

Object-Oriented Programming

Generic Programming

Dr. Seán Russell
`sean.russell@ucd.ie`

School of Computer Science,
University College Dublin

November 26, 2018

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
- be able to define generic methods

Table of Contents

- 1 Problems With Data Structures
- 2 Generics
- 3 Defining Generic Classes and Interfaces
- 4 Reasoning about Generic Types
- 5 Why Use Generics
- 6 Generics and Primitive Types

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

```
1 public interface Stack{  
2     public Object top();  
3     public Object pop();  
4     public void push(Object o);  
5     public int size();  
6     public boolean isEmpty();  
7 }
```

Problems

- The the return type of `top` and `pop` and the parameter of `push` are all `Object`
- Using polymorphism, any type that extends `Object` can be used
- If we wish to use a `Stack` to store `Point` objects, this interface will work
- There are some problems with this
- A stack that can store point object, can also store any other type of object
- When we `pop` an item from the stack, it will be returned as an object (not a point)
- We may have to typecast the value returned

Using Returned Values

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

- If an object that was not a point was pushed onto the stack, then when we attempt to typecast it there will be an exception
- Using polymorphism in this way means that our code is more complicated and it is easier to make mistakes when programming

Possible Solution

- There are two possible solutions to this problem
- The first solution is to create a new copy of the data structure for each type of data that we want to hold
- Each copy could only store that type of data and would return values in the correct type
- For example, if we needed to use a stack to store point objects, we would create a copy of the interface and implementation classes, perhaps named `PointStack` or something similar, and change the code so that the parameters and return types of the relevant methods are all `Point`

Point Stack

```
1 public interface PointStack{  
2     public Point top();  
3     public Point pop();  
4     public void push(Point o);  
5     public int size();  
6     public boolean isEmpty();  
7 }
```


Problems With Solution

- This is not a very good solution
- There will be many copies of the same classes and interfaces, each with different return types and parameters
- If we want to make even a small change to how the data structure works, we would need to make this change in every copy that we have created

Table of Contents

- 1 Problems With Data Structures
- 2 Generics
 - Using Generics
- 3 Defining Generic Classes and Interfaces
- 4 Reasoning about Generic Types
- 5 Why Use Generics
- 6 Generics and Primitive Types

Generics

- The second solution is to use a feature of Java called **generics**
- Generic programming is a style of programming where algorithms and classes are written in terms of types that will be **specified later**
- The 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

- Generics allow is to choose what the type of **some** parameters, variables and return types will be when we are **using** a data structure
- This allows us to write a data structure or any other class that can store any type of data
- Generic class **do not** require us to typecast the values returned or allow different types of objects to be stored in the same object
- The same feature in C++ is called templates
- This is because we write code like a template where the important details such as types are filled in later
- We can easily use the same template again and again for different types

Table of Contents

- 1 Problems With Data Structures
- 2 Generics
 - Using Generics
- 3 Defining Generic Classes and Interfaces
- 4 Reasoning about Generic Types
- 5 Why Use Generics
- 6 Generics and Primitive Types

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 is acceptable code in Java, however it will generally give you warning that you are not specifying the type
- When we construct the object this way, it 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

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

- If we wish to create a linked list for storing strings, the code would be `LinkedList<String> strings = new LinkedList<String>();`
- If we want to create a linked list that can only store points the code would be `LinkedList<Point> points = new LinkedList<Point>();`

Restricted Types

- Once we have constructed a generic object, we are restricted to the correct type for certain parameters
- For example, the insertion 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

- 1 Problems With Data Structures
- 2 Generics
- 3 Defining Generic Classes and Interfaces**
 - Defining a Generic Interface
 - Defining a Generic Class
- 4 Reasoning about Generic Types
- 5 Why Use Generics
- 6 Generics and Primitive Types

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

- 1 Problems With Data Structures
- 2 Generics
- 3 Defining Generic Classes and Interfaces
 - Defining a Generic Interface
 - Defining a Generic Class
- 4 Reasoning about Generic Types
- 5 Why Use Generics
- 6 Generics and Primitive Types

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

```
1 public interface Stack<T>{  
2     public T top();  
3     public T pop();  
4     public void push(T o);  
5     public int size();  
6     public boolean isEmpty();  
7 }
```

- 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 if the return type of the size method changed depending on what type of data we are storing

Table of Contents

- 1 Problems With Data Structures
- 2 Generics
- 3 Defining Generic Classes and Interfaces
 - Defining a Generic Interface
 - Defining a Generic Class
- 4 Reasoning about Generic Types
- 5 Why Use Generics
- 6 Generics and Primitive Types

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

```
1 public class ArrayStack<T> implements Stack<T> {  
2     private T[] items;  
3     private int top;  
4     public ArrayStack(int s){  
5         items = (T[]) new Object[s];  
6         top = 0;  
7     }
```

