Object Oriented Software Engineering (COMP2003)

Lecture 8: Generics

Updated: 14^{th} May, 2019

Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

Outline

Reuse

- Reuse is good. (You knew that, hopefully!)
  - Avoid multiple classes or methods that do basically the same thing.
  - Less code = less maintenance effort.
- But some re-use is harder.
  - e.g. Two methods that are *similar* but not identical.
  - Write a single replacement method that "generalises" them.
  - Replace hard-coded values with parameters.
  - Use the Template Method pattern to handle algorithmic differences.
- But sometimes you need an extra trick up your sleeve.

Generic Containers

- You know about generics already:

```
List<String> theList = new ArrayList<>();
...
String str = theList.get(i);
```

- Containers (lists, sets and maps) are the most common use.
- Generics lets us specify the list element type (e.g. `String`).
- Otherwise we have to *assume* what's in the list:

```
List theList = new ArrayList();
...                                       // Needs unsafe
String str = (String)theList.get(i); // downcasting
```

  - If we assume wrong, we'll get logic errors.
- Or, we must have different lists for each type:

```
StringList theList = new StringArrayList(); // Ugh!
```

Type Parameters

- But lists, sets and maps are only *one use* of generics.
- We can attach "type parameters" to any class or method:

```
public class MyClass<T> // Declare type param 'T'
{
    private T value;    // Use T as a data type
    public MyClass(T newVal)
    {
        this.value = newVal;
    }

    public T getValue()
    {
        return value;
    }
}
```

Type Parameters

- So if we write this:

```
public class MyClass<T> { ... }
```

- Then we can later write this:

```
MyClass<String> obj1 = new MyClass<>("xyz");
MyClass<Integer> obj2 = new MyClass<>(42);
```

- We're effectively saying that `T = String`, or `T = Integer`.
- It's (sort of) like we've made two versions of `MyClass`.

Note

"`<>`" (the diamond operator) is a short-cut. We could write it out in full: `MyClass<String> obj1 = new MyClass<String>("xyz");`

Bounded Type Parameters

We can specify that T must extend a given class/interface:

```
public class MyClass<T extends JComponent>
{                              // We don't know what T is,
    private T component;       // but we do know it extends
                               // (or is) JComponent, so we
    public void show()         // know some of its methods.
    {
        component.setVisible(true);
    }

    public T getComponent() {...}
    public void setComponent(T component) {...}
}
```

- `MyClass<JButton>` works, but `MyClass<String>` is an error.
- Note: use "`extends`" for *interfaces* too, not "`implements`".

Multiple Bounds (Intersection Types)

- You can require T to extend *multiple* things.
- This is an "intersection type" (think of intersecting sets).

```
public interface Encodable { String encode(); }
public interface Viewable { UIElement view(); }

public class EmailAttachment<T extends Encodable &
                                       Viewable>
{
    private String filename;
    private T attachedObject;
    ...
} // T is anything that implements both interfaces,
  // and so it has the methods from both.
```

- Generics is sometimes used *just* to get intersection types.

Generic Methods

- Individual methods can have their own type parameters.
- Say we want a method that finds and returns the middle element of a list:

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

- The first <T> is the type parameter declaration.
- This method takes in a list of *something*, and returns *something* of the same type.
- The compiler can usually figure out what T is based on the parameters:

```
List<String> list = ...;
String s = obj.getMiddle(list); // T = String
```

Generic Methods (2)

- Sometimes the compiler can't figure out what the type parameters actually are.
  - If it can't, it will tell you.
- You can explicitly pass type parameters to a method like this:

```
myObj.<String,Integer>method();
MyClass.<LocalTime>staticMethod();
```

- The syntax *requires* a class name or object reference beforehand.
  - Use "this" if necessary.
- e.g.

```
method2(this.<String>method1());
```

- The compiler probably needs your help when:
  - The type parameter occurs only in the return type; and
  - You're passing the return value onto another method, itself accepting a generic type.

Where and Why?

- Generics are used when the alternatives are:
  - Multiple versions of the same code with different types; OR
  - Using a general type (like `Object`, `JComponent`, etc.), but assuming you actually have a more specific type.
    - And downcasting one to the other, which is usually undesirable.
- Put another way, generics lets you:
  - Reuse the same code with different datatypes; AND
  - Keep "type safety" (the compiler verifies you're using the correct datatypes).
- You don't create generic classes every day.
  - The above situations don't happen that often.
  - They're more likely to come up when you create a library, to be used in various ways across different applications.

