Object Oriented Programming

Generic Programming

Dr. Seán Russell
sean.russell@ucd.ie

School of Computer Science,
University College Dublin

September XX, 2019

Learning outcomes

After this lecture and the related practical students should...

- understand the concept of generic programming
- be able to correctly construct and use generic classes
- be able to define generic classes and interfaces

Table of Contents

Data Structure Types

- Initial data structures can only be used to store a single type of data
- After learning inheritance, the same data structure that can store any type of object

```java
public interface Stack{
  public Object top();
  public Object pop();
  public void push(Object o);
  public int size();
  public boolean isEmpty();
}
```

Problems

- The the return type of `top` and `pop` and the parameter of `push` are all `Object`
- Any type of object will be accepted
- A stack that can store one type of object, can also store any other type of object
- When we pop an item, it will not be returned as it's correct type
- We may have to **typecast** the value returned

Using Returned Values

```java
Stack st = new ArrayStack();
// place some point objects on the stack
Object o = st.pop();
Point p = (Point)o;
System.out.println(p.getX() +", " + p.getY());
```

- Any object we attempt to typecast may not be a `Point`, then there will be an exception
- This makes our code look more complicated and it is easier to make **mistakes** when programming

Possible Solution

- We could create a new copy of the data structure for each type of data
- Each copy could only store that type of data and would return values in the correct type
- For `Points`, we would create a copy of the interface and implementation classes, named `PointStack`, and change parameters and return types of the important methods to `Point`

Point Stack

```java
public interface PointStack{
  public Point top();
  public Point pop();
  public void push(Point o);
  public int size();
  public boolean isEmpty();
}
```

Problems

- We would have many copies of the same classes and interfaces, each with different return types and parameters
- A small change to how the data structure works, we would require changes to every copy that we have created

Table of Contents

Generics

- A better solution is to use a feature of Java called **generics**
- Generic programming allows algorithms and classes are written for types that will be **specified later**
- They actual type used is then specified when the object is **constructed**, not when the class is written
- This style of programming is particularly useful for creating data structures

Generics

- Using generics we can choose the type of **some** parameters, variables and return types will be when we are **using** a data structure
- This way Classes that can store any type of data
- Generic class **do not** require us to typecast returned values accept parameters as `Object` type
- This feature is often called template programming
- We can easily use the same template again and again for different types

Table of Contents

Using a Generic Class

- When an object is constructed based on a generic class we must also specify what type(s) it will use
- When the type is specified, Java fills in the template and we can use the object
- As an example we will look at the `linkedList` class in the Java API
- This is a generic class that can be used to store any type of data

Generic Object without Type

- Normally, to construct an object of the `linkedList` class, we would use the code

```
LinkedList list = new LinkedList();
```

- This code will cause warning that you are not specifying the type
- This list can store any type of data and we would be required to typecast data when we remove it from the list

Constructing a LinkedList Object

- The syntax for constructing a generic object is almost the same as constructing any other object
- We simply need to add the type inside angle bracket (<>) at the declaration of the variable and at the use of the constructor

```
ClassName<Type> var = new ClassName<Type>(params...);
```

- Strings: `LinkedList<String> strings = new LinkedList<String>();`
- Points: `LinkedList<Point> points = new LinkedList<Point>();`

Restricted Types

- Once we construct a generic object, we can only use the correct type for parameters
- For example, the `add` method of the `strings` object will only accept the type `String`
- All other types will cause a compiler error
- The insertion methods of the `points` object will only accept the type `Point`
- This prevents us from causing errors in our programs at run time by storing the incorrect type of data

Table of Contents

Declaring a Generic Class or Interface

- To define a class or interface as generic, we need to add a little bit of information to the definition
- Generic classes and interfaces can have many type parameters to define different types, however in all of our example we will only be using a single type
- Defining a class or interface as generic is done by adding angle brackets containing a single upper case character for each generic type we will use after the class or interface name

