INFO1113 Object-Oriented Programming

Week 7A: Generic classes and methods

Copyright Warning

Topics

Generics

As part of usage with collection classes, you have been able to specify a type that will be contained within the collection.

Generics gives us the ability to handle multiple different types without needing to rewrite the same code.

Refer to Chapter 12.1, pages 905-914, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Generics

The advantages of generics

- Stronger type checks at compile time.
- Elimination of casts.
- Enabling programmers to implement generic algorithms.

Generics

Generics are specified as part of the class definition. We are able to show the parameter types that can be generalised by the class class.

Syntax:

```
    [public] class ClassName<Param0[,Param1..]>
```

Example:

```java
public class Container<T>
```

Generics

Generics are specified as part of the class definition. We are able to show the parameter types that can be generalised by the class class.

Syntax:

```
    [public] class ClassName<Param0[,Param1..]>
```

Example:

```
public class Container<T>
```

We have specified a type parameter here. This allows us to create a variable to represent the type within our class.

Generics

Generics are specified as part of the class definition. We are able to show the parameter types that can be generalised by the class class.

Syntax:

```
    [public] class ClassName<Param0[,Param1..]>
```

Example:

```java
public class Container<T>
```

We have specified a type parameter here. This allows us to create a variable to represent the type within our class.

We are not limited to just one type variable as we can specify many as we want. **However**, only utilise generics when necessary.

Generics in classes

We will use the generic identifier within our class so we can annotate methods and variables with it. This allows the method to be annotated with the generic variable.

Syntax:

```
[public] [static] T methodName()

[public] [static] void methodName(T parameter)

T variable

Type<T> variable
```

Generic Container

In the following example we will be writing a container that will store any type we want.

```java
public class Container<T> {

    private T element;

    public Container(T element) {
        this.element = element;
    }

    public T set(T element) {
        T oldElement = this.element;
        this.element = element;
        return oldElement;
    }

    public T get() {
        return element;
    }

    public boolean isNull() {
        return element == null;
    }
}
```

Generic Container

In the following example we will be writing a container that will store any type we want.

```
public class Container<T> {

    private T element;

    public Container(T element) {
        this.element = element;
    }

    public T set(T element) {
        T oldElement = this.element;
        this.element = element;
        return oldElement;
    }

    public T get() {
        return element;
    }

    public boolean isNull() {
        return element == null;
    }
}
```

We have defined T as our **type parameter**. This allows us to use this identifier through out our class

Generic Container

In the following example we will be writing a container that will store any type we want.

We have defined T as our **type parameter**. This allows us to use this identifier through out our class

We can use the type parameter as a **data type** for our variable.

```
public class Container<T> {

    private T element;

    public Container(T element) {
        this.element = element;
    }

    public T set(T element) {
        T oldElement = this.element;
        this.element = element;
        return oldElement;
    }

    public T get() {
        return element;
    }

    public boolean isNull() {
        return element == null;
    }
}
```

Generic Container

In the following example we will be writing a container that will store any type we want.

```
public class Container<T> {

    private T element;

    public Container(T element) {
        this.element = element;
    }

    public T set(T element) {
        T oldElement = this.element;
        this.element = element;
        return oldElement;
    }

    public T get() {
        return element;
    }

    public boolean isNull() {
        return element == null;
    }
}
```

We have defined T as our **type parameter**. This allows us to use this identifier through out our class

We can use the type parameter as a **data type** for our variable.

