School of Electrical Engineering and Computing

SENG2200/6220 PROGRAMMING LANGUAGES & PARADIGMS
(S1, 2020)

Java Generics

Dr Nan Li
Office: ES222
Nan.Li@newcastle.edu.au

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

Outline

- Java Generics
  - *Introduction*
  - *Generic class/method*
  - *Wildcards*
  - *Bounded Wildcards*
  - *Type Erasure*
- C++ Templates
- Comparable Interface

What are Generics

- Enhancement to the Java language introduced in Java 1.5
- Similar concept to C++ templates
- Allows you to abstract over a type
- Provides additional compile time checks
- Increased type safety
- Used primarily with container type objects

Without Generics – Example 1

```
List myList = new LinkedList();
myList.add(new Integer(10));
Integer n = (Integer)myList.getFirst();
```

- myList can store any Object
- Explicit cast must be made when retrieving an item from the list, so you need to know the actual type of the Object that is being accessed in the list so that a correct reference can be set.

Without Generics – Example 2

```
List yourList = new LinkedList();
yourList.add(new Student(“John”,”Doe”));
Student jd = (Student)yourList.getFirst();
```

- yourList can also store any Object
- Explicit cast must once again be made when retrieving an item from the list
- Notice that, once again, the programmer must still know the (actual) type of the Object being accessed within the list.

Without Generics

- Problems include
  - *Possibility of runtime exception when non-Integer objects are inserted into the list*

```
myList.add(new Student("John","Doe"));
```

will still be allowed by the compiler, with the error not being discovered until Object is retrieved using Example 1 line of code

```
Integer n = (Integer)myList.getFirst();
```

  - *Messy casting required each time an object is retrieved from the list*

Boxing/Unboxing

- Boxing
  - *Convert a primitive type to an object*

```
Integer i = 10;   // assign int to Integer object
```

- Unboxing
  - *Convert an object to a primitive type*

```
int m = i;  // assign an Integer object (value) to int
```

- Autoboxing (compiler translates)
  - *Automatic boxing*

```
Integer i = Integer.valueOf(10);
```

  - *Automatic unboxing*

```
int m = i.intVlue();
```

Boxing/Unboxing

- With collections

```
myList.add(new Integer(10));// equals to
myList.add(10);  // compiler translates it to
                 // myList.add(Integer.valueOf(10));
```

- Warning
  - *Unboxing does not work with operators == nor !=*

```
Integer i = new Integer(10);
Integer j = new Integer(10);
if(i == j) … // not equal
```

With Generics – Example 3

```
List<Integer> myList = new LinkedList<Integer>();
myList.add(new Integer(10));
Integer n = myList.getFirst();
```

- myList can now store only Integer objects
- No cast required on retrieval

With Generics – Example 4

```
List<Student> yourList = new LinkedList<Student>();
yourList.add(new Student (“John”,”Doe”));
Student jd = yourList.getFirst();
```

- yourList can now store only Student objects
- No cast required on retrieval

Compile-Time Type Checking

- Haven't we just moved the clutter around ?
- No - compile time type checks can now be made when objects are added to the collection. Ensures that only objects of that type can be added to the collection.
- Reduction in the risk of runtime errors. A programmer error where incorrect objects are added to a list is caught at compile time rather than at runtime.

Short Summary

- With generics, we can still use the lists incorrectly, but ...
  1. *The possibility of runtime exception when non-Integer objects are inserted into the list no longer exists:*

     ```
     myList.add(new Student("John","Doe"));
     ```

     is not allowed by the compiler, a compile error is listed, so we do not get the runtime error, and the program cannot be run until the above is corrected and extraction is OK

     ```
     Integer n = myList.getFirst();
     ```

  2. *Messy casting is no longer required each time an object is retrieved from the list as the specific type is known and can also be type-checked at compile time.*
  3. *A similar compiler error would occur if we extract an object from* `yourList` *and then tried to have the wrong reference point to it*

     ```
     Integer n = yourList.getFirst(); // Student
                                      // refs only
     ```

Defining your own Generics

- The object in the <>'s is what we call a

  Formal Type Parameter

- Can be used in your own custom classes

```
public class Sack<T> {
    void insert(T x) {…}
    T getRandom() {…}
}
```

- Generic types
  - *A generic type is a generic class or interface that is parameterised over types, e.g.,* ***T****.*

Defining your own Generics

- **Parameterised type** is the type used in the actual class for instantiation, e.g.,

```
Sack<String> mySack = new Sack<String>();
```

- In this case “String” is the parameterised type.