Some Examples from the Java API

- Containers (lists, sets and maps), as you know.
- `JComboBox<E>` – a Swing GUI widget that shows a drop-down list, where that list contains any specific datatype.
- `Comparator<T>` – for performing a comparison (<, = or >) between two objects of the same specific type.
- `Iterator<E>` – for retrieving objects from a container.
- `Stream<T>` – for applying functional operations to a sequence of objects of any specific type.
- `Callable<V>` and `Future<V>` – for representing a parallelisable task that, when finished, will produce a result of some specific type.
- `WeakReference<T>` and `SoftReference<T>` – for referring to an object of a specific type without preventing it from being garbage collected.

Limitations of Generics (in Java)

- No primitives.
  - ✗ ~~`MyClass<int>`~~
  - ✓ `MyClass<int[]>`, `MyClass<Integer>`
- Erasure. Type parameters *don't exist* at runtime.
  - The compiler validates type parameters, but then erases and replaces them with (generally) `Object`.
  - ✗ ~~`if(T == String) {...}`~~
  - ✗ ~~`if(list instanceof List<String>) {...}`~~
- No generic array creation.
  - To do with erasure and backwards compatibility. Arrays must know their element type at runtime.
  - ✗ ~~`T[] array = new T[10];`~~ (array of T)
  - ✗ ~~`List<T>[] array = new List<T>[10];`~~ (array of lists of T)
  - ✓ `void method(T[] array) {...}`
  - ✓ `List<T> list = new ArrayList<T>();`

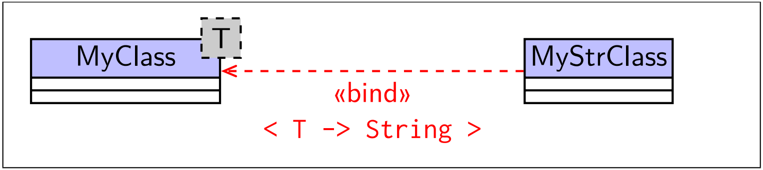
Naming Conventions

- Type parameter names are usually a single uppercase letter.
- Java's API uses the following names[1]:
  - `E` – Element (e.g. in lists and sets).
  - `K` and `V` – Key and Value (e.g. in maps).
  - `N` – Number.
  - `T` – Type (used when nothing else is obviously applicable).
  - `S`, `U` and `V` – Additional miscellaneous types following `T`.
- Are these one-letter names bad?
  - Technically, you can use *any* name (following the same rules as for variables, methods and classes).
- Actually, they are good for type parameters.
  - Clearly identifies them.
  - You won't need many (more than two is extremely rare).
  - Any names you pick will be extremely *generic* anyway.

[1] https://docs.oracle.com/javase/tutorial/java/generics/types.html

Generics (“Templates”) in UML

- UML does a terrible job (probably due to C++’s influence):



  - This is vaguely like an inheritance relationship.
  - MyClass on the left is a *template*, with type parameter T.
  - MyStrClass is a class, equivalent to `MyClass<String>`.
  - The line is a *template binding*, specifying what T is.
- But. . . the whole point was to *avoid* things like `MyStrClass`!
  - Sadly, UML requires a whole separate class for each value of T.
  - It doesn’t really have “true” generics.
- So, clumsy, highly redundant, and hard to draw. Yay!
- Maybe best to avoid this notation. Invent your own if needed!

Simple Example

Let's create our own (possibly useful) generic class:

```
public class Finder<T>
{
    private T toFind;
    public Finder(T toFind) { this.toFind = toFind; }

    public T getBefore(Collection<T> collection) { ... }
    public T getAfter(Collection<T> collection) { ... }
}
```

- This class will:
  - Search any collection for `toFind`;
  - Return the *previous* or *next* element (or `null`).
- By making this a generic class, we ensure that:
  - `toFind` is always the right type for the collection.
  - No typecasting is needed for the return value.

