UNIVERSITY OF CALGARY
SCHULICH
School of Engineering

ENSF 619

1 - Java Generics

Topics

- What is generic programming
- basics of generic classes
- generic code and JVM
- generic methods and algorithms
- inheritance and generic types
- bounded type parameters
- Limitations of generics
- wildcard types and wildcard type capture

What is Generic Programming

- Background
  - JDK 1.5 introduces several extensions to the Java programming language. One of these is the introduction of *generics*.
  - Generics allow you to abstract over types.
  - The most common examples are container types, such as those in the Collection hierarchy.
    - The class `ArrayList` is a *parameterized:*

```
ArrayList <Integer> myArray = new ArrayList<Integer> (100);
```

Why generic programming

- Lets you write code that is safer and easier to read
- Generic types are a powerful tool to write reusable object-oriented components.
- However, the generic language features are not easy to master and can be misused
  - Their full underestanding requires knowledge of the type theory of programming languages:
    - Covariant
    - *Contravariant*
    - *Invariant*
- It is especially useful for general data structures

Java generics

- in principle, supports *statically-typed* data structures
  - *early detection* of type violations
    - cannot insert a string into ArrayList <Number>
  - also, hides automatically generated casts
- *superficially* resembles C++ templates
  - C++ templates are factories for ordinary classes and functions
    - a new class is always instantiated for given distinct generic parameters (type or other)
- In java the compiled code replaces generic type references with object type references and adds casts where necessary.

How LinkedList Used to be

- Prior to Java 1.5, a typical usage of class LinkedList was:

```
List x = new LinkedList();
x.add(new Integer(0));
Integer y = (Integer) x.getFirst();
```

- What kind of data is allowed into a particular list? Notice the cast operation.
- Compiler only ensures that an Object will be returned by the iterator.
- Casting a variable means this assignment is true for this particular point of code.

What is Generics?

- The core idea is to restrict the list to contain a particular data type

```
List<Integer> x = new LinkedList<Integer>();
x.add(new Integer(0));
Integer y = x.getFirst ();
```

- No more arbitrary List.
- Not more type casting is required.
- When we declare a list with parameterized type such as `<Integer>`, it holds true for the entire program, and whenever.
- A parameterized class is compiled just like any other class.

How to Define of a Generic class

- Here is the general format:

```
class class_name <TYPE> {
      public TYPE X;
      public TYPE Y;
        …
      public class_name (TYPE f, TYPE s)
     {
            // some code
     }
    …

    public TYPE fucntion_name()
    {
         // some code
    }
}
```

Generic code and the JVM

- in Java, generic types are *compile-time* entities.
  - A generic type definition is compiled once.
  - A corresponding *raw type* is produced.
  - the name of the raw type is the same name but type parameters are removed.
    - This action is called type erasure.
- in C++, *instantiations* of a class template are *compiled separately* as source code, and *tailored code* is produced for each one

Generic code and the JVM (cont.)

- *Pair* `<String>` and *Pair* `<Employee>` use the same bytecode generated as the raw class *Pair*
- when translating generic expressions, such as

```
Pair <Employee> buddies = new Pair <Employee>();
Employee buddy = buddies.first;
```

- the compiler uses the raw class and automatically inserts a cast from *Object* to *Employee*:

```
Employee buddy = (Employee)buddies.first;
```

Example

```
public class Sample<T>
{
    private T data;

    public void setData(T newData)
    {
        data = newData;
    }

    public T getData()
    {
        return data;
    }
}
```

- `Sample <String>  foo = new Sample<String> ("XYZ");`
- `String s = foo.getData();`

Java Generic Limitations

Generic Type Limitations

- The type plugged in for a type parameter cannot be primitive types such as `int, double, char…`
- Instantiation of arrays such as the following are illegal:

```
Pair<String>[] a = new Pair<String>[10];
```

- Instantiating an object of a parameterized type is impossible because instantiation requires a call to a constructor, which is unavailable if the type is unknown. For example, the following code will not compile:

```
T instantiateElementType(List<T> arg) {
  return new T(); //causes a compile error
}
```

