

Object-Oriented Programming

CSCI-UA 0470-001

Class 12

Instructor: Randy Shepherd

Parametric Polymorphism

- *Parametric polymorphism* is a way to make a language more expressive, while still maintaining *static type-safety*.
- A function or a class can be written such that it can handle *values* identically without depending on their *type*.
- Such functions and data types are called *generic* and form the basis of *generic programming*.

Generic Programming

- *Generic programming* is a style of programming in which algorithms are written in terms of *types to-be-specified-later*
- These types are then instantiated when needed for specific types, provided as parameters
- Moreover, generics allow you to abstract over types.

Those are words



Java Generics

Java Generics

- Java Generics is an example of a generic programming language construct.
- Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods.
- Like formal parameters used in method declarations, type parameters provide a way for you to re-use the same code with different inputs.

Motivation

- We have a compiler for a reason. It finds bugs for us. Runtime bugs are generally much more difficult to solve.
- Similarly, generics give us another way of expressing some constraints to the compiler and therefore to prevent certain types of bugs.
- Generic programming also allows us in some cases to write less code, which is always good thing.

Benefits

- *Stronger type checks at compile time.*
 - The compiler applies strong type checking to generic code and issues errors if the code violates type safety.
- *Elimination of casts.*
 - Generic data structures can be inserted and extracted from without casting
- *Enabling programmers to implement generic algorithms.*
 - Programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe

You've seen this...

- Common examples are found in the Collections library.
- Have you seen something like this before?

```
List<String> list = new ArrayList<String>();
```

Example: Box Class

- A simple box class

```
1  public class Box {  
2      private Object object;  
3  
4      public void set(Object object) { this.object = object; }  
5      public Object get() { return object; }  
6  }
```

- Since its methods accept or return an Object, you are free to pass in whatever you want, provided that it is not one of the primitive types.
- There is no way to verify, at compile time, how the class is used.

Generics Syntax

- A generic class is defined with the following format:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

- The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the type parameters (also called type variables) T1, T2, ..., and Tn.
- A type variable can be any *non-primitive* type you specify: any class type, any interface type, any array type, or even another type variable.

Generic Box

- To update the Box class to use generics, you create a generic type declaration by changing the code "public class Box" to "public class Box<T>".
- This introduces the type variable, T, that can be used anywhere inside the class.

```
1  public class Box<T> {  
2      // T stands for "Type"  
3      private T t;  
4  
5      public void set(T t) { this.t = t; }  
6      public T get() { return t; }  
7  }
```

Use in Java Collections

```
1 // Without Generics
2 ArrayList l1 = new ArrayList();
3 l1.add("hello");
4 String s1 = (String) l1.get(0); // Note the cast
5
6 // With Generics
7 ArrayList<String> l2 = new ArrayList <String>();
8 l2.add("hello");
9 String s2 = l2.get(0);
```

- Note the cast on line 4
- The programmer knows what kind of data has been placed into the list.
- The compiler can only guarantee that an Object will be returned.
- To ensure the assignment to a variable of type String is type safe, the cast is required.

Use in Java Collections

```
1 // Without Generics
2 ArrayList l1 = new ArrayList();
3 l1.add("hello");
4 String s1 = (String) l1.get(0); // Note the cast
5
6 // With Generics
7 ArrayList<String> l2 = new ArrayList <String>();
8 l2.add("hello");
9 String s2 = l2.get(0);
```