Using our Example Generic Class

```
List<String> theList = new ArrayList<>();
theList.add("abc");
theList.add("def");
theList.add("ghi");

Finder<String> finder = new Finder<>("def");
String before = finder.getBefore(theList);
String after = finder.getAfter(theList);
```

Bounded Example

Let's consider a variation:

- Rather than an *exact match*, let's find the 1st *smaller* object.
- For this, our objects must be Comparable:

```
public class SmallFinder<T extends Comparable<T>> { ... }
```

- The declaration may look recursive, but it isn't really.
- A class either implements Comparable<*itself*>, or it doesn't.
  - The Object class does not, so T cannot be Object.
  - But the String class *does* implement Comparable<String>.
  - You can do likewise:

```
public class Xyz implements Comparable<Xyz> {...}
```

```
SmallFinder<Xyz> obj = new SmallFinder<>(new Xyz());
```

- As mentioned, inside <...> we write "extends" for both classes and interfaces.

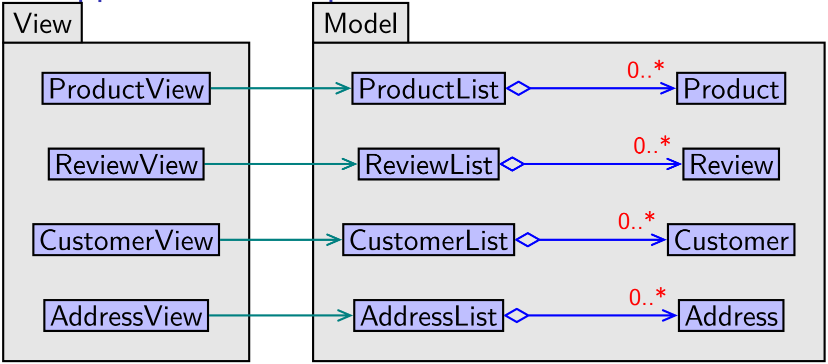
Bounded Example – Inner Workings

- In case you need convincing about how this works:

```
public class SmallFinder<T extends Comparable<T>>
{
    private T toFind; // What we're looking for
    public SmallFinder(T toFind) { this.toFind = toFind; }

    public T getBefore(Collection<T> collection)
    {
        T prev = null;
        for(T curr : collection) // Iterate through container
        {
            if(toFind.compareTo(curr) < 0) { return prev; }
            prev = curr;
        }
        return null;
    }   // The compareTo(T) method exists in Comparable<T>.
}       // It returns a negative value if toFind < curr.
```

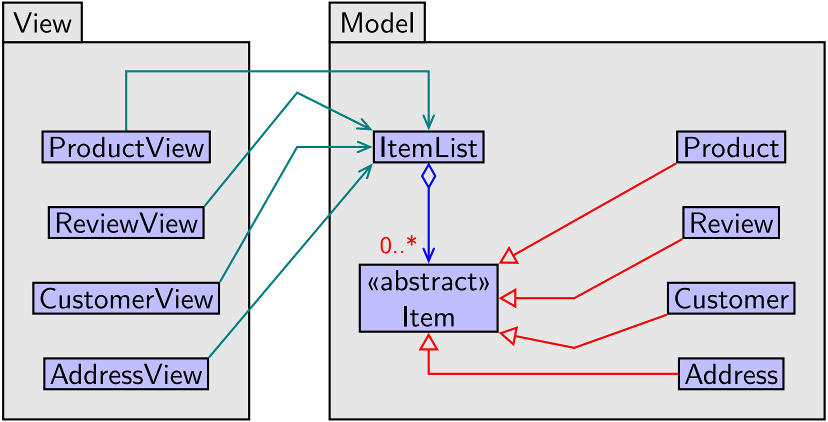
Retail Application Example



- Consider this system – just *this* simplified aspect of it.
    - Obviously lots of detail omitted.
- Say the `List` classes are all essentially the same.
    - Storing items in memory.
    - Saving/loading items to/from a database table.
    - Registering/notifying observers.
    - (Not *just* `java.util.List` stuff.)

Retail Application Example (2)

- To reduce redundancy, we merge all the `List` classes into one.
- ... And create a common superclass for `Product`, `Review`, `Customer` and `Address`.



Retail Application Example (3)

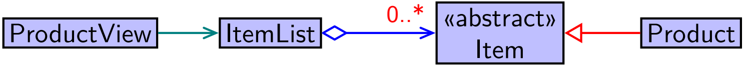
```
public class ItemList // New all-purpose model class.
{
    private List<Item> items = new ArrayList<>();
    private Set<Observer> observers = new ArrayList<>();

    // Used by the controller:
    public void addItem(Item item) {...}
    public void removeItem(Item item) {...}
    public void saveToDatabase() {...}

    // Used by the view:
    public List<Items> getItems() {...}
    public void addObserver(Observer ob) {...}
    ...
}
```

Retail Application Example (4)

- PROBLEM: we just destroyed type safety in the `View`!