We have defined T as our **type parameter**. This allows us to use this identifier through out our class

Generic Container

In the following example we will be writing a container that will store any type we want.

```
public class Container<T> {

    private T element;

    public Container(T element) {
        this.element = element;
    }

    public T set(T element) {
        T oldElement = this.element;
```

We are now utilising our generic **Container** class with a **String** type.

```
public static void main(String[] args) {

    Container<String> c = new Container<String>("Hello Box!")
    String s1 = c.get();
    c.set("New string here!");
    String s2 = c.get();

    System.out.println(s1);
    System.out.println(s2);

}
```

```
> java ContainerProgram
Hello Box!
New string here!
<Program End>
```

Generic Container

In the following example we will be writing a container that will store any type we want.

```java
public class Container<T> {

    private T element;

    public Container(T element) {
        this.element = element;
    }

    public T set(T element) {
        T oldElement = this.element;
```

We are able to infer what type is going to be used through the generic identifier.

```java
public static void main(String[] args) {

    Container<String> c = new Container<String>("Hello Box!");
    String s1 = c.get();
    c.set("New string here!");
    String s2 = c.get();

    System.out.println(s1);
    System.out.println(s2);

}
```

```
> java ContainerProgram
Hello Box!
New string here!
<Program End>
```

Generic container demo

Generics

Motivation: Generics stem from the need to have type guarantees with usage. We have looked into checked and unchecked operations but it is best to understand what motivates the decision to have this degree of checking.

Without generics, we would be required to perform casting between objects (if we were to store them as an **Object** type within the data structure).

Or we will need to duplicate the class multiple times for different types.

But why?

Generics (Case 1)

Let's say we live in a world without generics. Let's go through two scenarios where we want to implement an **ArrayList** for every type we use in our program.

It becomes apparent that we will be duplicating code for every data type, we would need to create:

- **IntArrayList**
- **DoubleArrayList**
- **FloatArrayList**
- **StringArrayList**

This is awful, now we're maintaining duplicate code for a change in data type.

Generics (Case 2)

Previous problem can be resolved through casting! However, we run into issues with this.

So as we have learned from previous lectures that all **reference types** inherit from **Object**. We could treat all instances as **Object.**

Cool, we only need to write it once.

We effectively generalise all instances to an **Object[]** that will contain each **type.**

Hang on! If everything is Object, how do we know what types is stored?

Generics (Case 2)

The compiler wouldn't know what is stored in the **ArrayList**, it could be any kind of element, however this isn't the only issue.

Let's use the container as an example:

```java
public class Container {

    private Object element;

    public Container(Object element) {
        this.element = element;
    }

    public Object set(Object element) {
        Object oldElement = this.element;
        this.element = element;
        return oldElement;
    }

    public Object get() {
        return element;
    }

    public boolean isNull() {
        return element == null;
    }
}
```

Generics (Case 2)

The compiler wouldn't know what is stored in the **ArrayList**, it could be any kind of element, however this isn't the only issue.

Let's use the container as an example:

Without generics we will need to use the **Object** to refer to the any **Reference** type instance.

```java
public class Container {

    private Object element;

    public Container(Object element) {
        this.element = element;
    }

    public Object set(Object element) {
        Object oldElement = this.element;
        this.element = element;
        return oldElement;
    }

    public Object get() {
        return element;
    }

    public boolean isNull() {
        return element == null;
    }
}
```

Generics (Case 2)

The compiler wouldn't know what is stored in the **ArrayList**, it could be any kind of element, however this isn't the only issue.

Let's use the container as an example:

```java
public class Container {

    private Object element;

    public Container(Object element) {
        this.element = element;
    }

    public Object set(Object element) {
        Object oldElement = this.element;
        this.element = element;
        return oldElement;
    }

    public Object get() {
        return element;
    }

    public boolean isNull() {
        return element == null;
    }
}
```

Without generics we will need to use the **Object** to refer to the any **Reference** type instance.

So, looking at set, how would we know what type is being added?

Generics (Case 2)

The compiler wouldn't know what is stored in the **ArrayList**, it could be any kind of element, however this isn't the only issue.

Let's use the container as an example:

Without generics we will need to use the **Object** to refer to the any **Reference** type instance.

So, looking at set, how would we know what type is being added?