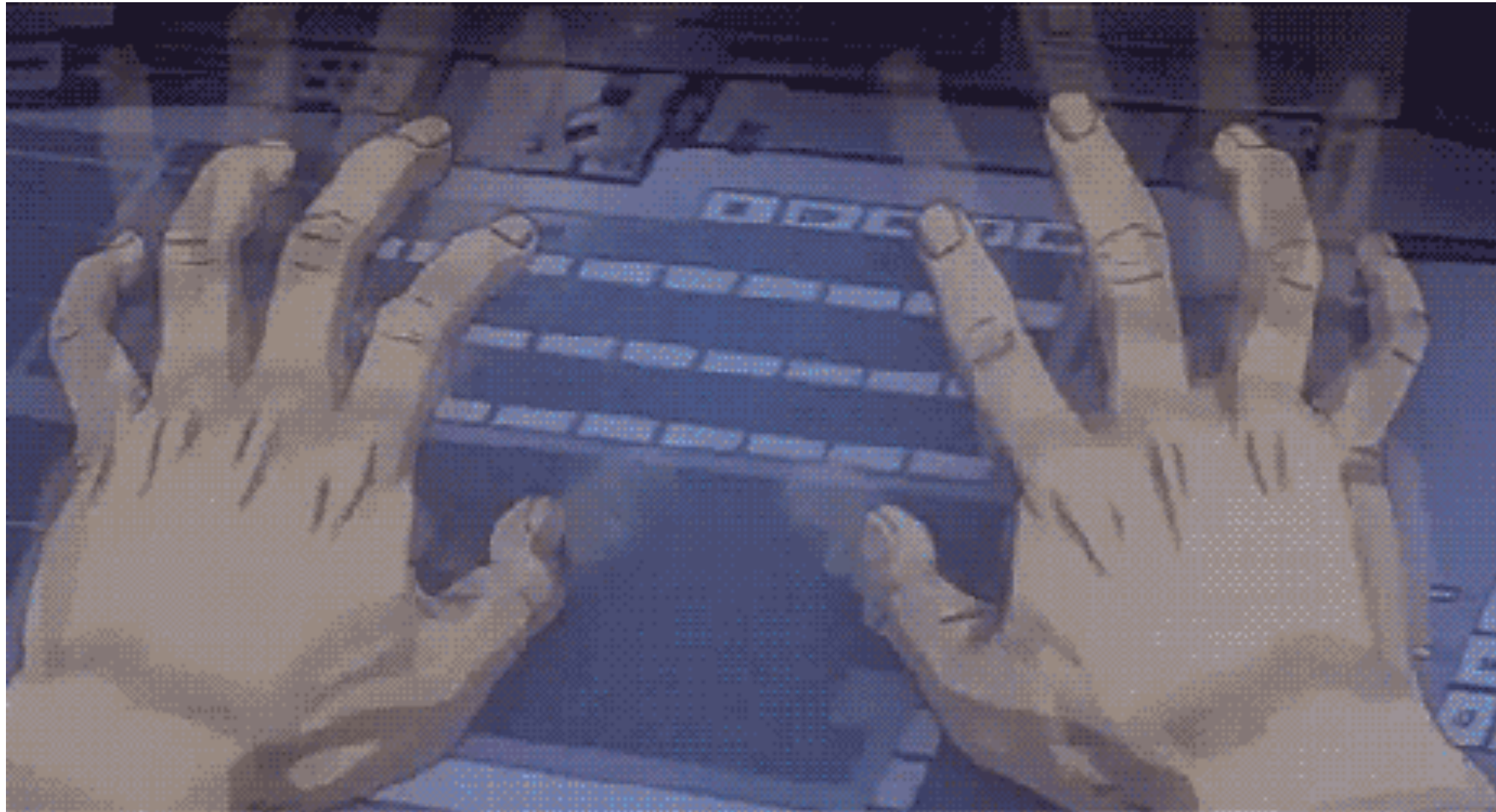
- There is the possibility of a runtime error, since the programmer may be mistaken.
- Enter Generics. Notice the type declaration on line 7.
- It specifies that this is a List of of String, written `ArrayList<String>`.
- Why? Type-safety! The compiler can now catch certain errors.