- `ProductView` needs to access `Product` *specifically*, not `Item`.

```
public void displayProducts()   // ProductView.java
{
    for(Item item : itemList.getItems())
    {
        Product p = (Product)item;
        ... // To display the Products, we must
    }       // downcast Item, since Item won't
}           // tell us what we need.
```

- Downcasting can't be sanity-checked by the compiler.
  - Makes future defects more likely!
  - Possible future bug: `ProductView` gets the wrong type of `Item`.

Retail Application Example (5): Use Generics!

```
public class ItemList<I extends Item>
{
    private List<I> items = new ArrayList<>();
    private Set<Observer> observers = new ArrayList<>();

    // Used by the controller:
    public void addItem(I item) {...}
    public void removeItem(I item) {...}
    public void saveToDatabase() {...}

    // Used by the view:
    public List<I> getItems() {...}
    public void addObserver(Observer ob) {...}
    ...
}
```

Retail Application Example (6): Use Generics!

- Now each `View` class can have an `ItemList` with a different type parameter; e.g.:

```
public class ProductView
{
    private ProductList productList;  // Original field
    private ItemList<Product> productList; // New field
    ...
    public void displayProducts()
    {
        for(Product p : productList.getItems())
        {
            ... // Display product. No downcasting!
        }
    }
}
```

Retail Application Example (7): Summary

- We achieved both reuse and type-safety.
- We combined four classes into one (because they were essentially the same).
- *But* they each dealt with a different datatype.
- *So*, we gave the new class a type parameter:

```
public class ItemList<I extends Item>
```

  - The type parameter `I` can be any class that extends `Item`.

- When we need a specific kind of `ItemList`, we can have it:

```
private ItemList<Product> productList;
```

Generics and Inheritance (1)

- Generics and inheritance interact in various ways.
- You can pass concrete types to a generic superclass.
- For instance:

```
public class MyList implements List<String> {...}

public class MySet extends HashSet<LocalTime> {...}
```

1. Implements the `List` interface, but only for lists of strings.
2. Creates a `HashSet` subclass, specifically for `LocalTime`s.

- On one hand, you're inheriting from a *specific version* of a class/interface.
- On the other hand, technically there is only one `List` interface, and only one `HashSet` class, anyway.
  - At runtime, `List<String>` simply becomes `List`.

Generics and Inheritance (2)

- You can pass a type parameter to the superclass:

```
public class MyList<E> implements List<E> {...}

public class MyMap<K,V> extends TreeMap<K,V> {...}
```

  - New `List` and `Map` classes that accept any element type.

- But be careful about what <E> actually means!

Declares the type parameter E

```
public class MyList<E> implements List<E> {...}
```

Passes E on to List

Generics and Inheritance (3): Combining Concepts

- Combining the two previous cases:

```
public class MyMap<V> implements Map<String,V> {...}
```

  - A Map implementation, where the keys must be strings, but the values can be anything.

- Bounded type parameter *and* inheritance:

```
public class OrderedSet<T extends Comparable<T>>
    extends HashSet<T> {...}
```

  - Don't mix up the two "`extends`".
  - `T` *must* extend `Comparable<T>`.
    - We're just setting the rules for how `OrderedSet` can be used.
  - `OrderedSet` *does* extend `HashSet`.
    - We're actively making that happen.

Wildcards

- Wildcards are a way to *not care* about a type parameter.
  - Or at least not care *exactly* what it is.
  - This syntax is very Java-specific.
- There are three forms:

`List<?> var1;`	*Unbounded:* any list.
`List<? extends MyClass> var2;`	*Upper-bounded:* a list of `MyClass` or any subclass.
`List<? super MyClass> var3;`	*Lower-bounded:* a list of `MyClass` or any superclass.

- These are special reference types, all more general than `List<MyClass>`.
  - `List<MyClass>` can be upcasted to any of them.

