# Lecture 17 — Generics

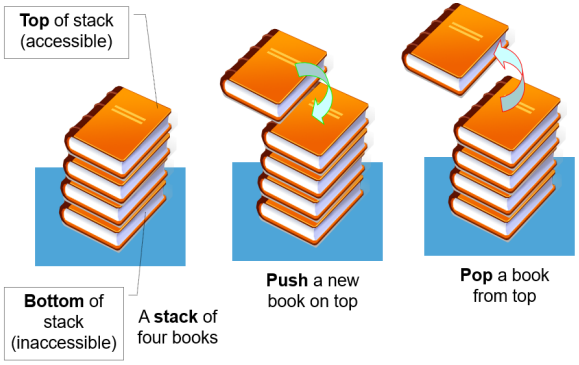
## CITS2005 Object Oriented Programming

Department of Computer Science and Software Engineering
University of Western Australia

# Contents

- See Chapter 13 of the textbook
- `Object` and generic code
- Type parameters and type erasure
- Generic classes and interfaces
- Generic methods

# The Stack Class




Taken from `https://visualgo.net/en/list`

- In lectures 14 and 15, we looked at the stack class
- Supports pushing and popping elements

# StringStack

```
public class StringStack {
    public StringStack(int capacity) { ... }
    public void push(String s) { ... }
    public String pop() { ... }
}
```

- Here is an outline of the basic class
- Notice how it only handles `Strings`?
- If we wanted to handle `ints`, we would need to rewrite the entire class
- A *generic* class would be able to handle any kind of type

# ObjectStack

```
public class ObjectStack {
    public ObjectStack(int capacity) { ... }
    public void push(Object s) { ... }
    public Object pop() { ... }
}
```

- One way to do this is to use `Object`
- Recall that all classes (except `Object`) are subclasses of `Object`
- This means an `Object` can hold an instance of any class
- This will work so long as we only want to hold non-primitive types

# Autoboxing and Unboxing

```
public static void main(String[] args) {
    ObjectStack ss = new ObjectStack(5);
    ss.push(3); // autoboxed into Integer
    ss.push("two");
    ss.push(1.0); // autoboxed into Double
    double one = (Double) ss.pop(); // unboxed into double
    System.out.println(ss.pop());
    int three = (Integer) ss.pop(); // unboxed into int
}
```

- `ObjectStack` works seamlessly with primitive types
- This is because of autoboxing and unboxing
- We can also store multiple different types since `Object` is the parent of all other classes

# `ObjectStack` pros and cons

- `ObjectStack` is good in some ways
- It is generic. We only need to write one stack, and it works for all types ✓
- It can lead to runtime errors ✗
- `int x = (Integer)stack.pop()` — what if we didn't push an `int`?
- We are essentially throwing away all the type checking Java does for us to achieve generic code
- There is a better way: enter *generics*

# GenericStack

```
public class GenericStack<T> {
    public GenericStack(int capacity) { ... }
    public void push(T s) { ... }
    public T pop() { ... }
}
```

- Notice the <T>
- This is a *type parameter*
- The name `T` was picked arbitrarily, it could be any name you like
- A type parameter is a placeholder for a real type
- The actual type of `T` is determined when a `GenericStack` is created

# GenericStack

```
public static void main(String[] args) {
    GenericStack<Integer> intStack = new GenericStack<Integer>(5);
    intStack.push(3);
    intStack.push(2);
    intStack.push(1);
    // intStack.push("Hello"); // compile-time error
    System.out.println(intStack.pop());
    System.out.println(intStack.pop());
    System.out.println(intStack.pop());
}
```

- We gave the type parameter `T` the type argument `Integer`
- Conceptually, all the `T`s are replaced with `Integer`s
- This stack only holds `Integer`s
- Note that `T` must be a class (not `int` for example)
- Autoboxing and unboxing makes this painless

# Generics with Different Types

```
public class GenericStackExample {
    public static void main(String[] args) {
        GenericStack<Integer> intStack = new GenericStack<Integer>(5);
        // GenericStack<String> stringStack = intStack; // compile-time error
        GenericStack<Integer> intStack2 = intStack; // OK
    }
}
```

- Generics with different type variables are checked for type compatibility
- It is analogous to having arrays of specific types (`int[]`, `String[]`)

# ArrayList

```
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<Integer> intList = new ArrayList<Integer>();
        intList.add(3);
        intList.add(2);
    }
}
```

- Recall `ArrayList` from the labs
- It is implemented using generics

# GenericPair

```
public class GenericPair<T, V> {
    public T first;
    public V second;

    public GenericPair(T first, V second) {
        this.first = first;
        this.second = second;
    }
}
```

- Generic classes can have multiple type parameters
- The syntax is a comma separated list of names

# GenericPair

```
import java.util.ArrayList;

public class ListPairExample {
    public static void main(String[] args) {
        var studentList = new ArrayList<GenericPair<Integer, String>>();
        studentList.add(new GenericPair<Integer, String>(10243549, "Alan Turing"));
        studentList.add(new GenericPair<Integer, String>(10243550, "Ada Lovelace"));
        studentList.add(new GenericPair<Integer, String>(20873551, "Don Knuth"));
        for (GenericPair<Integer, String> student : studentList)
            System.out.println(student.first + ": " + student.second);
    }
}
```