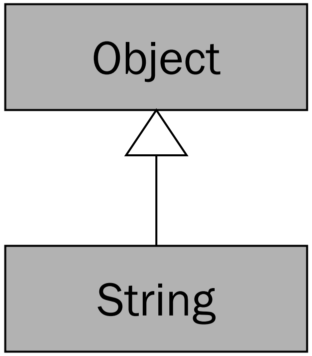
Generics and Inheritance

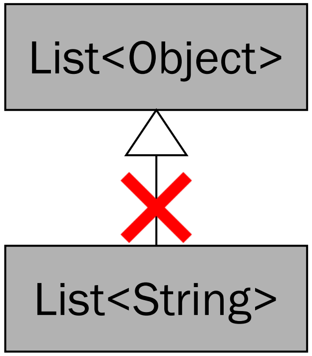
- Inheritance and sub-typing between Generic objects is not straightforward

- A List<String> is not subtype of List<Object>
    - *But, List<String> is a subtype of List (backward compatible with previous Java versions).*

```
List<String> stringLt = new List<String>();
List l = stringLt; // compiler allows it
```

- Defining generics at runtime allows the compiler to check for any sub-typing problems





Wildcards

- Supertype of a generic class is accessed using a Wildcard - ?
- To print out the contents of our sack, no matter what type it was instantiated with:

```
public void printSack(Sack<?> c) {
    System.out.println(c.getRandom());
}
```

Wildcards

- Instantiated with any type as the formal type parameter. This method can be used to print from a Sack that has been

```
Sack<MyClass> s = new Sack<MyClass>();
………  // add some MyClass items to the sack
printSack(s);
```

Wildcards

- Restrictions
  - *Using a Wildcard allows you to get all objects in the collection, but will not allow insertion of any objects*

```
Sack<?> mySack = new Sack<String>();
mySack.insert(new Object());
```

  - *The above would fail at compile time as the compiler cannot know what type of object mySack contains.*

- Wildcard can be used for <u>type safety control</u>.

Generic Methods

- What's wrong with this code?

```
class Gsack {
        ...
        public void fillSack(Object[] newItems, Sack<?> mySack)
        {
                for (Object o : newItems) {
                        mySack.insert(o);
                }
        }
}
```

Remember - you cannot insert items when using Wildcards as we don't know what type of object the Sack holds. The compiler will stop you doing this.

Generic Methods

- A solution:

```
class Gsack {
	...
	public <T> void fillSack(T[] newItems, Sack<T> mySack)
	{
		for (T o : newItems) {
			mySack.insert(o);
		}
	}
}
```

Allows the whole method to be parameterised to a certain type and enforces type compatibility between the array and the Sack.

Generic Methods

- Call generic method (example):

```
Gsack gs = new Gsack();
String[] newItems = {"report", "paper"};
Sack<String> mySack;
...

gs.fillSack(newItems, mySack); // or explicitly specify

gs.<String>fillSack(newItems, mySack); // type
```

Bounded Wildcards

- Wildcards in their basic form assume everything is an Object.
- Wildcards can be made more specific by placing bounds on them
- Bounds are made using the keyword super or extends.

```
public class Ball extends Toy {…}
public class Rattle extends Toy {…}
public void play(Sack<? super Toy> mySack) {
      mySack.insert(new Ball());
}
public void play(Sack<? extends Toy> mySack) {
      Toy t = mySack.getRandom();
} // & can separate multiple bounds
```

Bounded Wildcards

- `Syntax: supertype bounds (super) and subtype bounds (extends)`

  `? super T`

  `? extends T`

  - *Extends can also be used in this context with interfaces, see the Comparable interface example later.*

- `Generally`

  Supertype bounds allow write to a generic type T.

  Subtype bounds allow read to a generic type T.

