

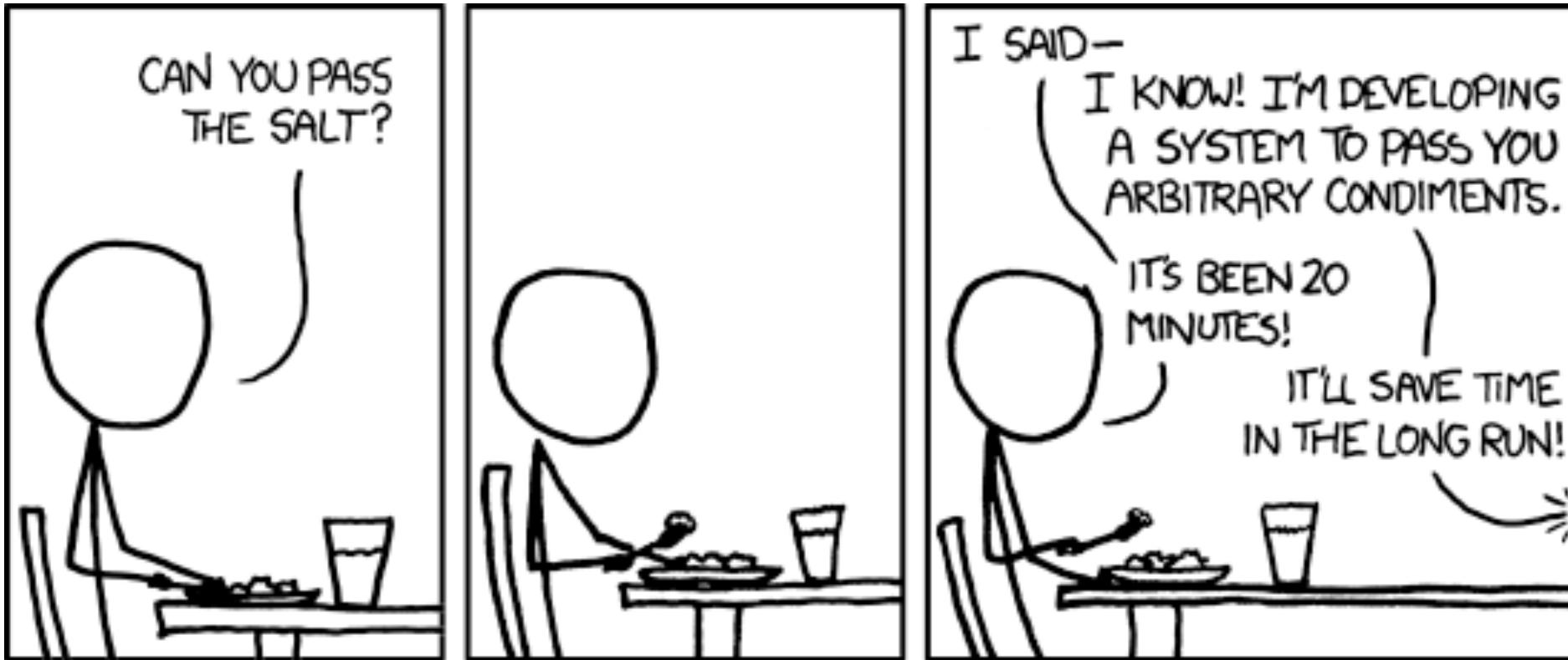


Courant Institute of Mathematical Sciences
Department of Computer Science
CS102 Data Structures
Prepared by Prof. Anasse Bari

Recitation Four: Introduction to Generics



Objectives



Why Types?

- Reason about a program
- Debugging/preventing bugs
- Compile time vs. run time bugs
- Type theory and the study of programming languages is an important field in computer science!

What are Generics?

- “Parametric polymorphism”: increase the expressiveness of a language/program while also introducing stronger static type checking
- Write functions/classes that handle their inputs identically without depending on their types
- To remember: write program with types-to-be-specified-later
- We use type parameters such as $\langle T \rangle$ and specify a type later
- “We abstract over types”
- This paradigm is called **GENERIC PROGRAMMING**

Why Generics?

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

```
1 List list = new ArrayList(); // no type parameter is specified
2 list.add("hello");
3 String s = (String) list.get(0); // this cast is necessary
4
```

```
1 List<String> list = new ArrayList<String>(); // parametrize
2 list.add("hello");
3 String s = list.get(0); // no cast
4
```

A Quick Note on Java Types

- Technically, every non-primitive in Java has a “super-parent” of type Object
- Let A, B be two types such that B extends A
 - We say that B is a subtype of A (i.e. B is child of A)
 - We say that A is a supertype of B (i.e. A is parent of B)
- Both A and B have supertype Object
- Take away: everything is (implicitly) inherited from object “Object”
- Extra credit:
 - <https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html>

Java Generics

- Introduced after Java 5.0 (you can see the casting example in slide 6 [top] for legacy Java code)
- A widely known example is the Java Collections Library (again refer to the casting example in slide 6 [bottom] for collections code)
- Motivating example:

```
1  \\\ holds some object
2  public class Box {
3      private Object obj;
4
5      public void set(Object o) { obj = o; }
6      public Object get() { return obj; }
7
8  }
```

Java Generics (Continued)