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

```
1  public T top() {  
2      return items[top - 1];  
3  }  
4  public T pop() {  
5      T item = items[top - 1];  
6      top--;  
7      return item;  
8  }
```

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

```
1  public void push(T o) {  
2      items[top] = o;  
3      top++;  
4  }  
5  public int size() {  
6      return top;  
7  }  
8  public boolean isEmpty() {  
9      return top == 0;  
10 }  
11 }
```

Table of Contents

- 1 Problems With Data Structures
- 2 Generics
- 3 Defining Generic Classes and Interfaces
- 4 Reasoning about Generic Types**
 - Bounded Type Parameters
 - Multiple Bounds
- 5 Why Use Generics
- 6 Generics and Primitive Types

Reasoning about Generic Types

- When we use a generic type in our class, we do not know exactly what type the object will be until the class is used
- This means that while we are writing code, we have no knowledge of what methods we can use on the objects of the generic type
- This means that the only methods that we can use are those that are available in all classes, i.e. the methods defined in the `Object` class

Table of Contents

- 1 Problems With Data Structures
- 2 Generics
- 3 Defining Generic Classes and Interfaces
- 4 Reasoning about Generic Types**
 - Bounded Type Parameters
 - Multiple Bounds
- 5 Why Use Generics
- 6 Generics and Primitive Types

Bounded Type Parameters

- There may be times when you want to restrict the types that can be used as type arguments in a parametrised type
- For example, a method that operates on numbers might only want to accept instances of `Number` or its subclasses
- This is what bounded type parameters are for
- To declare a bounded type parameter, list the type parameter's name, followed by the `extends` keyword, followed by its upper bound, which in this example is `Number`

Using Bounded Type Parameters

- Where we would normally declare a type as `<T>`, we instead use `<T extends ClassOrInterface>`
- This means the class can only be used with types that either are or extend the named class or implement the named interface
- Now that we know that the generic type parameter must be of a particular type (or subclass), we can make use of the methods of that class or interface
- This means that where previously we were restricted to the methods of objects, the generic type parameters are much more useful once the type is bound

Table of Contents

- 1 Problems With Data Structures
- 2 Generics
- 3 Defining Generic Classes and Interfaces
- 4 Reasoning about Generic Types**
 - Bounded Type Parameters
 - Multiple Bounds**
- 5 Why Use Generics
- 6 Generics and Primitive Types

Multiple Bounds

- It is possible to apply multiple bounds to the same generic type parameter
- A type variable with multiple bounds is a subtype of all the types listed in the bound
- If one of the bounds is a class, it must be specified first
- Multiple bounds are shown by having a list of classes and interfaces separated by the ampersand character (&)

Multiple bounds on a type parameter

```
1 <T extends A & B & C>
```


Multiple Bounds

Multiple bounds on a type parameter

```
1 <T extends A & B & C>
```

- In this definition, it is assumed that A, B and C are all classes or interfaces in Java
- Assuming that A is a class and B and C are interfaces, the type that is used must be a subclass of A and implement both the interfaces A and B

Table of Contents

- 1 Problems With Data Structures
- 2 Generics
- 3 Defining Generic Classes and Interfaces
- 4 Reasoning about Generic Types
- 5 Why Use Generics**
 - Stronger type checking
 - Reducing the use of typecasting
- 6 Generics and Primitive Types

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

- 1 Problems With Data Structures
- 2 Generics
- 3 Defining Generic Classes and Interfaces
- 4 Reasoning about Generic Types
- 5 Why Use Generics**
 - Stronger type checking
 - Reducing the use of typecasting
- 6 Generics and Primitive Types

Stronger type checking

- When a program is compiled, 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

- 1 Problems With Data Structures
- 2 Generics
- 3 Defining Generic Classes and Interfaces
- 4 Reasoning about Generic Types
- 5 Why Use Generics**
 - Stronger type checking
 - Reducing the use of typecasting**
- 6 Generics and Primitive Types

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

```
1 Stack s = new ArrayStack();  
2 s.push("S1");  
3 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

```
1 Stack<String> s = new ArrayStack<String>();  
2 s.push("S2");  
3 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

- 1 Problems With Data Structures
- 2 Generics
- 3 Defining Generic Classes and Interfaces
- 4 Reasoning about Generic Types
- 5 Why Use Generics
- 6 Generics and Primitive Types
 - Autoboxing and Unboxing

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`

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

Table of Contents

- 1 Problems With Data Structures
- 2 Generics
- 3 Defining Generic Classes and Interfaces
- 4 Reasoning about Generic Types
- 5 Why Use Generics
- 6 Generics and Primitive Types
 - Autoboxing and Unboxing

Wrapping a Primitive Value

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

```
1 Stack<Integer> s = new ArrayStack<Integer>();  
2 int i = (some caculated computations);  
3 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

```
1 Stack<Integer> s = new ArrayStack<Integer>();  
2 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