	Write	Read
super	Yes	No
extends	No	Yes

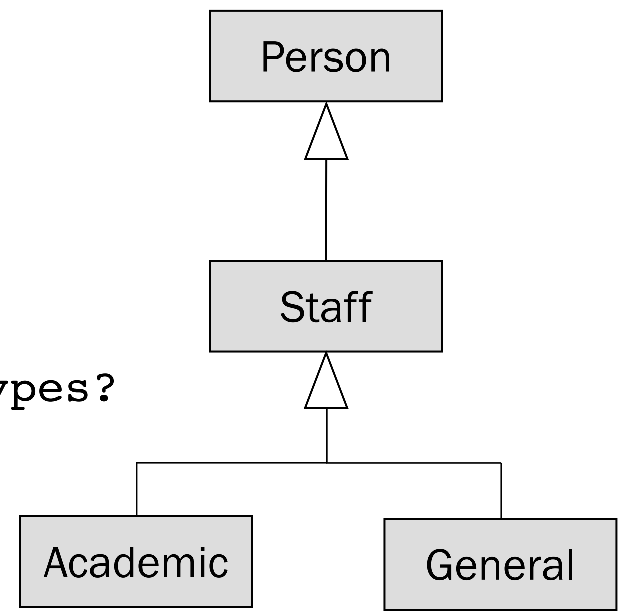
Bounded Wildcards

```
public class Employee<T> {…}

Employee<? extends Staff> e1 = new
Employee<Staff>();
Employee<? super Staff> e2 = new
Employee<Staff>();

e1.set(new Staff()); // error
Staff p1 = e1.get(); // correct
e2.set(new Staff()); // correct
Staff p2 = e2.get(); // error

What about the other super and sub types?
```



Type Erasure

- A *raw type* is automatically provided when you define a generic type.
- Compiler erases the generic types and replaces them by their bounding types or Object (without bounds)
- Unbounded type

```
public class Sack<T> {
    void insert(T x) {…}
    T getRandom() {…}
}
```

➡

```
public class Sack {
    void insert(Object x) {…}
    Object getRandom() {…}
}
```

Type Erasure

- Bounded types

```
public class Sack<T extends Toy> {
    void insert(T x) {…}
    T getRandom() {…}
}
```

⬇

```
public class Sack {
    void insert(Toy x) {…}
    Toy getRandom() {…}
}
```

When would you use Generics ?

- When using the standard Java collection objects e.g.,

```
List<String> myList = new ArrayList<String>();
Vector<String> myVector = new Vector<String>();
```

- When creating your own <u>container</u> objects, will see it in the next topic.

Java Type Parameter Naming Conventions

- By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable naming conventions that you already know about, and with good reason.
- Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.
- The most commonly used type parameter names are:
  - *E - Element (used extensively by the Java Collections Framework)*
  - *K - Key*
  - *N - Number*
  - *T - Type*
  - *V - Value*
  - *S,U,V etc. - 2nd, 3rd, 4th types*
- You'll see these names used throughout the Java SE API.

The Diamond

- This pair of angle brackets, <>, is informally called *the diamond*.
- In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context.
  - *For example, you can create an instance of Box<Integer> with the following statement:*

```
Box<Integer> integerBox = new Box<>();
```

Multiple Type Parameters

- A generic class can have multiple type parameters.
- For example, the generic OrderedPair class, which implements the generic Pair interface:

```
public interface Pair<K, V> {
	public K getKey();
	public V getValue();
}
```

```
public class OrderedPair<K, V> implements Pair<K, V> {
	private K key;
	private V value;
	public OrderedPair(K key, V value) {
		this.key = key;
		this.value = value;
	}
	public K getKey() { return key; }
	public V getValue() { return value; }
}
```

Multiple Type Parameters

- The following statements create two instantiations of the OrderedPair class:

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String>  p2 = new OrderedPair<String, String>("hello", "world");
```

- The code, new `OrderedPair<String, Integer>`, instantiates `K` as a `String` and `V` as an `Integer`.
- Therefore, the parameter types of OrderedPair's constructor are String and Integer, respectively. Due to *autoboxing*, it is valid to pass a string and an int to the class.
- It can be in a shortened version:

```
OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);
OrderedPair<String, String>  p2 = new OrderedPair<>("hello", "world");
```

- To create a generic interface, follow the same conventions as for creating a generic class.

Some Restrictions

- Type parameters cannot be instantiated with primitive types.
- Because of the Type Erasure
  - *Runtime type inquiry only works with raw types.*

```
Sack<Ball> ballSack = new Sack<Ball>();
Sack<Rattle> rattleSack = new Sack<Rattle>();
if(ballSack.getClass() == rattleSack.getClass()) // true
```

  - *Cannot create arrays of parameterised types.*

