# CSC 325 Adv Data Structures

## Lecture 6

## Generics

Languages like Java have what's called a "static type system". This means that data types must be figured out at compile time. Furthermore, this means that the compiler must know the data type of every variable/object reference in your code. On the surface, this seems to force us into creating data structures where the actual data used type must be "hard coded" (i.e. specified directly). In fact, without the concept of generics, this is exactly what would happen!

Imagine having to create a stack of integers. You write all your code with integers in mind and make sure that every item in the stack is an integer. A fellow developer walks up to you and says that they want a stack for strings as well. What do you do? You could copy and paste the entire class and then find and replace the word "Integer" with "String" everywhere. But you think for a second. The logic of the stack really has nothing to do with integers! In fact, the same exact logic could work with any data type, even though you were thinking about integers when you wrote it. So why are you creating two classes that are exactly the same in logic and only differ in data type being stored? Instead, let's use generics.

Consider the following two classes:

```
class StackOfInts {
  private Integer[] data;

  public StackOfInts() { ... }
  public void push(Integer datum[1]) { ... }
  public Integer pop() { ... }
  public Integer peek() { ... }
  public int size() { ... }
}
```

```
class StackOfStrings {
  private String[] data;

  public StackOfStrings() { ... }
  public void push(String datum) { ... }
  public String pop() { ... }
  public String peek() { ... }
  public int size() { ... }
}
```

The implementation details aren't important and so they have been omitted. The two classes above are identical in functionality. The implementation of each function in each class is identical. In computer science we have a concept known as D.R.Y. which stands for Don't Repeat Yourself. This is definitely a violation of that principal. When you repeat yourself in code and therefore duplicate your logic, you run the big risk of introducing bugs. I know from first-hand experience just how easy it is for duplicate code to get out of sync, where a change to one doesn't make it to the other. It is incredible easy to forget to make a change in all places that they change applies to. So, the goal is to make it apply to AS FEW places as possible! By reusing logic in multiple places (for example, creating a function and calling it in multiple places instead of rewriting the code every time), we end up having far fewer places to update when a change occurs (and changes will absolutely occur!). So, let's follow this DRY principal and let's make this a generic class, thereby using the same class for both Integers and Strings:

[1] Yes the singular form of "data" is in fact "datum" (src: https://www.merriam-webster.com/dictionary/datum)

```
class Stack<T> {
  private T[] data;

  public Stack() { }
  public void push(T datum) { ... }
  public T pop() { ... }
  public T peek() { ... }
  public int size() { ... }
}
```

We introduced a new variable called “T” here. This variable is also known as a “type parameter” since it stores a data type. If the above code confuses you, just replace “T” with “String” and look at the class again. We really haven’t changed anything. The only difference is that the data type is now up to the object to decide. How does the object decide this? Like so:

```
// stack of integers
Stack<Integer> stackWithInts = new Stack<Integer>();

// stack of strings (using "diamond notation" on instantiation)
Stack<String> stackWithStrs = new Stack<>();
```

Each object when it is created is allowed to specify the data type it wants to use. Different objects can specify different data types. We could literally have thousands of objects all with different data types and yet we only had to write a single stack class! This definitely follows the DRY principal very well.

Why did I use “Integer” instead of “int”? Well, remember that “int” is a “primitive” data type and therefore it is considered to be a “value type”. This means that it can’t participate in generics. We can only use reference types in generics. So instead of using int, we use Integer. Not really much of a difference though, as in the end we are still storing whole numbers.

Why can’t it participate in generics? Good question! Actually, if you look at C# or C++, a simple primitive int can absolutely be used with generics:

**C#**

```
public class Stack<T> {
  private T[] data;

  public Stack() { }
  public void Push(T datum) { ... }
  public T Pop() { ... }
  public T Peek() { ... }
  public int Size() { ... }
}

public class Program {
  public static void Main(string[] args) {
    Stack<int> stack = new Stack<int>();
  }
}
```

**C++**

```
template <typename T>
class Stack {
  private: T* data;

  public:
    Stack();
    void push(T datum);
    T pop();
    T peek();
    int size();
};

int main() {
  Stack<int>* stack = new Stack<int>();
}
```

As you can see, a simple "int" which is a value type in both languages will actually work as a generic parameter. Now, full disclosure, the code above will not compile cleanly because I have left out some details. A full stack implementation would introduce unnecessary details and would obfuscate the point I am trying to make. The point is, there are no issues from creating a generic class and using a primitive data type as the type parameter.