- static fields are not allowed

```
class Singleton <T> {
        private static T singleOne; // ERROR
}
```

  - Because after type erasure, one class and one shared static field for all instantiations and their objects

Multiple Type Parameters

- A generic class definition can have any number of type parameters.
  - Multiple type parameters are listed in angular brackets just as in the single type parameter case, but are separated by commas.
- you can have multiple type parameters

```
class Pair <T, U> {

    public T first;
    public U second;
    public Pair (T x, U y) { first = x; second = y; }
    public Pair () { first = null; second = null; }
}
```

- to instantiate: `Pair <String, Number>`

A Generic Classes and Exceptions

- It is not permitted to create a generic class with `Exception`, `Error`, `Throwable`, or any descendent class of `Throwable`
  - The following code is not allowed:

```
public class GEx<T> extends Exception {}
```

  - The above example will generate a compiler error message

Bounds for Type Parameters

- Sometimes it makes sense to restrict the possible types that can be plugged in for a type parameter `T`.
  - For instance, to ensure that only classes that implement the **`Comparable`** interface are plugged in for **`T`**, define a class as follows:

```
public class RClass<T extends Comparable>
```

  - "**`extends Comparable`**" serves as a *bound* on the type parameter **`T`**.
  - Any attempt to plug in a type for **`T`** which does not implement the **`Comparable`** interface will result in a compiler error message.

Example

```
public class Pair<T extends Comparable>
{
    private T first;
    private T second;

    public T max()
    {
        if (first.compareTo(second) <= 0)
            return first;
        else
            return second;
    }
}
```

Bounds for Type Parameters

- A bound on a type may be a class name (rather than an interface name)
  - Then only descendent classes of the bounding class may be plugged in for the type parameters:

```
public class ExClass<T extends Class1>
```

- A bounds expression may contain multiple interfaces and up to one class.

- If there is more than one type parameter, the syntax is as follows:

```
public class Two<T1 extends Class1, T2 extends Class2 & Comparable>
```

Generic Interfaces

- An interface can have one or more type parameters.
- The details and notation are the same as they are for classes with type parameters.

Generic Methods

- When a generic class is defined, the type parameter can be used in the definitions of the methods for that generic class.

- In addition, a generic method can be defined that has its own type parameter that is not the type parameter of any class

  - A generic method can be a member of an ordinary class or a member of a generic class that has some other type parameter.

  - The type parameter of a generic method is local to that method, not to the class.

Generic Methods (Cont'd)

- The type parameter must be placed (in angular brackets) after all the modifiers, and before the return type:

```
class NonGeneric {
    public static <T> T genMethod() {
        ...
    }
}
```

- When one of these generic methods is invoked, the method name is prefaced with the type to be plugged in, enclosed in angular brackets:

```
String s = NonGeneric.<String> genMethod();
```

- You can define generic methods both inside ordinary classes and inside generic classes

Inheritance with Generic Classes

- A generic class can be defined as a derived class of an ordinary class or of another generic class
  - In the case of ordinary classes, an object of the subclass type would also be of the superclass type

- Given two classes: `A` and `B`, and given `G`: a generic class, there is no relationship between `G<A>` and `G<B>`
  - This is true regardless of the relationship between class **`A`** and **`B`** (e.g. if class **`B`** is a subclass of class **`A`**)

Wildcards in generics

- You can define a function as:

```
void f(Set<?> set) { ... }
```

  - What does that mean?

    The Set passed to f might be a `Set<Object>`, or a `Set<String>`, or a `Set<Set<Integer>>`, or …