Stack Implementation

```
1 class NonGenericStack {
2     private ArrayList contents = new ArrayList();
3
4     public void push(Object element) {
5         contents.add(element);
6     }
7
8     public Object pop() {
9         int top = contents.size()-1;
10        Object result = contents.get(top);
11        contents.remove(top);
12        return result;
13    }
14 }
15
16 // Somewhere else in the codebase...
17
18 NonGenericStack ngStack = new NonGenericStack();
19 ngStack.push("String");
20 ngStack.push(123);
21 String str1 = (String) ngStack.pop();
```

```
1 class GenericStack<E> {
2     private ArrayList<E> contents = new ArrayList<E>();
3
4     public void push(E element) {
5         contents.add(element);
6     }
7
8     public E pop() {
9         int top = contents.size()-1;
10        E result = contents.get(top);
11        contents.remove(top);
12        return result;
13    }
14 }
15
16 // Somewhere else in the codebase...
17
18 // This is a compile error
19 GenericStack<String> gStack = new GenericStack<String>();
20 gStack.push("String");
21 gStack.push(123); // <-- here
22 String str2 = (String) ngStack.pop();
```


Lets look at some code...



<https://github.com/nyu-oop-fall16/generics>

Generic Implementation

- How are generics implemented by the Java compiler?
 - Type erasure -- removes generic type information from source code, adds casts as needed, and produces byte code
 - Essentially, the type parameters go away and the compiler generates exactly the same naive code we saw in our RawList.java
- Why is this the way the compiler works?
 - Backwards compatibility. this approach did not require the JVM to be changed

Generic Caveats

- No generic arrays
- No instantiations of generic types
 - i.e. `E e = new E ();` where `E` is a type parameter.
- No primitive types, which require wrapper classes. `int => Integer`
 - There is “auto-boxing” to ease conversions from primitive types to boxed types and back again, but this has a performance cost.
- Type information lost at runtime.
 - Can be very annoying, Especially when using reflection.

Quick Word on Reflection

- A Java API to examine or modify the runtime behavior of applications running in the Java virtual machine.
- For example, say you have an object of an unknown type in Java and you would like to call a 'doSomething' method on it if one exists.

```
1 | Method method = foo.getClass().getMethod("doSomething", null);  
2 | method.invoke(foo, null);
```

C++ Templates

C++ Templates

- Templates are the C++ analog to Java Generics
- They are somewhat more flexible
- As with Java, templates can be made for classes and functions, and said templates allow classes and functions to intake many different types
- Different than Java, they work with primitives, you can instantiate type parameters and there are generic *arrays*.

C++ Templates

- As stated Generics are implemented by *erasure*
- C++ templates are implemented by *expansion*
- Each instance of a template with a new type is generated and compiled
- Moreover, if you have lists of strings, ints and doubles, you have 3 different classes generated.
- This could lead to *code bloat*. In practice this is usually fine.

Template Example

- Class definition preceded by template keyword followed by type parameter.
- Similar syntax precedes *implementation* of functions
- In both cases, type param name used in definition where concrete type would appear.

```
1  template <class T>
2  class mypair {
3      T a, b;
4      public:
5          mypair (T first, T second):
6              a(first), b(second) {}
7          T getmax ();
8  };
9
10 template <class T>
11 T mypair<T>::getmax ()
12 {
13     T retval;
14     retval = a>b? a : b;
15     return retval;
16 }
17
18 int main () {
19     mypair <int> intPair (100, 75);
20     cout << intPair.getmax() << endl;
21
22     mypair <string> stringPair ("a", "b");
23     cout << stringPair.getmax() << endl;
24
25     return 0;
26 }
```

Template Example

- At construction, the caller provides the type parameters after the typename and before the identifier.
- Any type that supports operations performed on it will work just fine.
- What does this code output?

```
1  template <class T>
2  class mypair {
3      T a, b;
4  public:
5      mypair (T first, T second):
6          a(first), b(second) {}
7      T getmax ();
8  };
9
10 template <class T>
11 T mypair<T>::getmax ()
12 {
13     T retval;
14     retval = a>b? a : b;
15     return retval;
16 }
17
18 int main () {
19     mypair <int> intPair (100, 75);
20     cout << intPair.getmax() << endl;
21
22     mypair <string> stringPair ("a", "b");
23     cout << stringPair.getmax() << endl;
24
25     return 0;
26 }
```