It is worse with calling code as any type that we call will need to handle cast this to the correct type.

```java
public class Container {

    private Object element;

    public Container(Object element) {
        this.element = element;
    }

    public Object set(Object element) {
        Object oldElement = this.element;
        this.element = element;
        return oldElement;
    }

    public Object get() {
        return element;
    }

    public boolean isNull() {
        return element == null;
    }
}
```

Generics (Case 2)

The compiler wouldn't know what is stored in the **ArrayList**, it could be any kind of element, however this isn't the only issue.

Let's use the container as an example:

```java
public class Container {

    private Object element;

    public Container(Object element) {
        this.element = element;
    }

    public Object set(Object element) {
        Object oldElement = this.element;
        this.element = element;
        return oldElement;
    }

    public Object get() {
        return element;
    }

    public boolean isNull() {
        return element == null;
    }
}
```

Without generics we will need to use the **Object** to refer to the any **Reference** type instance.

So, looking at set, how would we know what type is being added?

It is worse with calling code as any type that we call will need to handle cast this to the correct type.

A bit of trivia: In dynamically typed languages, variables refer to a **Box** (similar to our **Container class**).

Generics (Case 2)

The compiler wouldn't know what is stored in the **ArrayList**, it could be any kind of element, however this isn't the only issue.

Let's use the container as an example:

```java
public static void main(String[] args) {

    Container<String> c = new Container<String>("Hello Box!");
    String s1 = c.get();
    c.set("New string here!");
    String s2 = c.get();

    System.out.println(s1);
    System.out.println(s2);

}
```

It is worse with calling code as any type that we call will need to handle cast this to the correct type.

```java
public class Container {

    private Object element;

    public Container(Object element) {
        this.element = element;
    }

    public Object set(Object element) {
        T oldElement = this.element;
        this.element = element;
        return oldElement;
    }

    public Object get() {
        return element;
    }

    public boolean isNull() {
        return element == null;
    }
}
```

Generics (Case 2)

The compiler wouldn't know what is stored in the **ArrayList**, it could be any kind of element, however this isn't the only issue.

Let's use the container as an example:

```java
public class Container {

    private Object element;

    public Container(Object element) {
        this.element = element;
    }

    public Object set(Object element) {
        T oldElement = this.element;
        this.element = element;
        return oldElement;
    }

    public Object get() {
        return element;
    }

    public boolean isNull() {
        return element == null;
    }
}
```

```java
public static void main(String[] args) {

    Container c = new Container("Hello Box!");
    String s1 = c.get();
    c.set("New string here!");
    String s2 = c.get();

    System.out.println(s1);
    System.out.println(s2);
}
```

Removing this information is going to result in the following error:

```
Container.java:26: error: incompatible types: Object
cannot be converted to String
     String s1 = c.get();
                      ^
Container.java:28: error: incompatible types: Object
cannot be converted to String
     String s2 = c.get();
                      ^
2 errors
```

Generics (Case 2)

The compiler wouldn't know what is stored in the **ArrayList**, it could be any kind of element, however this isn't the only issue.

Let's use the container as an example:

```java
public static void main(String[] args) {

    Container c = new Container("Hello Box!");
    String s1 = c.get();
    c.set("New string here!");
    String s2 = c.get();

    System.out.println(s1);
    System.out.println(s2);

}
```

```
Container.java:26: error: incompatible types: Object
cannot be converted to String
     String s1 = c.get();
                      ^
Container.java:28: error: incompatible types: Object
cannot be converted to String
     String s2 = c.get();
                      ^
2 errors
```

Since this is a **String** variable and the return type is **Object**. The compiler cannot guarantee type correctness here.

```java
public class Container {

    private Object element;

    public Container(Object element) {
        this.element = element;
    }

    public Object set(Object element) {
        T oldElement = this.element;
        this.element = element;
        return oldElement;
    }

    public Object get() {
        return element;
    }

    public boolean isNull() {
        return element == null;
    }
}
```

Generics (Case 2)

The compiler wouldn't know what is stored in the **ArrayList**, it could be any kind of element, however this isn't the only issue.