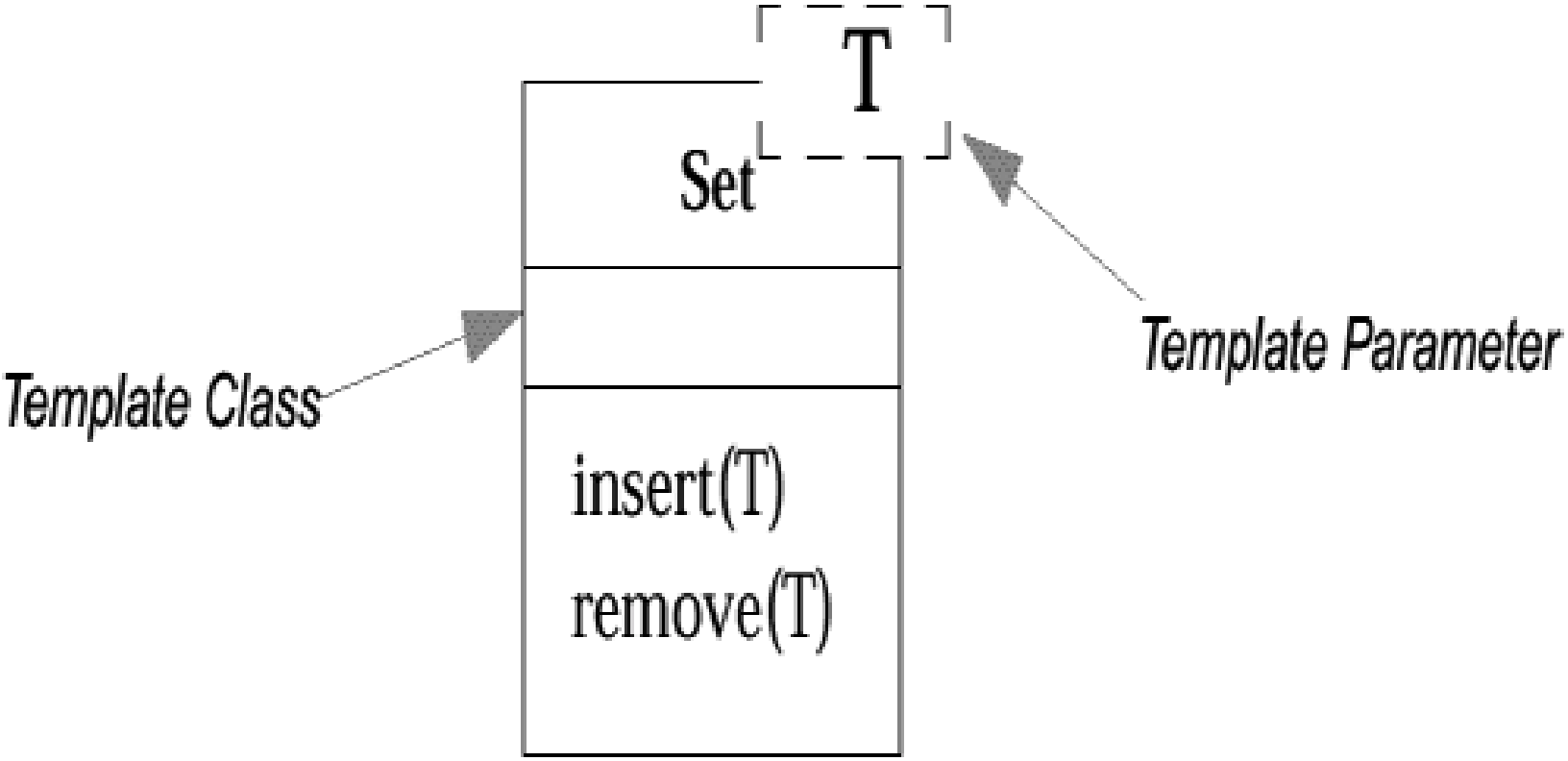
  - The only thing you know about it is that what you take out is some subtype of Object
  - Can you add a value to set?
  - Alternatively, you could provide a generic method

```
public static <T> f(Set <T> p) { …}
```

- Wildcard type (?) cannot be used as a declared type of a variable:

```
Pair <?> p = new Pair <String> ("one","two"); . .
p.first = p.second; // ERROR: unknown type
```

UML Notation for Parameterized Classes



Tip: Compile with the `-Xlint` Option

- There are many pitfalls that can be encountered when using type parameters
- Compiling with the `-Xlint` option will provide more informative diagnostics of any problems or potential problems in the code

```
javac -Xlint Sample.java
```