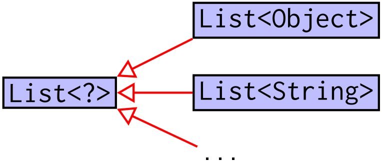
Wildcards: Unbounded Example

- This method can work with any kind of list:

```
public int countRepeats(List<?> theList)
{
    int repeats = 0;
    Object prev = null;
    for(Object current : theList) // We don't need to
    {                             // know the element
        if(current.equals(prev))  // type. All objects
        {                         // have .equals().
            repeats++;
        }
        prev = current;
    }
    return repeats;
}
```

List<?> ≠ List<Object>

- Why doesn't countRepeats() simply take in List<Object>?
  - Remember the discussion from Lecture 3 about upcasting containers.
- List<Object> is *not general enough*.
  - Part of the List contract is *adding* elements.
  - List<Object> is contractually-required to *accept any* Object.
  - But other list types can't meet this requirement, and countRepeats() doesn't need to impose it.
- For upcasting purposes, you can think of List<?> like this:



Disclaimer: this bends the rules of UML a lot, since these are only *reference types*. There's only one actual class involved.

`List<?>`: The Details

- "?" represents an unknown type.
- Everything (even unknown types) can be upcast to `Object`, so this works:

```
List<?> list = ...;
Object obj1 = list.get(0);   // Works
for(Object obj : list) {...} // Works
```

  - In this respect, `List<?>` *does* work like `List<Object>`.
- But *nothing* can be upcast to an unknown type, so we can't do this:

```
List<?> list = ...;
list.add(...);       // Can't add *any* object here.
```

  - In fact we *can* add `null`, because `null` is a valid value for every Java reference type.

Bounded Wildcards and PECS

- We use bounded wildcards for "producers" and "consumers".
- Remember: **Producer Extends; Consumer Super** (PECS).

Producers

- If you're only *getting* items from a list, it is a "producer".
  - A producer is any object that gives you other objects.
- You can accept producers of more-specific types.
- If you need `Animal`s, you can also accept a producer of `Bird`s.

```java
public Animal getFrom(List<? extends Animal> producer)
{
    return producer.get(0); // Producer: List<Animal>,
}                           // List<Bird>, List<Mammal>, etc.
```

- "`X<? extends Y>`" is an *upper-bounded* wildcard type.
  - Refers to an `X` object whose type parameter extends, *or is*, `Y`.

Bounded Wildcards and PECS

Consumers

- If you're only *putting* items into a list, it is a "consumer".
  - A consumer is any object *to which you give* other objects.
- If your code supplies `Plants` to a consumer, it's okay if the consumer accepts other things too.
  - You can accept a consumer (e.g. a list) of something *more general* than you have to give it.

```java
public void addTo(List<? super Plant> consumer)
{                                    // Consumer: List<Plant>
    consumer.add(new Plant());       // List<LifeForm> or
}                                    // List<Object>.
```

- "`X<? super Y>`" is a *lower-bounded* wildcard type.
  - Refers to an `X` object whose type parameter is a supertype of (*or is*) `Y`.

Variance

- OO theory has proper terms for what we just did.
- These describe how upcasting works with generic references.

Invariance – `List<MyClass>`

- Upcasting *cannot* change the generic type `MyClass`.

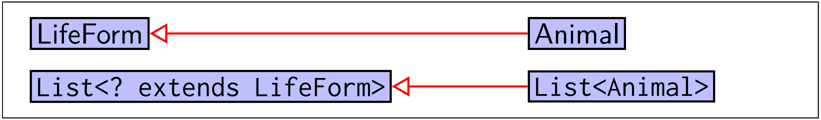
Covariance – `List<? extends MyClass>`

- Upcasting can move the type parameter “higher up”.
- Note: `List<?>` = `List<? extends Object>`.

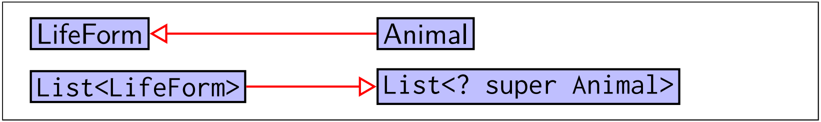
Contravariance – `List<? super MyClass>`

- Upcasting can move the type parameter “lower down”.

Covariance and Contravariance



- In **covariance** (“producer extends”), upcasting works in the same direction for both the following:
  - The producer reference type (`List<...>` in this case); and
  - The element type (`LifeForm`, etc.).



- In **contravariance** (“consumer super”), they work in opposite directions.
  - When upcasting the consumer reference type...
  - ...we may choose a *more specific* element type.

Other Languages

Python – No generics.

- Python is dynamically typed. Generics are fundamentally about static typing.

C++ – Templates.

- Looks a bit like generics, and solves a similar problem.
- Very different implementation!

C# – Reified generics.

