Generics

CSC02A2

Outline

Outline

Generics

Generic Programming

Generic programming is a *programming paradigm* that aims to create efficient, reusable software.

C++ uses the **`Standard Template Library (STL)`** for generic programming, **Java** uses Generics.

Generic principles:

- Code once and use repeatedly.
- Allows for types (classes and interfaces) to be parametrized when defining classes, interface and methods.

Using Generics

Generics can be used with either whole classes (e.g. **`ArrayList`**) or specific methods.

`ArrayList` is a resizable-array implementation of the List interface. It allows a resizable array of any type[1].

```
// T is can be any data type
// Either primative wrapper or reference type
ArrayList<T> array = new ArrayList<>();
```

The code below shows the syntax for declaring a ***generic method***.

```
// Note that <T> is used as a modifier for the method
// This modifier indicates that this method requires a type parameter
public static <T> void genericMethod1(T tVar1, T tVar2, int a)
{
  //Do something relating to the T-type parameter
}
```

[1] **<T>** is not the same as **<T1>** and not the same as **<Type>**...so pick one and stick to it

Example of a Generic Class I

```java
import java.util.ArrayList;

public class GenericStack<T>
{
  private ArrayList<T> list = new ArrayList<T>();

  public int getSize()
  {
    return list.size();
  }

  public boolean isEmpty()
  {
    return list.isEmpty();
  }

  public T push(T o)
  {
    list.add(o);
    return o;
  }

  public T pop()
  {
    T o = list.get(getSize() - 1);
    list.remove(getSize() - 1);
    return o;
  }
}
```


Example of a Generic Class II

```
1  public class Main
2  {
3    public static void main(String[] args)
4    {
5      // When specifying the <T> you must use wrapper classes
6      // (Integer/Double) for primitive types
7      GenericStack<Integer> intStack = new GenericStack<>();
8      intStack.push(400479);
9      intStack.push(2015);
10     intStack.push(31337);
11
12     // Note that the code in the comment below would not work.
13     // This intance of GenericStack can only work with Integers
14     // intStack.push("String ");
15
16     while (intStack.getSize() > 0)
17     {
18       word += intStack.pop();
19     }
20     System.out.println("The contents of the stack: " + word);
21   }
22 }
```


Example of a Generic Method

```java
public class GenericMethods
{
  private static Integer[]  even  = { 2, 4, 6, 8 }; // Note Integer not int
  private static String[]   odd   = { "Strange", "Weird", "Uncanny" };

  // Generic method
  public static <T> void print(T[] list)
  {
    // Using an array of type
    // Some restrictions exist when using arrays
    for (T t : list)
    {
      System.out.println(t);
    }
  }

  public static void main(String[] args)
  {
    // Calling generic method with Integer types
    GenericMethods.<Integer>print(even);
    // Calling generic method with reference types
    GenericMethods.<String>print(odd);
  }
}
```


Type Erasure and Generics Restrictions

Type Erasure

Generics were introduced to the **Java** language to provide tighter type checks at compile time and to support generic programming. To implement generics, the **Java** compiler applies ***type erasure***:

Type Erasure

Type Erasure replaces all ***type parameters*** in *generic types* with their *bounds* (BoundedTypes) or **`Object`** if the type parameters are *unbounded*.

The **Java** compiler does to the following:

- Replaces all type parameters at compile time.
- Insert type casts if necessary to preserve type safety.
- Generate bridge methods to preserve polymorphism in extended generic types.

Generics Restrictions

Due to ***type erasure***, **Java** generics have the following restrictions:

- You cannot instantiate a generic type: `T object = new T(); // not allowed`
- Generic array creation is forbidden: `T[] array = new T[5]; // not allowed`
  This can be avoided via casting: `T[] array = (T[])(new Object[5]);`
- You cannot use a generic type in a ***`static`*** context.
- Exception classes cannot be generic.
- Generics do not support primitive types (because of backwards compatibility with previous JVMs).

(Un)BoundedTypes, RawTypes and Wildcards

(Un)BoundedTypes and RawTypes

UnboundedTypes do not place any restrictions on the allowed types accepted.

BoundedTypes *restrict* the allowed types accepted by a generic method or class to a subclass of a given class. `<T extends SuperClassName>`

RawType is the name of a generic class or interface without any type arguments:

```
ArrayList arrList = new ArrayList();
```

- No **<T>** has been specified.
- Therefore, `arrList` defaults to the RawType of **`Object`**.
- Use of RawTypes enables backward compatibility.
- Due to numerous API classes not being generic prior to JDK 5.0, RawTypes are included.

Wildcards

To explain wildcards assume that **`Rectangle`** is a subtype of **`GeometricObject`**.

- It is *not* correct to say that **`ArrayList<Rectangle>`** is compatible with **`ArrayList<GeometricObject>`**.
- In order to write a generic method which would accept both you would need to use generic wildcards:
- `ArrayList<? extends GeometricObject>`
  e.g. `public void methodName(ArrayList<? extends GeometricObject> list`

There are three (3) ***wildcard formats*** for generics:

Unbounded : `<?>`

Bounded : `<? extends Type>` (match Type or one of its subclasses).

Lower-bounded : `<? super Type>` (match Type or one of its superclasses).