CS342: Software Design

Nov. 14, 2017

Today's topic

Java generics

Create a “trap”

```
public class Mouse {
   int hight;
   int weight;

   public int getHeight() {
      // ...
   }
   public int getWeight() {
      // ...
   }
}

public class MouseTrap {
   Mouse trapped;
   public void snare( Mouse trapped ) {
      this.trapped = trapped;
   }
   public Mouse release( ) {
      return trapped;
   }
}
```

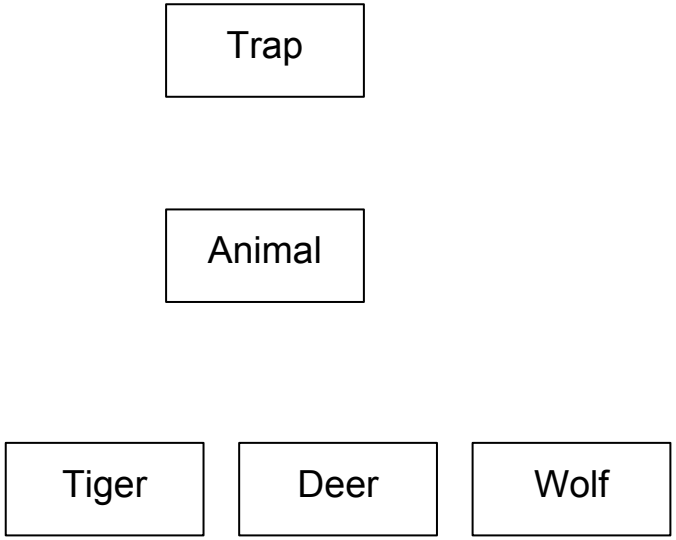
```
public class Bear {
   int hight;
   int weight;

   public int getHeight() {
      // ...
   }
   public int getWeight() {
      // ...
   }
}

public class BearTrap {
   Bear trapped;
   public void snare( Bear trapped ) {
      this.trapped = trapped;
   }
   public Bear release( ) {
      return trapped;
   }
}
```

What if there are more types of animals to trap

Tiger, deer, wolf, etc…

How can design patterns and principles help us?

- Program to an interface, not to an implementation
- Dependency inversion principle: depend on abstraction, not concrete classes
- Encapsulate what vary
- What if i want “Trap” to be used for non-animal objects?
- Is there a better way?



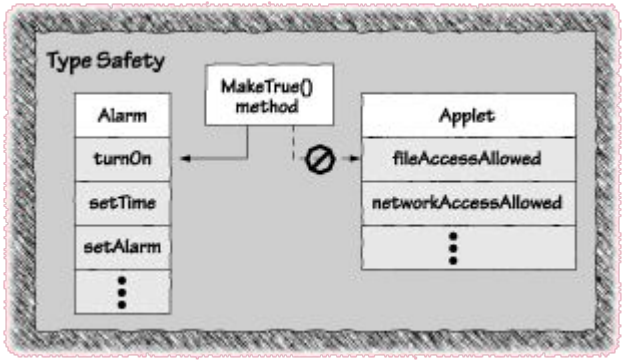
On the other hand, we have an issue with List interface

```
Date date = new Date( );
List list = new ArrayList( );
list.add( date );
String name = "CS342";
list.add(name);
//...
Date firstElement = (Date)list.get(0);
String secondElement = (String)list.get(1);

list.remove(0);
list.add(342);
```

In Java, a list holds an ordered collection of elements of type “Object”

- “Any” type == no type
- Consider each element as a “container”
- You can put any object in each container
- No compiler error… but what about runtime?
- Type safety

Can I code re-usability and type safety at the same time?



vs

Introducing java generics

An enhancement to the syntax of classes that allow us to specialize the class for a given type or set of types

- Requires type parameter(s) to customize the class
- Type variable: the identifier between angle brackets
- Indicate the class is generic and require a type as an argument to make it complete

```
public class List< E > {
   ...
   public void add( E element ) { ... }
   public E get( int i ) { ... }
}
```

```
List<String> listOfStrings;
```

```
List<Date> dates;
List<java.math.BigDecimal> decimals;
List<Foo> foos;
```

Instantiate (invoke) the type

Completing the type by providing a type parameter

- Now the generic class is “specialized”
- List< String > makes sure the elements are and can ONLY be of type String
- Can’t accept arbitrary “Object”

```
public class List< E > {
   ...
   public void add( E element ) { ... }
   public E get( int i ) { ... }
}
```

```
List<String> listOfStrings;

List<Date> dates;
List<java.math.BigDecimal> decimals;
List<Foo> foos;
```

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

Use specialized class

```
List<String> listOfStrings = new ArrayList<String>();
listOfStrings.add("eureka! ");
String s = listOfStrings.get(0); // "eureka! "

listOfStrings.add( new Date() ); // Compile-time Error!
```

```
public class Map< K, V > {
    ...
    public V put( K key, V value ) { ... } // returns any old value
    public V get( K key ) { ... }
}
```

```
Map< Integer, Employee > employees = new HashMap< Integer, Employee >();
Integer bobsId = ...;
Employee bob = ...;

employees.put( bobsId, bob );
Employee employee = employees.get( bobsId );
```

```
employees.put( 42, bob );
Employee bob = employees.get( 42 );
```