- Quite similar to Java.
- But no *erasure* – type information is preserved at runtime.

Scala – Higher-kinded generic types.

- Similar to Java.
- But *even more generic!*

Scala's Higher Kinded Types

- …For when your generics just aren't *generic* enough.
- I won't show you the actual Scala *syntax*.
- If Java had "Higher Kinded Types", they might look like this:

```
public class Converter<T<?>>              // Pseudo-Java
{
    public T<Integer> parseAll(T<String> strings)
    {
        ... // Convert all strings to integers.
    }
}   // T could be LinkedList, HashSet, Comparable, etc.
```

  - Takes in any type that has been parameterised with `String`.
  - Returns the same type, but parameterised with `Integer`.
- The theory is that:
  - `T` doesn't need to be a complete type;
  - Rather, `T` can be a *sort-of-function* that takes a type (or multiple types) and returns a type.

C++ Templates

```
template <class T>
class MyTemplateClass {
    ...
    T getData() const;
    void setData(T data);
};
```

```
MyTemplateClass<std::string> obj1;
MyTemplateClass<int> obj2;
```

- Looks like generics, but there are *two* classes here, not one.
- C++ takes the definition of `MyTemplateClass`, replaces `T` with `std::string` (or `int`), and compiles it.
- "`MyTemplateClass<int>`" is a class, on its own, that deals with `int`s.

C++ Template Functions

- Templates can be functions too:

```
template <class T>
T add(T first, T second)
{
    return first + second;
}
```

```
int x = add<int>(5, 10);

std::string s1 = "Hello";
std::string s2 = "World";
std::string s3 = add<std::string>(s1, s2);
```

- This creates two entirely separate functions.
- The compiled code for `add<int>()` and `add<std::string>()` is different.

C++ Templates: Duck Typing

```
template <class T>
void display(T obj)
{
    std::cout << obj.getName() << obj.getAge();
}
```

- How do we know that T has `getName()` and `getAge()`?
- We don't!
- C++ uses compile-time duck typing.

```
Person p("Sam", 30);
display<Person>(p); // Works

display<int>(10); // Error: 'int' has no 'getName()'
```

- The compiler makes the substitution, and *then* checks if the resulting code makes sense.

C++ Templates: Other Things

- You can *specialise* a template.
  - Explicitly write out a particular version of it.
  - You can use this to do specific things in specific cases.

```
// Template
template<class T> class MyTemplateClass {...};

// Specialisation for 'int'
template<> class MyTemplateClass<int> {...};
```

- Template parameters can also have ordinary datatypes:

```
template<class T, int i, char c>
class MyTemplateClass {...};
```

```
MyTemplateClass<float, 5, 'X'> obj;
```

  - These values need to be known at compile-time.
  - "`class`" or "`typename`" (equiv.) indicate a type parameter.

C# Reified Generics

- Pronounced “ray-i-fied” or “ree-i-fied”.
- We mentioned that Java performs *erasure*.
  - Generic type parameters are removed *at compile time*.
  - They get replaced by Object (or the upper bound, if present).
- C# *keeps* generic type information at runtime.
  - You can check whether an object is an instance of T or List<String> at runtime.
  - You can instantiate type T:

```
// C#
                Requires T to have a no-argument constructor

public class MyFactory<T> where T : new()
{
    public T Make()
    {
        return new T(); // We don't know what T is,
    }                   // but we can still make one!
}
```

Where's My Reification, Java?!

- "new T()" is extremely illegal in Java, due to erasure.
- But sometimes it would be nice to do.
  - You can call *methods* without knowing the exact class...
  - So why not *constructors* too? Polymorphic constructors!
- And...oh you can. Just not (entirely) using generics:

```
public class MyFactory<T>                                   // Java
{
    private Class<T> type;
    public MyFactory(Class<T> type) { this.type = type; }

    public T make() throws ReflectiveOperationException
    {
        return type.newInstance();
    }   // 'type' is an object that represents T at runtime.
}       // (Fails at runtime if T has no no-arg constructor.)
```

Reflection

- If/when the type system fails you, there is one place you can turn.
- `Java's` "`Class`" class is part of the "reflection" API.
- *Reflection* is the ability to examine a program *programmatically*.
  - It lets your code see how *other* code is structured.
  - You can get a list of the methods in a class.
  - You can get a list of a method's parameter types.
  - You can call a method without knowing what it is!
  - You can work with *annotations*.
- But that's for another day!