

The Joy of Programming
with Bob Ross

OBJECT-ORIENTED PROGRAMMING

Generic Programming

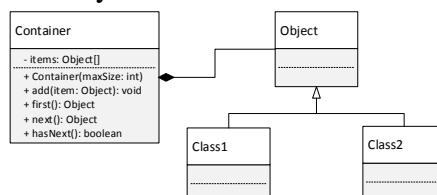
Lecture #6

Motivation

- Define software components with **type parameters**
 - A sorting algorithm has the same structure, regardless of the types being sorted
 - Stack primitives have the same semantics, regardless of the objects stored on the stack
- Most common use
 - Algorithms on containers – updating, iteration, search
- Existing implementations
 - C – macros (textual substitution)
 - Ada – generic units and instantiations
 - **OO languages (C++, Java, C#) – templates / generics**

Example – design

- Let's have a class that implements a container of various objects (Container)
 - For now the representation of the list (array, linked list, tree) is not important
 - We just want to have a **list of any type of objects**
- Using general class hierarchy and polymorphic reference we can implement commonly usable class Container



Example – implementation

- Implementation of class Container (let's use static array)

```
public class Container {  
    private Object[] items;  
    private int size;  
    private int iterator;  
    public Container(int maxSize) {  
        this.items = new Object[maxSize];  
        this.size = this.iterator = 0;  
    }  
    public void add(Object item) {  
        if (this.size < this.items.length)  
            this.items[this.size++] = item;  
    }  
}
```

Example – implementation

```
public Object first() {  
    this.iterator = 0;  
    return next();  
}  
public Object next() {  
    if (this.iterator < this.size)  
        return this.items[this.iterator++];  
    return null;  
}  
public boolean hasNext() {  
    if (this.iterator < this.size)  
        return true;  
    return false;  
}  
}
```

Example – testing usage

- Now let's test our construction

```
// Create new container
Container c = new Container(100);

// Add some items of type Class1 and Class2
c.add(new Class1());
c.add(new Class2());

// Get and use items from the container
for (Object o = c.first(); c.hasNext(); o = c.next()) {
    // Change type of the object to its original class
    if (o instanceof Class1) {
        Class1 myItem = (Class1) o;
        // Do something with myItem of Class1
    }
    else if (o instanceof Class2) {
        Class2 myItem = (Class2) o;
        // Do something with myItem of Class2
    }
    else {
        // Not implemented
    }
}
```

•All classes are derived from class Object

•The Container object can contain objects of any type

•Are programmers able to cover all castings in if-else constructs?

It is not very reusable solution!

Example – need for parametrized types?

- The Container is really universal implementation
 - Uses polymorphic reference to implement array of any objects
- Casting is checked at **run-time** so we are never sure about correct types
- Programmer can make a mistake and add object of type which is not allowed in the application
 - E.g. our example is using only Class1 and Class2, other classes are not implemented
- Can we make container where we specify what type of items are contained?
 - This can omit casting problem for programmers

Generic programming

- **Programming paradigm for developing efficient, reusable software libraries**
 - For example STL in ANSI/ISO C++
- The idea of generic programming process
 - **Lifting:** Providing suitable abstractions so that a single, generic algorithm can cover many concrete implementations
 - Focuses on finding commonality among similar implementations of the same algorithm
 - **Concepts:** Describe a set of abstractions, each of which meets all of the requirements of a concept
 - Concepts that emerge tend to describe the abstractions within the problem domain in some logical way
- The output of the generic programming process is not just a generic, reusable implementation, but a **better understanding of the problem domain**

Object-oriented principles for generic programming

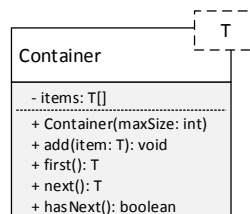
- Abstraction and encapsulation
- Subtyping and subclassing
- Subtype polymorphism
- **Templates / generics**
 - Classes, interfaces and functions with type parameters

Type as a parameter of a class

- The generic class (or interface) can be defined with a type parameter(s)
 - **Parametric polymorphism**
- Generic class must specify **generic behavior** that can work with any substituted type
- Generic class defines
 - Generic attributes – where attributes are without concrete type
 - Generic methods – methods that does not specify concrete return type or their parameters can have different types

Example – redesign with generics

- Let's have the same example – a class that implements a container of various objects (Container)
- We will use generic class with type parameter