- This code stores an ArrayList of pairs
- Generics type parameters can use other generic types
- Note the use of `var` to avoid writing the long type twice

# Mid-lecture Break

| | | |
|---|---|---|
| C, C++, C#, Java, JS, Python, Kotlin | Arrays start at 0 |  |
| Lua, MATLAB, Scratch | Arrays start at 1 |  |
| Perl | Arrays can start wherever ¯_(ツ)_/¯ |  |
| STAR WARS | Arrays start at 4, stop at 6, restart at 1, stop again at 3, restart at 7 then continue on |  |

# Bounded Type Parameters

- Java allows *bounded type parameters*
- Consider `<T extends SuperClass>`
- This type parameter will accept any type argument that is either `SuperClass` or a subclass of `SuperClass`
- They are called bounded since `SuperClass` is an upper bound on the type of `T`
- Usually, `<T>` would accept any type
- Note that this is equivalent to `<T extends Object>`

# BirdPair

```
abstract class Bird {
}

class Emu extends Bird {
}

class Hawk extends Bird {
}

class BirdPair<T extends Bird> {
    public T first;
    public T second;

    public BirdPair(T first, T second) {
        this.first = first;
        this.second = second;
    }
}
```

- `BirdPair` uses a bounded type parameter

# BoundedType example

```
public class BoundedType {
    public static void main(String[] args) {
        var emuPair = new BirdPair<Emu>(new Emu(), new Emu());
        var hawkPair = new BirdPair<Hawk>(new Hawk(), new Hawk());
        var birdPair = new BirdPair<Bird>(new Emu(), new Hawk());
        // var badPair = new BirdPair<String>("Hello", "World");
    }
}
```

- The compiler will not allow `BirdPair<String>` or similar due to the bounded type parameter

# Generic Interfaces

- Interfaces can be generic too
- Similar to generic classes, generic interfaces have type parameters
- Type parameters can be used in the method signatures
- Example: `Comparable<T>` interface in Java API

# Comparable<T> Interface

- `Comparable<T>` is a generic interface
- Requires the implementation of the `compareTo(T o)` method
- The `compareTo` method compares the current object with the specified object
- Returns a negative, zero, or positive integer if the current object is less than, equal to, or greater than the specified object
- Helps to sort objects in a natural order
- `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Comparable.html`

# RunningMaximum Example

```
public class RunningMaximum<T extends Comparable<T>> {
    private T currentMax;


    public RunningMaximum(T initial) {
        currentMax = initial ;
    }

    public void addNumber(T number) {
        if (number.compareTo(currentMax) > 0) {
            currentMax = number;
        }
    }

    public T getCurrentMax() {
        return currentMax;
    }
}
```

- Notice how `T` can appear as a type parameter of `Comparable`

# RunningMaximum Example Usage

```
public static void main(String[] args) {
    RunningMaximum<Integer> runningMax = new RunningMaximum<>(1);
    System.out.println(runningMax.getCurrentMax());
    runningMax.addNumber(5);
    System.out.println(runningMax.getCurrentMax());
    runningMax.addNumber(3);
    System.out.println(runningMax.getCurrentMax());
    runningMax.addNumber(7);
    System.out.println(runningMax.getCurrentMax());
}
```

- The `RunningMaximum` class allows us to compute the running maximum of a series of calls to `addNumber`
- Works with any type that implements `Comparable<T>`
- Note the `new RunningMaximum<>(1)` syntax

# Generic Methods

- Methods can be generic as too
- Declared with a type parameter similarly to classes
- The type parameter can be used within the method body
- Example: A generic max function

# Writing a Generic Max Function

```
public class GenericMax {
    public static <T extends Comparable<T>> T max(T a, T b) {
        if (a.compareTo(b) > 0) {
            return a;
        } else {
            return b;
        }
    }

    public static void main(String[] args) {
        // see full code
    }
}
```

- Static methods of generic classes cannot see the type parameter and do not work
- But you can write a generic static method instead

# Type Erasure

- In Java, generics are implemented using *type erasure*
- During compilation, generic types are checked
- After compilation, all generic types are erased
- In effect, `T` becomes `Object`
- However, the correct casts are added for parameters/return values
- This leads to numerous sharp edges
- Many of these are too complicated to cover here, but we will mention a few common examples
- You are not expected to know the details of any of these!
- They key point we expect you to understand is that Java uses type erasure, and this can lead to errors

# Common Tricky Aspects of Generics

- Cannot create a new instance of a type parameter (e.g., `new T()`)
- Cannot create a generic array (e.g., `new T[100]`)
- Cannot perform `instanceof` check with parameterized type (e.g., `if (obj instanceof ArrayList<String>)`)
- Generic class method ambiguity errors (e.g., `void myMethod(T o)` and `void myMethod(V o)`)

# Extra Bits

- Generics in Java are complicated
- In addition to the previously mentioned sharp edges, there are some features we do not have time to cover
- Wildcards and bounded wildcards (a way to accept generic parameters to methods)
- Generic constructors (similar to generic methods)
- Raw types
- You will not be expected to learn about any of these, but details can be found in the textbook for those who are interested