

## SENG2200/6220 Programming Languages & Paradigms

### Topic 5 - Generics in Java

Dr Nan Li  
Office: ES222  
Phone: 4921 6503  
Nan.Li@newcastle.edu.au

## What are Generics

Enhancement to the Java language introduced in 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

SENG2200/6220 PLP 2019

Java Generics

## Example 1 - Without Generics

### Without Generics

```
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.

Note: `getFirst()` is not a standard List operation in the Java libraries

SENG2200/6220 PLP 2019

Java Generics

## Example 2 - Without Generics

### Without Generics

```
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.

SENG2200/6220 PLP 2019

Java Generics

## Example - Without Generics

### Without Generics

Problems include

1. 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 the example 1 line of code  

```
Integer n = (Integer)myList.getFirst();
```
2. Messy casting required each time an object is retrieved from the list

SENG2200/6220 PLP 2019

Java Generics

## Example 1 - With Generics

### With Generics

```
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

SENG2200/6220 PLP 2019

Java Generics

## Example 2 - With Generics

### With Generics

```
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
```

## Benefits of Generics - 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.

## Example - With Generics

### 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() {...};  
}
```

## Defining your own Generics

The actual class used when instantiating your object is called the

### Parameterised Type

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

In this case "String" is the parameterised type.

## Comparison to C++ Templates

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.

## Generics and Inheritance

- Inheritance and sub-typing between Generic objects is not straightforward
- A `List<String>` is not subtype of `List<Object>`
- Defining generics at runtime allows the compiler to check for any sub-typing problems
- Supertype of a Generic class is accessed using a Wildcard - `?`

SENG2200/6220 PLP 2019

Java Generics



## Wildcards

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());  
}
```

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

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

SENG2200/6220 PLP 2019

Java Generics



## 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 objects mySack contains

- Wildcard can be used for type safety control.

SENG2200/6220 PLP 2019

Java Generics



## Generic Methods

Whats 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.

SENG2200/6220 PLP 2019

Java Generics



## 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.

SENG2200/6220 PLP 2019

Java Generics



## 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
```

SENG2200/6220 PLP 2019

Java Generics



## Bounded Wildcards

- Wildcards in their basic form assume everything is an Object.
- Can be made more specific by placing bounds on them
- Bounds are made using the keyword **super** or **extends**.
- Extends can also be used in this context with interfaces

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

SENG2200/6220 PLP 2019

Java Generics



## Bounded Wildcards

- Supertype bounds (**super**) and subtype bounds (**extends**)

```
? super T
? extends T
```

Generally

- Supertype bounds allows write to a **generic** type **T**
- Subtype bounds allows read to a **generic** type **T**.

	Write	Read
super	Yes	No
extends	No	Yes

SENG2200/6220 PLP 2019

Java Generics



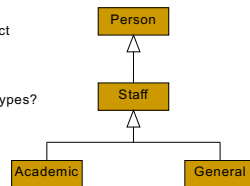
## Bounded Wildcards

```
class Employee<T> {...}
```

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

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

// What about the other super and sub types?



SENG2200/6220 PLP 2019

Java Generics



## When would you use Generics ?

- When using the standard Java collection objects

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

- When creating your own container objects

SENG2200/6220 PLP 2019

Java Generics



## 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.

SENG2200/6220 PLP 2019

Java Generics



## 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.

This pair of angle brackets, **<>**, is informally called **the diamond**.

For example, you can create an instance of **Box<Integer>** with the following statement:

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

SENG2200/6220 PLP 2019

Java Generics



## 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; }
}
```

SENG2200/6220 PLP 2019

Java Generics



## cont.

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.

As mentioned in a previous slide, because a Java compiler can infer the `K` and `V` types from the declaration `OrderedPair<String, Integer>`, these statements can be shortened using diamond notation:

```
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.

SENG2200/6220 PLP 2019

Java Generics



## Example - Comparable

- Java has a standard generic interface `Comparable<T>` to compare objects.  
<https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html>
  - The method
    - `int compareTo(T o);`
- No matter what you 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-binding)
- Allowing code to be flexible.

SENG2200/6220 PLP 2019

Java Generics



## Example - Comparable

- 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?

SENG2200/6220 PLP 2019

Java Generics



## Example - Comparable

```
1 public abstract class Animal implements Comparable<Animal> {
2     public abstract int getRankingOutOf10();
3
4     @Override
5     public int compareTo(Animal o) {
6         if (this.getRankingOutOf10() == o.getRankingOutOf10()) {
7             return 0;
8         }
9
10        if (this.getRankingOutOf10() > o.getRankingOutOf10()) {
11            return 1;
12        }
13
14        return -1;
15    }
16 }
17
```

SENG2200/6220 PLP 2019

Java Generics



## Example - Comparable

```
1 public class Dog extends Animal {
2     @Override
3     public int getRankingOutOf10() {
4         return 10;
5     }
6 }
7
8 public class Cat extends Animal {
9     @Override
10    public int getRankingOutOf10() {
11        return 0;
12    }
13 }
```

SENG2200/6220 PLP 2019

Java Generics



## Example - Comparable

```
1 import java.util.List;
2
3 public class SortDemo {
4
5     public static void main(String[] args) {
6         final List<Animal> animals = new LinkedList<>();
7
8         animals.add(new Cat());
9         animals.add(new Dog());
10
11         sort(animals);
12     }
13
14     /**
15      * Sorts a list of T objects, as long as T can be compared to other types of T
16      * @param listToSort the list that will be sorted in place.
17      * @param <T> the type of object to sort
18      */
19     public static <T extends Comparable<T>> void sort(final List<T> listToSort) {
20         //...
21         // do some sorting
22         //...
23     }
24 }
```

SENG2200/8220 PLP 2019

Java Generics



31



## 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  
Introduced in Java 1.5 (Java 5.0) - not backwards compatible with versions 1.4 or earlier

SENG2200/8220 PLP 2019

Java Generics



32