Let's use the container as an example:

```
public static void main(String[] args) {

    Container c = new Container("Hello Box!");
    String s1 = (String) c.get();
    c.set("New string here!");
    String s2 = (String) c.get();

    System.out.println(s1);
    System.out.println(s2);

}
```

```
> javac ContainerProgram.java
>
```

We will need to **cast** the object to the correct type. This is a **runtime check** and can lead to unsafe assumptions.

```
public class Container {

    private Object element;

    public Container(Object element) {
        this.element = element;
    }

    public Object set(Object element) {
        T oldElement = this.element;
        this.element = element;
        return oldElement;
    }

    blic Object get() {
        return element;
    }

    public boolean isNull() {
        return element == null;
    }
}
```

Data Structures

We explored collections in week 4. Generics are typically used within this area as the operations and patterns involved do not differ based on the type that is used.

A linked list that contains **integers** does not have different operations to a linked list that contains **doubles**.

Usage of generics in data structures

Let’s go over the use of generics with a **Linked List** that we saw in week 4.

```java
public class LinkedList<T> {

    private Node<T> head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }

    public void add(T v) {
        if(head == null) {
            head = new Node<T>(v);
        } else {

            Node<T> current = head;
            while(current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(new Node<T>(v));
        }
        size++;
    }

    //<rest of code snipped>

    public int size() {
        return size;
    }
}
```

```java
public class Node<T> {

    private T value;
    private Node<T> next;

    public Node(T v) {
        value = v;
        next = null;
    }

    public Node<T> getNext() {
        return next;
    }

    public void setNext(Node<T> n) {
        next = n;
    }

    public T getValue() {
        return value;
    }

    public void setValue(T v) {
        value = v;
    }

}
```

Whoa! You’re getting ahead of yourself again!

Usage of generics in data structures

Let’s go over the use of generics with a **Linked List** that we saw in week 4.

```
public class LinkedList<T> {

    private Node<T> head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }

    public void add(T v) {
        if(head == null) {
            head = new Node<T>(v);
        } else {

            Node<T> current = head;
            while(current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(new Node<T>(v));
        }
        size++;
    }

    //<rest of code snipped>

    public int size() {
        return size;
    }
}
```

We typically **provide a type argument** when we use a **collection.** The the parameter is used through the class.

```
public class Node<T> {

    private T value;
                      ext;

                          {



    public Node<T> getNext() {
        return next;
    }

    public void setNext(Node<T> n) {
        next = n;
    }

    public T getValue() {
        return value;
    }

    public void setValue(T v) {
        value = v;
    }

}
```

Usage of generics in data structures

Let’s go over the use of generics with a **Linked List** that we saw in week 4.

```java
public class LinkedList<T> {

    private Node<T> head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }

    public void add(T v) {
        if(head == null) {
            head = new Node<T>(v);
        } else {

            Node<T> current = head;
            while(current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(new Node<T>(v));
        }
        size++;
    }

    //<rest of code snipped>

    public int size() {
        return size;
    }
}
```

We can use it not only as part of class and instance attributes but part of method variable.

```java
public class Node<T> {

    private T value;
                   ext;

                        {

    }

    public Node<T> getNext() {
        return next;
    }

    public void setNext(Node<T> n) {
        next = n;
    }

    public T getValue() {
        return value;
    }

    public void setValue(T v) {
        value = v;
    }

}
```

Usage of generics in data structures

Let's go over the use of generics with a **Linked List** that we saw in week 4.

```java
public class LinkedList<T> {

    private Node<T> head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }

    public void add(T v) {
        if(head == null) {
            head = new Node<T>(v);
        } else {

            Node<T> current = head;
            while(current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(new Node<T>(v));
        }
        size++;
    }

    //<rest of code snipped>

    public int size() {
        return size;
    }
}
```