Example – reimplementation with generics

- Reimplementation of class Container with generics

```
public class Container<T> {
    private T[] items;
    private int size;
    private int iterator;
    public Container(Class<T> c, int maxSize) {
        this.items = (T[]) new Array.newInstance(c, maxSize);
        this.size = this.iterator = 0;
    }
    public void add(T item) {
        if (this.size < this.items.length)
            this.items[this.size++] = item;
    }
}
```

Example – reimplementation with generics

```
public T first() {
    this.iterator = 0;
    return next();
}
public T next() {
    if (this.iterator < this.size)
        return this.items[this.iterator++];
    return null;
}
public boolean hasNext() {
    if (this.iterator < this.size)
        return true;
    return false;
}
}
```

Example – testing usage with generics

- Now let's use our new construction with generics

```
// Create new container of Class1 type of items
Container<Class1> c = new Container<Class1>(Class1.class, 100);

// Add some items of type Class1
c.add(new Class1());
c.add(new Class1());

// Get and use items from the container
for (Class1 o = c.first(); c.hasNext(); o = c.next()) {
    // Do something with Class1 item
    // ...
}
```

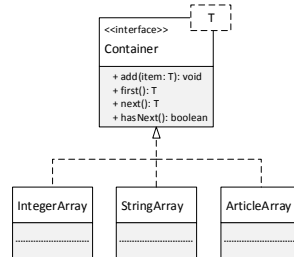
Example – conclusions

- Here we say that `Container` is a generic class that takes a type parameter `T`
 - `Class1` is substituted in the test case
- We no longer need to cast to a `Class1` since the `first()` and `next()` methods would return a reference to an object of a specific type
 - `Class1` in this case
- If we were to assign an extracted element to a different type, the error would be at **compile-time** instead of **run-time**
 - This early static checking increases the type safety of the language

Generic interface

- In Java also interfaces can be defined with a type parameters
- Let's rebuild our example another way – using generic interface

```
public interface Container<T> {
    public void add (T item);
    public T first();
    public T next();
    public boolean hasNext();
}
```



- This interface can be implemented by any class to add generic container behavior

Generic interface

- Implementation of `StringArray`

```
public class StringArray implements Container<String> {
    private String[] items;
    private int size;
    private int iterator;
    public StringArray(int maxSize) {
        this.items = new String[maxSize];
        this.size = this.iterator = 0;
    }
    public void add(String item) {
        if (this.size < this.items.length)
            this.items[this.size++] = item;
    }
}
```

Generic interface

```
public String first() {  
    this.iterator = 0;  
    return next();  
}  
  
public String next() {  
    if (this.iterator < this.size)  
        return this.items[this.iterator++];  
    return null;  
}  
  
public boolean hasNext() {  
    if (this.iterator < this.size)  
        return true;  
    return false;  
}  
}
```

Generic methods

- In Java genericity is not limited to classes and interfaces, you can define generic methods
- Static methods, non-static methods, and constructors can all be parameterized in almost the same way as for classes and interfaces
- Generic methods are also invoked in the same way as non-generic methods

Generic methods

- Consider the following segment of code that prints out all the elements in a container

```
public void printContainer(Container c) {
    for (Object o = c.first(); c.hasNext(); o = c.next()) {
        if (o instanceof Class1) {
            Class1 myItem = (Class1) o;
            // Print item of type Class1
        }
        else if (o instanceof Class2) {
            Class2 myItem = (Class2) o;
            // Print item of type Class2
        }
        else {
            // Not implemented
        }
    }
}
```

•Again, there is a strange construction with casting

•The problems can occur during run-time when casting is checked

Generic methods

- Using generics, this can be re-written as follows (note that the `Container<T>` is the container of any type)

```
public <T> void printContainer(Container<T> c) {
    for (T o = c.first(); c.hasNext(); o = c.next()) {
        // Print item of any type T
    }
}
```

•This indicates the method is polymorphic

•It denotes "for all types T"

Wildcards instead of parameters

- There are three types of wildcards
 - “**? extends T**” – Denotes a family of subtypes of type **T** i.e. $? \leq T$
 - “**? super T**” – Denotes a family of supertypes of type **T** i.e. $T \leq ?$
 - “**?**” – Denotes the set of all types or **any**
- Our polymorphic method with wildcard instead of parameter

```
public void printContainer(Container<?> c) {
    for (Object o = c.first(); c.hasNext(); o = c.next()) {
        // Now we again need casting!!!
    }
}
```

Examples with wildcards

- Consider a `draw()` method that should be capable of drawing any shape such as circle, rectangle, and triangle
 - Shape is an abstract class with three subclasses: Circle, Rectangle, and Triangle

```
public void drawShapes(Container<Shape> shapes) {
    for (Shape s = shapes.first(); shapes.hasNext(); s = shapes.next()) {
        s.draw();
    }
}
```