```
Sack<Ball>[] ballSack = new Sack<Ball>[5];   // error
```

`Sack[]` ~~→~~ `object[]`

C++ Templates – Example

```
template <class T>
class Sack {
  public:
    void insert(T x);
    T getRandom();
};

template <class T>
T Sack<T>::getRandom() {
  T rnd;
  // do something ...
  return rnd;
}
```

```
int main() {
  Sack<int> mysack;
  std::cout << mysack.getRandom();
  return 0;
}
```

C++ Templates vs Java Generics

Java generics and C++ templates are similar, but…

- *C++ has the keyword "template".*
- *C++ cannot restrict the types of template parameters.*
- *The use of Java Generics results in a single compiled copy of the methods and classes.*
- *C++ templates actually invoke a search-replace on the formal type parameter for each parameterised type used in the code at compile time - resulting in multiple compiled methods and classes.*

Comparable Interface

- Java has a standard generic interface Comparable<T> to compare objects.

  https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

```
int compareTo (T o);
```

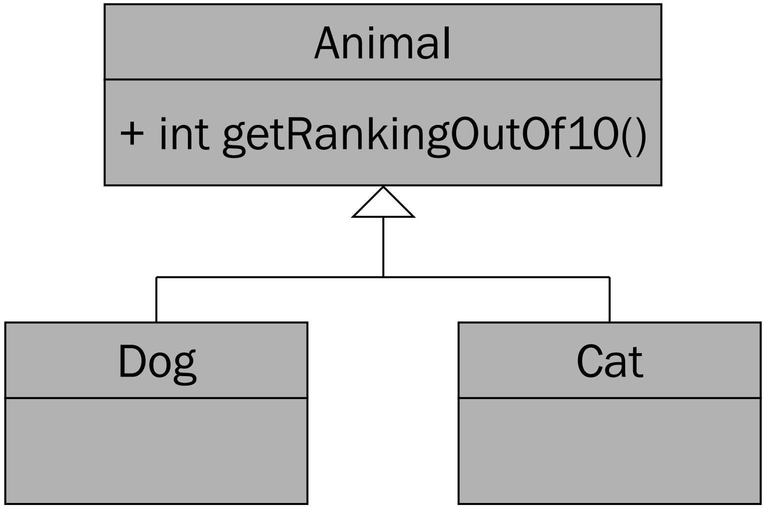
- No matter what your class is, if it implements Comparable<T>, it can compare to objects of type T.
  - *The comparison is up to the class' own definition of it (Polymorphism, late-binging)*
- Allowing code to be flexible.

Example

- Combining the generic interface *Comparable<T>* with the generic container *List<E>*.
  - *Allow generic sorting algorithms to work;*
  - *An algorithm can be defined to compare all the items in a list using the method **compareTo(T o).***

- How to write it?

Example



Example

```java
public abstract class Animal implements Comparable<Animal> {
    public abstract int getRankingOutOf10();

    @Override
    public int compareTo(Animal o) {
        if (this.getRankingOutOf10() == o.getRankingOutOf10()) {
            return 0;
        }

        if (this.getRankingOutOf10() > o.getRankingOutOf10()) {
            return 1;
        }

        return -1;
    }
}
```

Example

```java
public class Dog extends Animal {
    @Override
    public int getRankingOutOf10() {
        return 10;
    }
}

public class Cat extends Animal {
    @Override
    public int getRankingOutOf10() {
        return 0;
    }
}
```

Example

```java
import java.util.List;

public class SortDemo {

    public static void main(String[] args) {
        final List<Animal> animals = new LinkedList<>();

        animals.add(new Cat());
        animals.add(new Dog());

        sort(animals);
    }

    /**
     * Sorts a list of T objects, as long as T can be compared to other types of T
     * @param listToSort the list that will be sorted in place.
     * @param <T> the type of object to sort
     */
    public static <T extends Comparable<T>> void sort(final List<T> listToSort) {
        //...
        // do some sorting
        //...
    }
}
```

Summary

- Generic specifications provide a way of binding a certain instantiated object to a parameterised type
- Allows additional compile time checks to be made to protect against runtime type exceptions
- Can be implemented at the class level and the method level
- Wildcards can provide type safety control.
- Java generics is similar to C++ templates.