<Vector>();
Generic<Stack> g5 = new Generic<Stack>();
```

Bounded Generic Types

- But even though there is a superclass and subclass relationship between the generic types involved, it is not a valid widening conversion for classes parameterized with related generic types

Bounded Generic Types

- Remember a class with a bounded generic type

```
class Generic<T extends Comparable>
Generic<Comparable> g1 = …
Generic<String> g2 = …
```

- However, assignment won't work

```
g1 = g2;    // is a compiler error
            // Generic<String> is not
            // a valid subtype of
            // Generic<Comparable>
```

Wildcards

- However, we can get around some of the preceding restrictions by using *wildcards*
- Wildcard Generic Types

  `<?>` is an extended wildcard

  same as `<? extends Object>`

  `<? extends T>` is a bounded wildcard

  ? must be T or some subclass of T

  `<? super T>` is a lower-bound wildcard

  ? must be T or some superclass of T

Wildcards

- Use as a variable type

  `Integer` is a subtype of `Number`

  `List<Integer>` is not a subtype of `List<Number>`

- However, with a wildcard we can get Integer elements out of a `List<? extends Number>`

```
List<Integer> ints = Arrays.asList(1,2);
List<? extends Number> nums = ints;
for(Number n : nums)
    System.out.println(n);
```

Wildcards

- Use as a type for a method parameter:

```
boolean addAll(Collections<? extends T> c)
```

- Another collection containing a subtype of T can be added to a collection of type T

```
ArrayList<Comparable> c = new . . . ;
ArrayList<String> s = new . . . ;
c.addAll(s);
```

Wildcards

- We can't use a wildcard in the class header where a dummy generic type will need to be used in code

```
public class ClassName<?>
public class ClassName<? extends Number>
```

- How could we write lines of code that refer to the generic type for this ? class?

```
? myQuestionMark = ...   // compiler is ☹
public ? myMethod()      // compiler is ☹
```

Evolution, not Revolution

- An important goal in the generic design was to ensure that Java 4.2 legacy code would work with the Java 5.0 generic library
- Java recognizes the un-parameterized type of each parameterized class/interface in the library, e.g. `Iterable` and `Iterable<T>`
- The parameterized types are subtypes of the corresponding un-parameterized "raw" type used in the legacy code

Evolution, not Revolution

- Legacy code used “raw” un-parameterized types for its reference variables pertaining to the now parameterized types in the Java 5.0 class library

```
ArrayList myList = new ArrayList();
```

- A value of a parameterized type can be passed where a raw type is expected – a normal widening conversion
- A value of a raw type can also be passed where a parameterized type is expected, but the compiler produces an “unchecked” warning

Evolution, not Revolution

- You also get warnings on passing of object references where type <T> is expected
- Use compiler switch `-source 1.4` or add annotations to the legacy code to suppress these warnings:

```
@SuppressWarnings("unchecked")
```

- But if you are editing the legacy source, why not just make it generic instead?

Meta-data/Annotations

- In JDK 5.0, a dedicated annotation facility was added, probably due to success of C#'s attributes
- One of the first practical uses of annotations is a as a way to suppress compiler warnings

```
@SupressWarnings(“type of warning”)
```

- This allows the developer to signal a “respecting” compiler that it should forgo a particular warning
- It is up to the compiler to make sense of whatever you put inside the string, the only value mandated in Java Language Specification is `"unchecked"`

Meta-data/Annotations

- Compilers as well as IDE's implement their own set of warning types, e.g. the Eclipse IDE defines more than NetBeans does
- See a compiler's support with `javac -X`
- Sun JDK1.6.0_03 supports types : cast, deprecation, divzero, empty, unchecked, fallthrough, path, serial, finally, overrides

Meta-data/Annotations

- The following code would get two warnings

```
public void uncheckedTest()
{
   List nonGenericList = new ArrayList();
   nonGenericList.add("Some string");
   List<String> genericStringList =
             (List<String>)nonGenericList;
}
```

```
warning: [unchecked] unchecked call to add(E) as
member of the raw type java.util.List
nonGenericList.add("Some string");
```

```
warning: [unchecked] unchecked cast
  found : java.util.List
  required: java.util.List
List genericStringList = (List)nonGenericList;
```

Meta-data/Annotations

- We can use “unchecked” to avoid warnings on unchecked casts when using generics
- It can be applied in front of a type, a field, a method, a parameter, a constructor as well as a local variable

```
@SuppressWarnings(“unchecked”)
public void uncheckedTest()
{ …
```