Let's focus on this one

```java
public class Node<T> {

    private T value;
    private Node<T> next;

    public Node(T v) {
        value = v;
        next = null;
    }

    public Node<T> getNext() {
        return next;
    }

    public void setNext(Node<T> n) {
        next = n;
    }

    public T getValue() {
        return value;
    }

    public void setValue(T v) {
        value = v;
    }

}
```

Usage of generics in data structures

Let’s go over the use of generics with a **Linked List** that we saw in week 4.

```
public class LinkedList<T> {

    private Node<T> head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }

    public void add(T v) {
        if(head == null) {
            head = new Node<T>
        } else {

            Node<T> current = 
            while(current.getN
                current = curr
            }
            current.setNext(new Node<T>(v));
        }
        size++;
    }

    //<rest of code snipped>

    public int size() {
        return size;
    }
}
```

```
public class Node<T> {

    private T value;
    private Node<T> next;

    public Node(T v) {
        value = v;
          xt = null;


          Node<T> getNext() {
          turn next;



          void setNext(Node<T> n) {
          xt = n;



    public T getValue() {
        return value;
    }

    public void setValue(T v) {
        value = v;
    }

}
```

We can see the type parameter is specified for Node. We are able to pass the type argument to **Node**.

So if a LinkedList is defined to use String (ie LinkedList<String>), Node will also use String when utilised within this class. (ie Node<String>).

Type system is doing a lot of work!

Usage of generics in data structures

Let's go over the use of generics with a **Linked List** that we saw in week 4.

```
public class LinkedList<T> {

    private Node<T> head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }

    public void add(T v) {
        if(head == null) {
            head = new Node<T>
        } else {

            Node<T> current = head;
            while(current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(new Node<T>(v));
        }
        size++;
    }

    //<rest of code snipped>

    public int size() {
        return size;
    }
}
```

```
public class Node<T> {

    private T value;
    private Node<T> next;

    public Node(T v) {
        value = v;
        xt = null;


     Node<T> getNext() {
        turn next;


    public void setNext(Node<T> n) {
        next = n;
    }

    public T getValue() {
        return value;
    }

    public void setValue(T v) {
        value = v;
    }

}
```

Yes it is!

The compiler's type system is verifying how we are utilising the variable even in the current context.

Usage of generics in data structures

Let's go over the use of generics with a **Linked List** that we saw in week 4.

```
public class LinkedList<T> {

    private Node<T> head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }

    public void add(T v) {
        if(head == null) {
            head = new Node<T>(v);
        } else {

            Node<T> current = head;
            while(current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(new Node<T>(v));
        }
        size++;
    }

    //<rest of code snipped>

    public int size() {
        return size;
    }
}
```

```
public class Node<T> {

    private T value;
    private Node<T> next;

    public Node(T v) {
        value = v;
        next = null;
    }

    public Node<T> getNext() {
        return next;
    }

    public void setNext(Node<T> n) {
        next = n;
    }

}
```

Let's examine the usage here.
Within the **add** method, we are utilising .getNext() method of Node.

Why does the compiler know what type we are using?

Usage of generics in data structures

Let’s go over the use of generics with a **Linked List** that we saw in week 4.

```java
public class LinkedList<T> {

    private Node<T> head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }

    public void add(T v) {
        if(head == null) {
            head = new Node<T>(v);
        } else {

            Node<T> current = head;
            while(current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(new Node<T>(v));
        }
        size++;
    }

    //<rest of code snipped>

    public int size() {
        return size;
    }
}
```

```java
public class Node<T> {

    private T value;
    private Node<T> next;

    public Node(T v) {
        value = v;
        next = null;
    }

    public Node<T> getNext() {
        return next;
    }

    public void setNext(Node<T> n) {
        next = n;
    }
```

Why does the compiler know what type we are using?