Ok, so why does Java have this restriction when other languages do not? Well, this is due to a limitation in Java's generic type system. This limitation actually has a name. It is called "type erasure". Essentially, when Java first introduced generics in Java 5 (in 2004), it didn't want to break existing code. Java came out in 1995 and almost a decade later, plenty of programs were written in Java. It had gained popularity and was being used a good bit. So, any change to the language had to be taken seriously as it could break backwards compatibility. So, for better or for worse, Java decided to introduce generics only as a "compiler trick". Basically, the compiler knows about generics, but the runtime doesn't! This is accomplished by the compiler adding explicit casts, type checking things, and essentially hiding any form of generics in the compiled byte code. Generics in Java are really just syntactic sugar! To see what I mean, this is what a class looks like before and after the compiler hides generics[2]:

| **With generics** | **After type erasure** |
|---|---|
| `class Stack<T> {`<br>`  private T[] data;`<br><br>`  public Stack() { }`<br>`  public void push(T datum) { ... }`<br>`  public T pop() { ... }`<br>`  public T peek() { ... }`<br>`  public int size() { ... }`<br>`}` | `class Stack {`<br>`  private Object[] data;`<br><br>`  public Stack() { }`<br>`  public void push(Object datum) { ... }`<br>`  public Object pop() { ... }`<br>`  public Object peek() { ... }`<br>`  public int size() { ... }`<br>`}` |

As you can see, Java just uses the super class "Object" instead of the generic type parameter. If the generic type is desired, Java can insert casts:

| **With generics** | **After type erasure** |
|---|---|
| `Stack<Integer> stack = new Stack<>();`<br>`Integer val = stack.pop();` | `Stack stack = new Stack();`<br>`Integer val = (Integer)stack.pop();` |

Now Java just has to replace "T" with the data type and the data will be casted to that type and returned from the pop function. When the code is actually executed, "T" will never be seen.

This is the reason why if we create two classes with different generics, Java will see them as the same exact thing:

[2] Here is a good article explaining this further https://www.baeldung.com/java-type-erasure

**In File: DataStorage.java**

```
class DataStorage<T> {
  private T data;
}

class DataStorage<S,T> {
  private S data1;
  private T data2;
}
```

**On Terminal attempting to compile:**

```
$ javac DataStorage.java
DataStorage.java:5: error: duplicate class: DataStorage
class DataStorage<S,T> {
^
1 error
```

Java actually sees the above classes as the exact same class even though they differ in their type parameters. This is because the compiler, after type erasure, sees the classes like this:

```
class DataStorage {
  private Object data;
}

class DataStorage {
  private Object data1;
  private Object data2;
}
```

This is definitely a limitation of Java's generics[3]. This also explains why we can't use "int" in generics. This is because ints do not inherit from Object! We can only use data types that inherit from Object since Object is what the generic type erases into. All reference types inherit from Object, so any reference types are fair game to use.

So, does this limitation really matter? Sometimes yes, but mostly no. We still have a lot of power with Java's generics and the main mission of writing the logic once and reusing it with different data types is absolutely accomplished here.

[3] Curiously enough, C++ doesn't allow this either, but C# does. You might have to Google why.

**Generics on methods only:**

We can actually use generics just at the method level without declaring it for the entire class. Consider the following code:

```
public class Scratch {
  private static<T> T foo(T arg) {
    return arg;
  }

  private <U> U bar(U arg) {
    return arg;
  }

  public static void main(String[] args) {
    System.out.println(Scratch.<Integer>foo(45));
    Scratch s = new Scratch();
    System.out.println(s.<String>bar("hello"));
  }
}
```

Here we have the method foo which is generic with type parameter T. Note that T is only available for foo to use and not the entire class. The method bar has its own generic type parameter U. The syntax is a bit different here compared to a class. Whereas with a class, we put the generic type AFTER the classname, here we put the generic type BEFORE the method name. Example:

```
class Scratch<T> {         // goes after the class name
private <U> void baz() {   // goes before the method name
```

Actually, on methods the type parameter goes before the return type of the method too. The thought process here is that the generic type parameter must be defined before any spot where it can be used, and the first spot it can be used is in the return type (look at foo and bar above).

To call a generic method, you put the type parameter immediately after the dot operator:

```
    System.out.println(Scratch.<Integer>foo(45));
    Scratch s = new Scratch();
    System.out.println(s.<String>bar("hello"));
```

This is also backwards from a generic class. If Scratch was generic, we would do the following:

```
    Scratch<Float> s = new Scratch<Float>();
```

So, to instantiate a generic class, we put the type parameter AFTER the class name (much like at class declaration). To call a generic method, we put the type parameter BEFORE the method name (again much like at method declaration).