- It is worth noting that the `draw()` method can only be called on lists of Shape and cannot be called on a list of Circle, Rectangle, and Triangle for example

```
public void drawShapes(Container<? extends Shape> shapes) {
    for (Shape s = shapes.first(); shapes.hasNext(); s = shapes.next()) {
        s.shape();
    }
}
```

Subtyping generics

- Let's have types A, B where $B \leq A$, and generic type $C<T>$ where T is substituted either by A (we have $C<A>$) or by B (we have C)
- ~~$C \leq C<A>$~~
- $C \leq C<? \text{ extends } A>$

Animals in the cages

- A cage is a collection of things, with bars to keep them in

```
public interface Cage<T> extends List<T> { ... }
```
- A lion is a kind of animal, so `Lion` would be a subtype of `Animal`

```
public class Lion extends Animal { ... }
Lion lionKing = new Lion("Simba");
```
- Where we need some animal, we are free to provide a lion

```
Animal a = lionKing;
```
- A lion can of course be put into a lion cage

```
Cage<Lion> lionCage = new ArrayList<Lion>();
lionCage.add(lionKing);
```
- And a butterfly into a butterfly cage:

```
public class Butterfly extends Animal { ... }
Butterfly motylEmanuel = new Butterfly("Emanuel");
Cage<Butterfly> butterflyCage = new ArrayList<Butterfly>();
butterflyCage.add(motylEmanuel);
```

Is a lion cage a kind of all-animal cage?

- But what about an **all-animal cage**?

```
Cage<Animal> animalCage = new ArrayList<Animal>();
```

- This is a cage designed to hold all kinds of animals, mixed together. It must have bars strong enough to hold in the lions, and spaced closely enough to hold in the butterflies

```
animalCage.add(lionKing);
animalCage.add(motylEmanuel);
```

- Since a `Lion` is a subtype of `Animal`: **Is a lion cage a kind of all-animal cage? Is `Cage<Lion>` a subtype of `Cage<Animal>`? No!**

```
animalCage = lionCage;
animalCage = butterflyCage;
```

Compile-time error because `Cage<Lion>` is not a subtype of `Cage<Animal>`

- Without generics, the animals could be placed into the wrong kinds of cages, where it would be possible for them to escape

Is a lion cage a kind of all-animal cage?

- To specify a cage capable of holding **some kind of animal**

```
Cage<? extends Animal> someCage;
```

Only reference! We cannot instantiate a cage of generic type "`? extends Animal`"

- While `Cage<Lion>` and `Cage<Butterfly>` are not subtypes of `Cage<Animal>`, they are in fact subtypes of `Cage<? extends Animal>`

```
someCage = lionCage;
someCage = butterflyCage;
```

This is OK, `Cage<Lion>` and `Cage<Butterfly>` are subtypes of `Cage<? extends Animal>`

- Can you add butterflies and lions directly to `someCage`? No!**

```
someCage.add(lionKing);
someCage.add(motylEmanuel);
```

Compile-time error. Type of items someCage cannot be resolved (wildcard "`? extends Animal`" is used)

- If `someCage` is a butterfly cage, it would hold butterflies just fine, but the lions would be able to break free. If it is a lion cage, then all would be well with the lions, but the butterflies would fly away

Is a lion cage a kind of all-animal cage?

- So, if you cannot put anything at all into `someCage`, is it useless? No, because you can still read its contents

```
public void feedAnimals(Cage<? extends Animal> someCage) {
    for (Animal a : someCage)
        a.feedMe();
}
```

- You could house your animals in their individual cages, and invoke this method first for the lions and then for the butterflies

```
feedAnimals(lionCage);
feedAnimals(butterflyCage);
```

- Or, you could choose to combine your animals in the all-animal cage instead

```
feedAnimals(animalCage);
```

Examples with wildcards

- Another example of a generic method that uses wildcards to sort a list into ascending order

- Elements in the list must implement the `Comparable` interface
- Elements in the list must be comparable with all their supertypes (at least with their type)
- See Java API how `Comparable`, `List`, `ListIterator` and `Arrays` work

```
public static <T extends Comparable<? super T>> void sort(List<T> list) {
    Object[] array = list.toArray();
    Arrays.sort(array);
    ListIterator<T> iterator = list.listIterator();
    for (int i = 0; i < array.length; i++) {
        iterator.index();
        iterator.set((T) array[i]);
    }
}
```

Readings

- ECKEL, B.: *Thinking in Java*. 4th Edition, Prentice Hall, 2006
 - Generics (pp. 439 – 534)