Within LinkedList, we are using the type parameter on Node, therefore forcing the generic type of both LinkedList and Node to be the same.

Usage of generics in data structures

Let’s go over the use of generics with a **Linked List** that we saw in week 4.

```java
public class LinkedList<T> {

    private Node<T> head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }

    public void add(T v) {
        if(head == null) {
            head = new Node<T>(v);
        } else {

            Node<T> current = head;
            while(current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(new Node<T>(v));
        }
        size++;
    }

    //<rest of code snipped>

    public int size() {
        return size;
    }
}
```

```java
public class Node<T> {

    private T value;
    private Node<T> next;

    public Node(T v) {
        value = v;
        next = null;
    }

    public Node<T> getNext() {
        return next;
    }

    public void setNext(Node<T> n) {
        next = n;
    }
```

Since **current** can be assigned to head and is the same type, we are able to depend on getNext() returning the correct type as well.

getNext() utilises the type parameter within Node.

Type Checking

As we saw when we remove the type argument in our collection types, the compiler is **unable to check the type being used.**

Ultimately this is a **horrible idea,** if we have this feature that allows us to get a guarantee from the compiler, we would want to utilise this and in effect know the binding and let the compiler check for errors that we could make.

Generics and collections demo

What about static methods?

Generic Static Method

We can define a generic static method within our class that can operate on a set of data. Its syntax and usage is different than other instance methods as the type argument can be passed when called.

Syntax:

```
[public] static <Param0[,Param1..]> return_type methodName([,Param1..])
```

Example:

```
public static <T> T find(T needle, T[] haystack)
```

Usage:

```
Points.<AbsolutePoint>findClosestPoint(points);
```

Generic Static Method

We can define a generic static method within our class that can operate on a set of data. Its syntax and usage is different than other instance methods as the type argument can be passed when called.

Define a type parameter to be used in our method, we can also provide a type bound here

Syntax:

```
[public] static <Param0[,Param1..]> return_type methodName([,Param1..])
```

Example:

```
public static <T> T find(T needle, T[] haystack)
```

Usage:

```
Points.<AbsolutePoint>findClosestPoint(points);
```

Generic Static Method

We can define a generic static method within our class that can operate on a set of data. Its syntax and usage is different than other instance methods as the type argument can be passed when called.

Define a type parameter to be used in our method, we can also provide a type bound here

Syntax:

```
[public] static <Param0[,Param1..]> return_type methodName([,Param1..])
```

Type parameter can be used as return type and method parameter type

Example:

```
public static <T> T find(T needle, T[] haystack)
```

Usage:

```
Points.<AbsolutePoint>findClosestPoint(points);
```

Generic Static Method

We can define a generic static method within our class that can operate on a set of data. Its syntax and usage is different than other instance methods as the type argument can be passed when called.

Define a type parameter to be used in our method, we can also provide a type bound here

Syntax:

```
[public] static <Param0[,Param1..]> return_type methodName([,Param1..])
```

Type parameter can be used as return type and method parameter type

Example:

```
public static <T> T find(T needle, T[] haystack)
```

Since the method can be used without an instance, the type argument needs to known

Usage:

```
Points.<AbsolutePoint>findClosestPoint(points);
```

Generic Static Method

We can define a generic static method within our class that can operate on a set of data. Its syntax and usage is different than other instance methods as the type argument can be passed when called.

Define a type parameter to be used in our method, we can also provide a type bound here

Syntax:

```
[public] static <Param0[,Param1..]> return_type methodName([,Param1..])
```

Type parameter can be used as return type and method parameter type

Example:

```
public static <T> T find(T needle, T[] haystack)
```

Since the method can be used without an instance, the type argument needs to known

Usage:

```
Points.<AbsolutePoint>findClosestPoint(points);
```

We pass the type argument to the static method when we want to invoke it. This type argument can be used to ensure a method parameter has a type association

What happens to our type information at runtime?