Multiple Generic Parameters

- For a single generic type, we usually use the letter `T`, for example the definition of a generic stack interface would be `public interface Stack<T>{`
- If we are declaring multiple generic parameters, we use a different letter for each generic type and separate them using commas
- A typical example of this is the Map class, which has a generic type for the key (`K`) and a generic type for the value (`V`)
- The definition of the Map interface in the API is `public interface Map<K,V>{`

Table of Contents

Defining a Generic Interface

- The generic types that we use in the definition can then be used in place of actual types throughout the class or interface
- For example in the Stack interface, instead of defining the method `public Object top();`, we define it as `public T top();`
- This means that when we use the Stack interface, the type parameter we supply (String, Point or any other class) replaces T in the method definition

Generic Stack Interface

```
public interface Stack<T>{
  public T top();
  public T pop();
  public void push(T o);
  public int size();
  public boolean isEmpty();
}
```

- It is important to note that not every type in the interface is T
- Only the types representing the data we are storing
- It would not make very much sense it the return type of the `size` method changed depending on what type of data we are storing

Table of Contents

Implementing a Generic Interface

- As the stack interface is generic, if we are implementing the interface we must specify the type
- For example, if we were defining a array implementation of a stack for only strings we would use the definition

```
public class ArrayStack implements Stack<String>{
```

- This would work, but only for strings
- Instead we want it to work for different types of data, this means that the `ArrayStack` class must also be generic
- This gives us the code

```
public class ArrayStack<T> implements Stack<T> {
```

Implementing a Generic Interface

- `public class ArrayStack<T> implements Stack<T> {`
- Here we can see that there are two sets of angle brackets specifying a generic type
- The first states that the array stack class has a generic parameter `T`
- The second states that the array class is implementing the Stack interface based on the generic type `T`
- This basically means that whatever generic type the class is using is also used by the stack interface

Generic Instance Variables

- Next we need to define the instance variables that are represented in the stack
- In the array based version, the first will be an array
- However, we do not know what type of data will be stored in the array, so we use the generic type `T`
- Whatever type is used when the object is constructed, the array will store instances of that
- The second instance variable is an integer that represents the index of the array where the next item is added
- It does not matter what type of information is being stored in the array, this will always be an `int`

Generic Constructor

- Next we need to define the constructor for the class
- The constructor specifies the size of the array, and the number of items that can be stored in the stack
- It is not possible to create an array of a generic type, e.g. `new T[5];` will not compile
- The solution is to construct an array of Objects
- This can store any type, so it can store our type
- First need to type cast it to the type we are using
- This cannot be done for local variables, only instance variables

Constructor and Instance Variable

```java
public class ArrayStack<T> implements Stack<T> {
  private T[] items;
  private int top;
  public ArrayStack(int s){
    items = (T[]) new Object[s];
    top = 0;
  }
```

Defining Generic Methods

- Implementing the `top` and `pop` methods in the generic array stack class is little different that in the original class
- The only change we need to make is to change the return type to the generic type `T`
- This makes sense because when the type of data that the class is returning changes, the return types of these methods should match it

Defining Generic Methods

```
public T top() {
  return items[top -1];
}
public T pop() {
  T item = items[top-1];
  top--;
  return item;
}
```

Generic Method Parameters

- The push method is the only way that data can be added to the stack
- If we want our stack to store only one type of data, then the parameter of the method is what defines what is allowed
- If we change the type of this to the type `T`, then this is the only type of data that can be added
- The `size` and `isEmpty` methods do not change
- No matter what type of data is being stored, the size will always be an integer and condition of being empty will always be boolean

Generic Method Parameters

```
  public void push(T o) {
    items[top] = o;
    top++;
  }
  public int size() {
    return top;
  }
  public boolean isEmpty() {
    return top == 0;
  }
}
```

Table of Contents

Why Use Generics

- Generic programming is a very useful feature of most modern object-oriented programming languages
- However, application programmers (us) very rarely define generic classes
- That is because generics is most useful when developing data structures and most of these have already been provided within the Java libraries
- Application programmers mostly are required to be able to understand and use the generic classes that are available
- But what are the benefits of using generic classes?

