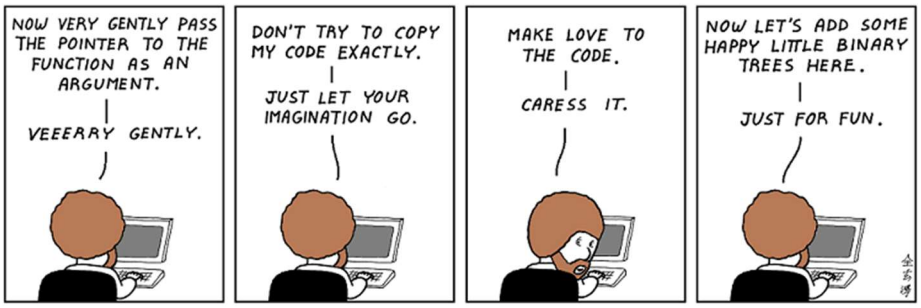


The Joy of Programming
with Bob Ross

OBJECT-ORIENTED PROGRAMMING

Generic Programming

Lecture #6

doc. Ing. Martin Tomášek, PhD.
Departments of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

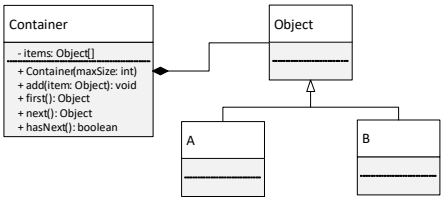
2023/2024

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 – usage of the container

- 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 A) {
      A a = (A) o;
      // Print item of type A
    }
    else if (o instanceof B) {
      B b = (B) o;
      // Print item of type B
    }
    else {
      // Not implemented
    }
  }
}
```

- There is a strange construction with casting
- The problems can occur during run-time when casting is checked

Example – need for parametrized types?

- The `Container` is 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 `A` and `B`, 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

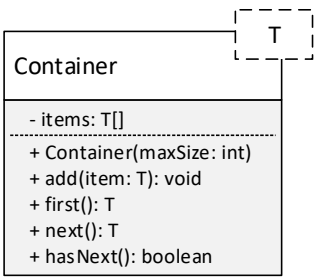
- Abstraction and encapsulation
- Subtyping and subclassing
- Subtype polymorphism
- **Templates / generics**
  - Classes, interfaces and functions with type parameters

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



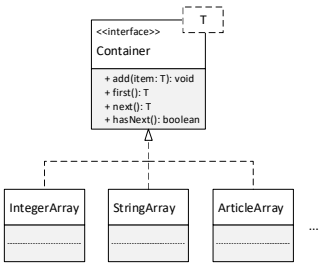
Example – conclusions

- Here we say that `Container` is a generic class that takes a type parameter `T`
  - `A` and `B` are substituted in the test case
- We no longer need to cast to e.g. an `A` since the `first()` and `next()` methods would return a reference to an object of a specific type
  - `A` 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> {
  void add(T item);
  T first();
  T next();
  boolean hasNext();
}
```



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

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

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

- This indicates the method is polymorphic
- It denotes "for all types T"

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

Wildcards instead of parameters

- There are three types of wildcards
  - `? extends T` – Denotes a family of subtypes of type `T` i.e. `? ≤ T`
  - `? super T` – Denotes a family of supertypes of type `T` i.e. `T ≤ ?`
  - `?` – 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.draw();
  }
}
```

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 $C<B>$)
- ~~$C<B> \leq C<A>$~~
- $C<B> \leq C<?\ extends\ A>$

Animals in the cages

- A cage is a collection of things, with bars to keep them in (let's use `List<T>`)
- A lion is a kind of animal, so `Lion` would be a subtype of `Animal`

```
public class Lion extends Animal { ... }
Lion simba = new Lion("Simba");
```

- Where we need some animal, we are free to provide a lion

```
Animal a = simba;
```

- A lion can of course be put into a lion cage

```
List<Lion> lionCage = new LinkedList<>();
lionCage.add(simba);
```

- And a butterfly into a butterfly cage:

```
public class Butterfly extends Animal { ... }
Butterfly emanuel = new Butterfly("Emanuel");
List<Butterfly> butterflyCage = new LinkedList<>();
butterflyCage.add(emanuel);
```

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

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

```
List<Animal> animalCage = new LinkList<>();
```

- 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(simba);
animalCage.add(emanuel);
```

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

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

Compile-time error because `List<Lion>` is not a subtype of `List<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**

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

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

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

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

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

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

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

Compile-time error. Type of items in `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(List<? extends Animal> someCage) {
  for (Animal animal : someCage)
    animal.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]);
  }
}
```