Template Example

- Key idea
 - Data structures can be created and used to handle many different types without declaring separate classes for each.
- Anybody see an application for our translator?

```
1  template <class T>
2  class mypair {
3      T a, b;
4      public:
5          mypair (T first, T second):
6              a(first), b(second) {}
7          T getmax ();
8  };
9
10 template <class T>
11 T mypair<T>::getmax ()
12 {
13     T retval;
14     retval = a>b? a : b;
15     return retval;
16 }
17
18 int main () {
19     mypair <int> intPair (100, 75);
20     cout << intPair.getmax() << endl;
21
22     mypair <string> stringPair ("a", "b");
23     cout << stringPair.getmax() << endl;
24
25     return 0;
26 }
```

Declaring and Defining Templates

- Class templates must be *declared* in the header file
- The *compiler* creates the template for that type, replacing all instances of T with the actual type.
 - Very different than Java!
- Class templates must also be *defined* in the header file.

Template Specialization

- Specialized versions of class templates can be created for specific types
- Defined in .cpp (implementation) files because they don't need to be instantiated by the compiler
- Motivation, for a specific type we want to override the behavior of a certain method.

```
1  template<class T>
2  class mycontainer {
3      T element;
4  public:
5      mycontainer(T arg) : element(arg) { }
6
7      T increase() { return ++element; }
8  };
9
10 // class template specialization for char
11 template<>
12 class mycontainer<char> {
13     char element;
14 public:
15     mycontainer(char arg) : element(arg) { }
16
17     char increase() {
18         if ((element >= 'a') && (element <= 'z'))
19             element += 'A' - 'a';
20         return element;
21     }
22 };
23
24 int main() {
25     mycontainer<int> myint(7);
26     mycontainer<char> mychar('j');
27     cout << myint.increase() << endl;
28     cout << mychar.increase() << endl;
29     return 0;
30 }
```

Moar Code Plz



<https://github.com/nyu-oop-fall16/templates>

Templates in the Project

- We can use templates to make our translation of arrays much easier.
- Much better than implementing a array-class-per-type! (Especially since we cannot use inheritance).
- This is the **only** place we may use templates.

Templates in the Project

- Array templates along with a few specializations are implemented for you in `java-lang-3`
- However, *you* must copy it to your translator project.
- But before you do that, we will have an in-class exercise next week on this subject.

Arrays in java-lang-3

- Although you cannot see it here, we've moved our array into the `__rt` namespace
- We have our forward declarations, those are templates as well.

```
1  template <typename T>
2  struct Array;
3
4  template <typename T>
5  struct Array_VT;
6
7  template <typename T>
8  struct Array {
9      Array_VT<T>* __vptr;
10     const int32_t length;
11     T* __data;
12
13     // The constructor (defined inline).
14     Array(const int32_t length)
15         : __vptr(&__vtable), length(length), __data(new T[length]()) {
16     }
17
18     // The function returning the class object for the array.
19     static java::lang::Class __class();
20
21     // The vtable for the array.
22     static Array_VT<T> __vtable;
23 };
24
25 // The vtable for arrays.
26 template <typename T>
27 Array_VT<T> Array<T>::__vtable;
28
29 // But where is the definition of __class()???
```

Arrays in java-lang-3

- Our data layout looks basically the same, with the exception of utilizing the type parameter.
- Note that the constructor is defined inline. Since for template 'implementation' should go in the headers.

```
1  template <typename T>
2  struct Array;
3
4  template <typename T>
5  struct Array_VT;
6
7  template <typename T>
8  struct Array {
9      Array_VT<T>* __vptr;
10     const int32_t length;
11     T* __data;
12
13     // The constructor (defined inline).
14     Array(const int32_t length)
15         : __vptr(&__vtable), length(length), __data(new T[length]()) {
16     }
17
18     // The function returning the class object for the array.
19     static java::lang::Class __class();
20
21     // The vtable for the array.
22     static Array_VT<T> __vtable;
23 };
24
25 // The vtable for arrays.
26 template <typename T>
27 Array_VT<T> Array<T>::__vtable;
28
29 // But where is the definition of __class()???
```