Erasure

Java compiler “erases” the generic nature of the class in the compiled form

- Maintain compatibility with non-generic code
- Java runtime doesn’t know anything about generics
- To runtime, there’s only one real type
- “instanceof” can’t be used for generic type
- Can’t implemented two different generic interfaces

```
List<Date> dateList  = new ArrayList<Date>();
System.out.println( dateList instanceof List );
```

```
dateList.add( new Object() ); // Compile-time Error!
```

```
System.out.println( dateList instanceof List<Date> );
// Illegal, generic type for instanceof
```

```
public abstract class DualList implements List<String>, List<Date> { }
// Error: java.util.List cannot be inherited with different arguments:
//    <java.lang.String> and <java.util.Date>
```

Raw types

Without providing parameterized types, a generic is degenerated to a raw type

- Old plain type
- Ensure compatibility with legacy code (pre-5.0)
- Java 5.0 and above gives “unchecked” warning when class is used in an “unsafe” way

```
// nongeneric Java code using the raw type, same as always
List list = new ArrayList( ); // assignment ok
list.add("foo"); // unchecked warning on usage of raw type
```

Parameterized type relationships

Parameterized types shared a common raw typ

- Therefore List<Date> is a List at run time
- Can assign any instantiation of List to raw type List
- Compiler is running the show here

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

List<Date> dates = new ArrayList(); // unchecked warning

List<Date> dates = new ArrayList<String>(); // Compile-time Error!
```

```
Collection<Date> cd;
List<Date> ld = new ArrayList<Date>();
cd = ld; // Ok!
```

List<Date> is NOT a List<Object>

```
List<Object> lo;
List<Date> ld = new ArrayList<Date>();
lo = ld; // Compile-time Error!  Incompatible types.
```

```
Collection<Number> cn;
List<Integer> li = new ArrayList<Integer>();
cn = li; // Compile-time Error!  Incompatible types.
```

Cast

```
Collection<Date> cd = new ArrayList<Date>();
List<Date> ld = (List<Date>)cd; // Ok!
```

```
Collection<Date> cd = new TreeSet<Date>();
List<Date> ld = (List<Date>)cd; // Runtime ClassCastException!
ld.add( new Date() );
```

```
Object o = new ArrayList<String>();
List<Date> ldfo = (List<Date>)o; // unchecked warning, ineffective
Date d = ldfo.get(0); // unsafe at runtime, implicit cast may fail
```

Use generic classes for the trap example

```
class Mouse { }
class Bear { }

class Trap< T >
{
    T trapped;

    public void snare( T trapped ) { this.trapped = trapped; }
    public T release( ) { return trapped; }
}
```

```
// usage
Trap<Mouse> mouseTrap = new Trap<Mouse>();
mouseTrap.snare( new Mouse( ) );
Mouse mouse = mouseTrap.release( );
```

```
List<T> trappedList = new ArrayList<T>();
```

```
public void trapAll( List<T> list ) { ... }
```

```
trapAll( List<Mouse> list ) { ... }
```

Note “Mouse” and “Bear” can be totally unrelated classes -- meaning no need for abstract class or interface.

Which OOP principles did generic class help us achieve?

Subclassing generics

```
class DateList extends ArrayList<Date> { }

DateList dateList = new DateList();
dateList.add( new Date() );
List<Date> ld = dateList;
```

Create a non-generic subclass

```
class AdjustableTrap< T > extends Trap< T > {
    public void setSize( int i ) { ... }
}
```

Create a generic subclass

Bounds

A constraint on the type of a type parameter

- Use “extends” keyword
- T needs to be “Employee” or its subclasses.
- “Employee” is the upper bound
- Can further require implement interfaces. Use “&” syntax

```
class EmployeeList< T extends Employee > { ... }

class EmployeeList< T extends Employee & Ranked & Printable > { ... }
```

```
class EmployeeList< T extends Employee & Ranked & Printable >
{
    Ranked ranking;
    List<Printable> printList = new ArrayList<Printable>();

    public void addEmployee( T employee ) {
        this.ranking = employee;  // T as Ranked
        printList.add( employee );  // T as Printable
    }
}
```

Wildcards

Implement polymorphism in generic class

- Use “?”: Unbounded wildcard
- Any instantiation is acceptable
- Bounded wildcards: limit range of assignable types, use extends

```
// A List<Object> is not a List<Date>!
List<Object> objectList = new ArrayList<Date>() // Error!

// A List<?> can be a List<Date>
List<?> anyList = new ArrayList<Date>(); // Yes!
```

```
List<?> anyInstantiationOfList = new ArrayList<Date>();
anyInstantiationOfList = new ArrayList<String>(); // another instantiation
```

```
List<? extends Date> dateInstantiations = new ArrayList<Date>();
dateInstantiations = new ArrayList<MyDate>(); // another instantiation
```

```
Trap< ? extends Catchable & Releaseable > trap;
```

Lower bounds

When i want the parameter to be a certain type of any of its supertypes (up to Object)

- Use “super”

```
List< ? super MyDate > listOfAssignableFromMyDate;
listOfAssignableFromMyDate = new ArrayList<MyDate>();
listOfAssignableFromMyDate = new ArrayList<Date>();
listOfAssignableFromMyDate = new ArrayList<Object>();
```

Generic methods

- Have a parameter type declaration using <> syntax
- Syntax appears before the return type of the method

```
class GenericClass< T > {
    // method using generic class parameter type
    public void T cache( T entry ) { ... }
}
```

```
// generic method
<T> T cache( T entry ) { ... }
```

```
BlogEntry newBlogEntry = ...;
NewspaperEntry newNewspaperEntry = ...;

BlogEntry oldEntry = cache( newBlogEntry );
NewspaperEntry old = cache( newNewspaperEntry );
```

```
class MathUtils {
    public static <T extends Number> T max( T x, T y ) { ... }
}
```