- Free to pass in whatever we want, provided it is not a primitive type
 - Why? (We will get to this later)
- However, there is no way to verify at compile time how this class is being used
 - Why?
- So we create generic Box

```
1  \ holds some type T
2  ┌─ public class Box<T> {
3  |   private T obj;
4  |
5  |   public void set(T o) { obj = o; }
6  |   public Object get() { return obj; }
7  }
8 }
```

Java Generics (Continued)

- We introduce the type variable T with the class declaration, then we can use this type variable throughout our code
- Now we can do:

```
1 Box<Integer> integerBox = new Box<Integer>();  
2
```

- This way the compiler can reason about the code in a static context and prevent issues like the following:

```
1 integerBox.set(5); // this is fine  
2 String s = integerBox.get(); // uh oh  
3
```

Java Generics (Continued)

- This is also valid:

```
1 Box<Integer> integerBox = new Box<>();  
2
```

- However, this is not:

```
1 Box<Integer> integerBox = new Box();  
2
```

- Why? Object is used for all generic type parameters which will result in the compiler giving a warning.

How are generics implemented?

- Type erasure:
 - Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded
 - Insert type casts if necessary to preserve type safety
 - Generate bridge methods to preserve polymorphism in extended generic types
 - The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods
- Why?
 - Because it did not require the JVM to be changed (Java 5+)
 - Type erasure still allows the compiler to catch static type safety errors

Limitations

- No generic arrays
 - Why? See the Bonus Topic slide
 - This is technically allowed but is not type safe therefore discouraged (see page 94 of textbook of a BAD workaround)
- No instantiations of generic types
 - Why? (consider what it means for the compiler when we try to write `T t = new T();`)
- No instantiations with primitive types
 - See next slide
- Type information is lost at run-time
 - Can be annoying, especially if using reflection (advanced Java topic)

Primitives

- Primitives types are stored on the stack
 - We call these values unboxed
- Non-primitive types are stored on the heap
 - We call these values boxed
- Thus, we need to introduce “big” types (int -> Integer) to have generic primitives
 - Why? Primitives do not inherit from Object
 - Integer holds a field, say value, that has some int value
 - Generally, the compiler will automatically do these conversions although there is a run-time cost (autoboxing)

Bounded Types (Bonus Topic)

- When Java was first created there were no generics and arrays were allowed to be covariantly typed
 - This means that if B extends A (i.e. B is a subtype of A) then B[] is also a child of A[] (i.e. B[] is also a subtype of A[])
 - This is considered as a flaw in the Java language (the Cats-Dogs problem)
- When generics were introduced Java creators had a chance to fix this issue
- Thus, when generics are used they are typed invariantly
 - This means that if B is a subtype of A then List is not a subtype of List<A>

Cats-Dogs Problem

- Assuming a class Animal, assume also classes Cat and Dog where both Cat and Dog inherit from Animal, consider the following example:

```
1 List<Dog> dogs = new ArrayList<Dog>(); // ArrayList of Dogs (i.e. a litter of puppies)
2 List<Animal> animals = dogs; // If co-variance was allowed
3 animals.add(new Cat()); // by definition a Cat can now be added in
4 Dog dog = dogs.get(0); // well, what now? We have a very confused cat
5
```

Bounded Types (Continued)

- The invariant property of generic types introduces some restrictions, thus the creators of Java also introduced bounded types
- Upper bound: `<B extends A>`, B can be at most of type A or any other child of A
 - Ex: `public <B extends Number> add(B num1, B num2);` (i.e. we can add any two numbers such as Float, Integer and so on)
 - See more here: <https://docs.oracle.com/javase/tutorial/java/generics/bounded.html>
- Other bounds: In this case we use the wildcard keyword “?” along with a combination of “extends”, “super” or nothing, for even more Java mastery I suggest you look at here:
<https://docs.oracle.com/javase/tutorial/java/generics/wildcards.html>

Wildcards (Even more extra)

- This is an useful topic for Java interviews
- The wildcard character (?) is used to represent an unknown type and is never used as a type argument for a generic method invocation, a generic class instance creation, or a supertype
 - Upper bound: `public static void process(List<? extends Foo> list)`
 - This input list for process is parameterized over at most type Foo (and can be any subtype of Foo)
 - Lower bound: `public static void process(List<? super Foo> list)`
 - This input list for process is parameterized over at least type Foo (and can be any supertype of Foo)
 - Unbounded: `public static void process(List<?> list) { list.clear(); }`
 - Here we do not care what the type of the list is since clear is not type dependent
 - If we were to use only methods defined in Object then unbounded wildcard works too

Cats-Dogs Problem (Advanced)

- Consider: `List<? extends Animal>`
 - You can't add a Cat (or Animal) in, but you are guaranteed to retrieve an Animal, why?
 - In general: You cannot put anything into a type declared with an extends wildcard except for the value null
- Consider: `List<? super Animal>`
 - You can add an Animal in, but you cannot retrieve any Animal from it (you can retrieve an Object though), why?
 - In general: You cannot get anything out from a type declared with an super wildcard except for a value of type Object, which is a super type of every reference type

The Get-Put Principle (Advanced)

- Use an **extends** wildcard when you only get values out of a structure.
- Use a **super** wildcard when you only put values into a structure.
- And don't use a wildcard when you both want to get and put from/to a structure.