Type Erasure

Generics provide a **compile-time** guarantee and will prevent errors in our code prior to runtime. However, during runtime, type information is erased. You can consider the case where the compiler transforms all our get and add operations with **casts**.

This data removal can make it very difficult to debug. Since there is a lack of information in regards to the type of data that is being stored.

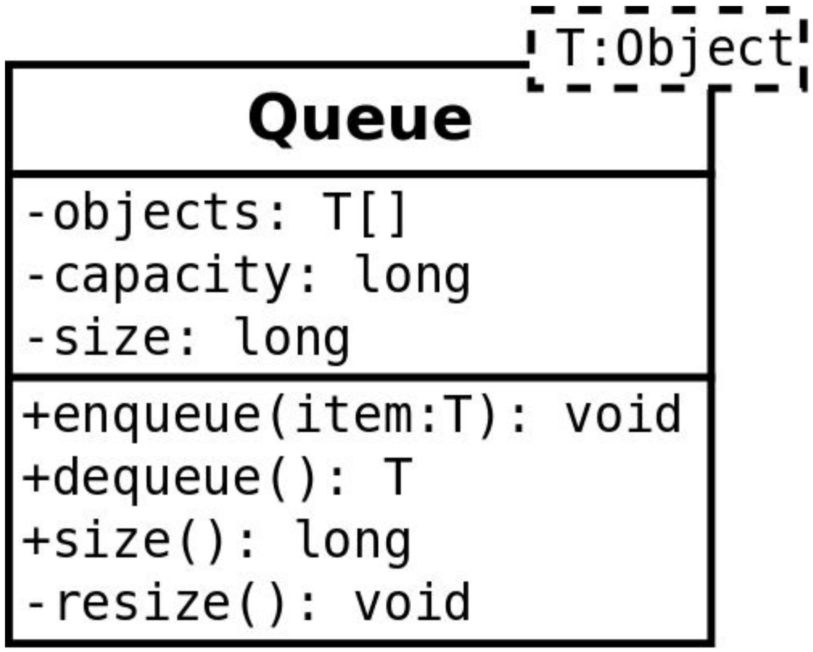
Why is it removed?
By design, instead of duplicating the code for each, it will perform the checks at compile time and perform the casts accordingly. This is a compromise to maintain backward compatibility with existing java source code.

Type Erasure, Oracle (https://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html#jls-4.6)

UML Template Class

UML modelling language defines a class with generics as a **Template Class**.