Benefits of Generics

- There are two main reasons to use generic classes in our applications:
  - It enables stronger type checking
  - It reduces the use of typecasting

Table of Contents

Stronger type checking

- When a program is complied, the compiler compares the type of each parameter in every method against the types that the method is defined to accept
- If a parameter is used that is not the same as the parameter in the method definition, then the compiler will show an error
- If we are defining our classes to accept and return Object, then the type check will always pass
- Replacing this data structure with a generic data structure means that this error will be detected by the compiler
- This allows us to find errors quickly

Stronger type checking

- If we are using a non-generic stack to store strings, we can push any type of data onto the stack
- If we accidentally push an integer onto the stack there will now be a data type that we did not want in our data structure
- When we eventually come to remove the integer, we will attempt to typecast it to a string and the program will throw a `ClassCastException`
- Replacing the stack with a generic stack, allows us to define the type parameter for the stack as String
- When we attempt to push an integer onto the stack, we will get a compiler error and the mistake will never appear at runtime

Table of Contents

Reducing the use of typecasting

- The use of generic classes means that the return types of our methods will generally be correct
- This means that we do not need to typecast objects when they are removed from the data structure that we are using
- Typecasting is a useful feature, but it is very easy for a mistake to be made and for this to cause a problem in your program

Reducing the use of typecasting

```java
Stack s = new ArrayStack();
s.push("S1");
String m = (String) s.pop();
```

- In this example, every time we remove a string from the stack, it is returned as an object
- This means that if we wish to actually use the strings (count characters, search, or get sub strings) we will have to typecast the object that is returned
- Every time that a type cast is used, there is potential for the program to fail

Reducing the use of typecasting

```java
Stack<String> s = new ArrayStack<String>();
s.push("S2");
String m = s.pop();
```

- In this example, there is no typecasting required
- Not only is the code a little easier to read, but there is also less potential problems that can happen during execution

Table of Contents

Generics and Primitive Types

- Generics types can only be used with objects
- The primitive types that we use, such as `int`, `float`, `double` and `char` are not objects
- If we want to have generic class that can store these types of data, we have to use the matching class
- Every primitive type has a class that can be used to represent it and perform certain useful functions
- These classes are more than just utilities that are useful for performing certain functions, they can also be used to create objects that represent a value of that primitive type
- For example `Integer i = new Integer(123);` creates an object that represents the value of the `int` 123

Wrapper Classes

The classes that represent each of the primitive types are given in the table below

Primitive type	Class Name
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Using Wrapper Classes

- This means that if we wish to use a generic class to represent the primitive type `int`, we would actually declare the object to with the type parameter `Integer`

```
Stack<Integer> s = new ArrayStack<Integer>();
```

Table of Contents

Wrapping a Primitive Value

- If the array stack in the example only accepts Integer object, then we need to create an integer object

```java
Stack<Integer> s = new ArrayStack<Integer>();
int i = (some caculated computations);
s.push(new Integer(i));
```

- This added code makes the use of our stack a little more difficult to understand
- We can simply use the `int i` and it will be automatically converted into an `Integer` object
- This process is called **autoboxing**
- We can replace the last line of the example with `s.push(i);`

Unboxing

- Additionally, when an object is returned that represents a primitive value, we can store this value directly in a primitive variable
- This process is called **unboxing**
- For example, the return type of the stack is `Integer`, rather than having to use the method `intValue` to return a primitive type we can simply assign the return value to the variable

```java
Stack<Integer> s = new ArrayStack<Integer>();
int i = s.pop();
```

Automatic boxing and unboxing

- This process is done automatically by the compiler every time it is necessary
- If a method expects an `int` and the parameter is an `Integer` object, it will be automatically unboxed
- If a method expects an `Integer` object and the parameter is an `int`, then the variable will be autoboxed