Arrays in java-lang-3

- As previously stated, all template 'implementation' code should go in header.
- So on line 26 we see our instantiation for the `__vtable` member.
- Where is the definition of `__class()`? Why is it not here?

```
1  template <typename T>
2  struct Array;
3
4  template <typename T>
5  struct Array_VT;
6
7  template <typename T>
8  struct Array {
9      Array_VT<T>* __vptr;
10     const int32_t length;
11     T* __data;
12
13     // The constructor (defined inline).
14     Array(const int32_t length)
15         : __vptr(&__vtable), length(length), __data(new T[length]()) {
16     }
17
18     // The function returning the class object for the array.
19     static java::lang::Class __class();
20
21     // The vtable for the array.
22     static Array_VT<T> __vtable;
23 };
24
25 // The vtable for arrays.
26 template <typename T>
27 Array_VT<T> Array<T>::__vtable;
28
29 // But where is the definition of __class()???
```

Arrays in java-lang-3

- New vtable code, “now even more hard-to-readness!”

(don't you just love C++?)

- We have a typedef here to alias a pointer to an array of containing some type.
- Otherwise, really not much different than what we have seen in the past.

```
1  template <typename T>
2  struct Array_VT {
3      typedef Array<T>* Reference;
4
5      java::lang::Class __isa;
6      int32_t (*hashCode)(Reference);
7      bool (*equals)(Reference, java::lang::Object);
8      java::lang::Class (*getClass)(Reference);
9      java::lang::String (*toString)(Reference);
10
11     Array_VT()
12         : __isa(Array<T>::__class()),
13           hashCode((int32_t (*)(Reference))
14                   &java::lang::__Object::hashCode),
15           equals((bool (*)(Reference, java::lang::Object))
16                 &java::lang::__Object::equals),
17           getClass((java::lang::Class (*)(Reference))
18                   &java::lang::__Object::getClass),
19           toString((java::lang::String (*)(Reference))
20                   &java::lang::__Object::toString) {
21     }
22 };
```

Arrays in java-lang-3

- Here is the specialization for and array of Objects
- Basically just an implementation of the `__class()` method.
- Why?
- Hmm.. that `__Class` constructor look a little different, I think.

```
1 // Template specialization for arrays of objects.
2 template<>
3 java::lang::Class Array<java::lang::Object>::__class() {
4     static java::lang::Class k =
5         new java::lang::__Class(literal("[Ljava.lang.Object;"),
6                                 java::lang::__Object::__class(),
7                                 java::lang::__Object::__class());
8     return k;
9 }
```

Arrays in java-lang-3

- Here is the specialization for and array of ints
- And now that I see this..there are definitely some changes to the Class constructor.
- Any guesses as to why?

```
1 // Template specialization for arrays of ints.
2 template<>
3 java::lang::Class Array<int32_t>::__class() {
4     static java::lang::Class ik =
5         new java::lang::__Class(
6             __rt::literal("int"),
7             (java::lang::Class)__rt::null(),
8             (java::lang::Class)__rt::null(),
9             true
10        );
11
12     static java::lang::Class k =
13         new java::lang::__Class(
14             literal("[I"),
15             java::lang::__Object::__class(),
16             ik);
17
18     return k;
19 }
```

Arrays in java-lang-3

- Note line 4
- Since we don't have a class representing an int, we have to construct the class.

```
1 // Template specialization for arrays of ints.
2 template<>
3 java::lang::Class Array<int32_t>::__class() {
4     static java::lang::Class ik =
5         new java::lang::__Class(
6             __rt::literal("int"),
7             (java::lang::Class)__rt::null(),
8             (java::lang::Class)__rt::null(),
9             true
10        );
11
12     static java::lang::Class k =
13         new java::lang::__Class(
14             literal("[I"),
15             java::lang::__Object::__class(),
16             ik);
17
18     return k;
19 }
```


In-class exercise

- Next class we will do an in-class exercise to get you familiar with templates and the new arrays in java-lang
- In the meantime, work on your translator.
- The new java-lang version with template-based arrays is on Github (v3)