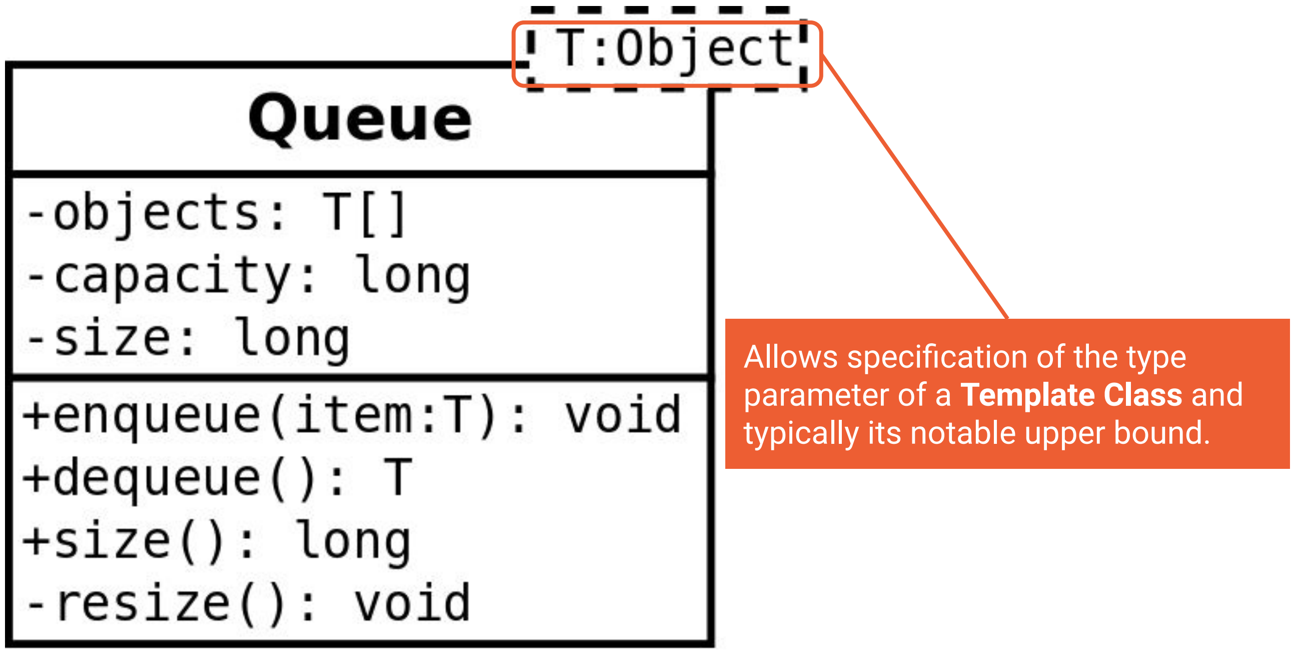
Within the visualisation it will contain annotation of the type parameter.



UML Template Class

UML modelling language defines a class with generics as a **Template Class**.

Within the visualisation it will contain annotation of the type parameter.

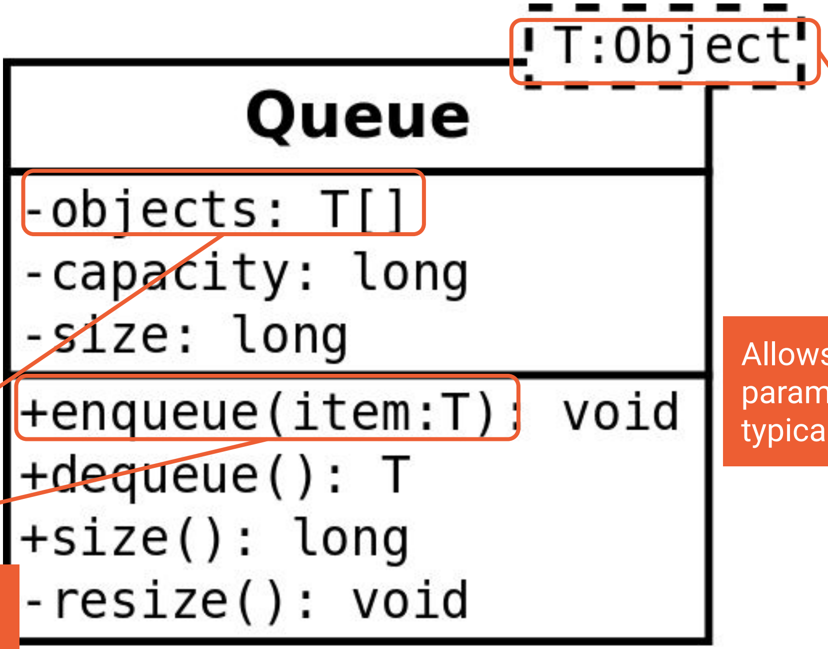


Allows specification of the type parameter of a **Template Class** and typically its notable upper bound.

UML Template Class

UML modelling language defines a class with generics as a **Template Class**.

Within the visualisation it will contain annotation of the type parameter.



Allows specification of the type parameter of a **Template Class** and typically its notable upper bound.

Similar to our class definition, we use the type parameter for attributes and methods.

If there is time, let’s build one more data structure

See you next time!