Chapter 19

Generics

- A generic is a type that can be instantiated with a concrete type
- Example

```
ArrayList<String> list = new ArrayList<>();
```

- List must be of `Strings`
- Definition of an `ArrayList` is

```
class ArrayList<E>
```

- `<E>` is a *formal generic type*, which can later be replaced with an *actual concrete type*
- Replacing a generic type is called *generic instantiation*

Generics

- By convention, a single capital letter such as `E` or `T` denotes a formal generic type
- Can think of as a *type variable*, value to be assigned later
- Only reference types (classes) can be used with Generics
- Casting is not needed to retrieve a value from a Generic type

```
String color = list.get(0);
```

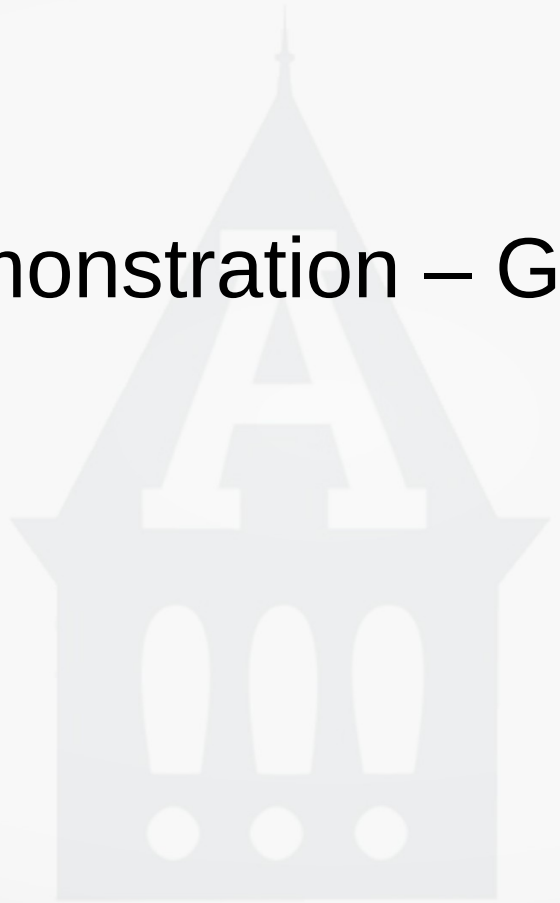
- Normal autoboxing and autounboxing still apply (when applicable)

```
ArrayList<Integer> list = new ArrayList<>();
list.add(1);
```

Defining Generic Classes and Interfaces

- You can use a generic type for a class or an interface
- Example: a custom `Stack` class
- Want a `Stack` of *any* reference type of element
  - Empty on creation
  - Add new element to the top of the stack
  - Return # of elements on the stack
  - Return (peek) top element of the stack
  - Return and remove top element of the stack
  - Return true if the stack is empty

Code Demonstration – Generic Stack



Generic Methods

- Concept: A generic type can be defined for a static method
- Consider the following method

```
public static <E> void printArray(E[] data) {
    for (E item : data) {
        System.out.printf("%s ", item);
    }
    System.out.println();
}
```

Using a Generic Method

- Can be used as follows

```
String[] cities = { "Paradise", "Nibley", "Logan" };
Integer[] zipCodes = { 84328, 84326, 84321 };

printArray(cities);
printArray(zipCodes);
```

- The method can also be invoked as follows

```
Generics.<String>printArray(cities);
Generics.<Integer>printArray(zipCodes);
```

Bounded Generic Types

- Possible to restrict the types allowed to match a Generic method
- Example

```
public static <T extends GeometricObject> boolean equalArea(T o1, T o2) {
    return o1.getArea() == o2.getArea();
}
```

- Known as a *bounded generic* (versus *unbounded*)

Bounded Generics and Polymorphism

Consider the following

```
public static double totalArea(ArrayList<GeometricObject> objects) {
    double total = 0;
    for (GeometricObject item : objects) {
        total += item.getArea();
    }
    return total;
}
```

Example continues

- Cannot pass `ArrayList<Circle>` because type must be `GeometricObject`
- But could specify as follows

```
public static <T extends GeometricObject> double totalArea(ArrayList<T> objects) {
    double total = 0;
    for (T item : objects) {
        total += item.getArea();
    }
    return total;
}
```

- This method will (only) accept an `ArrayList` of anything derived from `GeometricObject`

Raw Types and Backward Compatibility

- Generics were not in the original Java language release
- Must maintain backwards compatibility with previous code
- Issues this has created
  - Possible to use a generic class without specifying a concrete type

    ```
    GenericStack myStack = new GenericStack();
    ```

  - Same as

    ```
    GenericStack<Object> myStack = new GenericStack<Object>();
    ```

  - When you use a generic without specifying the type, it is called a *raw type*
  - You can accidentally do this and the compiler won't let you know you did
  - Read Section 19.8 on Type Erasure for additional details