JSR 14 - Adding Generics to Java presented by Dylan White, Sean Meier, and Ryan Roden

Introduction

- Java was introduced in 1995 with the intent of being able to write code once and run it anywhere
- It derived much of its original syntax from the C++ programming language
- Templates were left out
- Generics were proposed in JSR 14 to cover much of what was left out with templates

Why Not Use Templates?

Why didn't Java implement templates while implementing much of the rest of C++?

Templates Under the Hood

- When you call a template function in C++ the compiler translates it into a function for that specific usage
- Can create massive numbers of functions for a relatively small amount of code
- Can cause hard to read errors

Template Example

```
#include <stdio.h>
using namespace std;

template <class T>
T getMax(T a, T b) {
  return (a>b?a:b);
}
int main(void) {
  printf("max of 23,47 is %i\n", getMax(23,47));
  printf("max of 2.3,7.8 is %e\n", getMax(2.3,7.8));
}
```

Template Example (Cont.)

getMax(int a, int b)

```
0000000000400597 <_Z6getMaxIiET_S0_S0_>:
400597:       55                      push   %rbp
400598:       48 89 e5                mov    %rsp,%rbp
40059b:       89 7d fc                mov    %edi,-0x4(%rbp)
40059e:       89 75 f8                mov    %esi,-0x8(%rbp)
...
```

getMax(float a, float b)

```
00000000004005b3 <_Z6getMaxIdET_S0_S0_>:
4005b3:       55                      push   %rbp
4005b4:       48 89 e5                mov    %rsp,%rbp
4005b7:       f2 0f 11 45 f8          movsd  %xmm0,-0x8(%rbp)
4005bc:       f2 0f 11 4d f0          movsd  %xmm1,-0x10(%rbp)
```

Moving to Generics

How do generics handle this?

- Error checking - Static analysis is used to determine if the code should throw an error
- Type erasure - Java throws out the types of the objects at compile time and casts them to the appropriate type

Why is this important?

- Speeds up compile time
- Reduces binary size
- Errors can be understood by mere mortals

Errors at the Wrong Time

```
struct Foo{
  private:
    int foo(){return 7;}
};
template <class T>
int callFoo(T a){
  return a.foo
}
```

Errors at the Correct Time

```
public class Bar {
  public Integer foo(){
    return 10;
  }
}
public class BarPriv extends Bar{
  private Integer foo() {  //Error: "Cannot reduce the visi
    return 20;
  }
}
```

Fruit Class Hierarchy

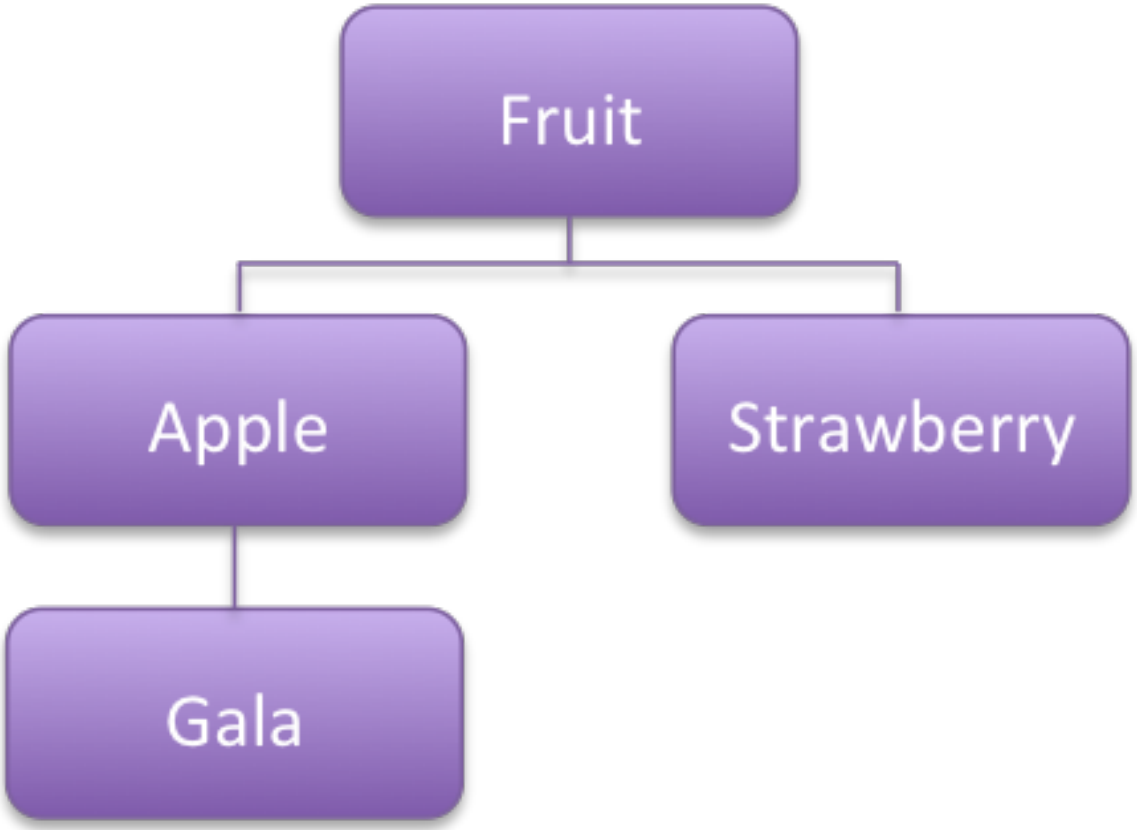


Figure : Type Hierarchy

Covariance and Contravariance

Covariance - Specialized to general (Apple -> Fruit)

Contravariance - General to specialized (Fruit -> Apple)

Invariance - Type conversion not possible (Apple -> Strawberry)

Invariance Between Generics

Generic types are invariant to one another, so the following Java code will not compile.

```
List<Apple> apples;
List<Fruit> fruits = ... ;
apples = fruits;    // error here
```

Bounded Type Parameters

Despite this invariance between generic types, you can still introduce covariant and contravariant relationships with Java generics.

This is accomplished with generic wildcards and **bounded type parameters**, which restrict the possible type arguments that can be passed to a type parameter.

Covariance with *Extends*

```
List<Apple> apples = new ArrayList<Apple>();
List<? extends Fruit> fruits = apples;
```

Contravariance with *Super*

```
List<Fruit> fruits = new ArrayList<Fruit>();
List<? super Apple> = fruits;
```

Before Generics

```
Vector v = new Vector();
v.add(new Apple());
v.add(new Strawberry());
Apple i = (Apple) v.get(0);
Strawberry bar = (Strawberry) v.get(1);
```

After Generics

```
List<Fruit> list = new ArrayList<Fruit>();
list.add(new Apple());
list.add(new Strawberry());

Apple i = list.get(0);
Strawberry bar = list.get(1);
```

Debugging

```
List<Fruit> list = new ArrayList<Fruit>();
list.add(new Integer(12));
```

This is an example of an error found at compile time due to generics.

Before Generics (Verbose)

```
List bowl = new ArrayList();
for (int i = 0; i < bowl.size(); i++) {
 if (!(bowl.get(i) instanceOf Fruit)) {
    continue;
  } else {
    Fruit fruit = (Fruit) bowl.get(i);
    eat(fruit);
  }
}
```

After Generics (Efficient)

```
List<Fruit> bowl = new ArrayList<Fruit>();
for(Fruit fruit:bowl) {
eat(fruit);
}
```

Pros and Cons

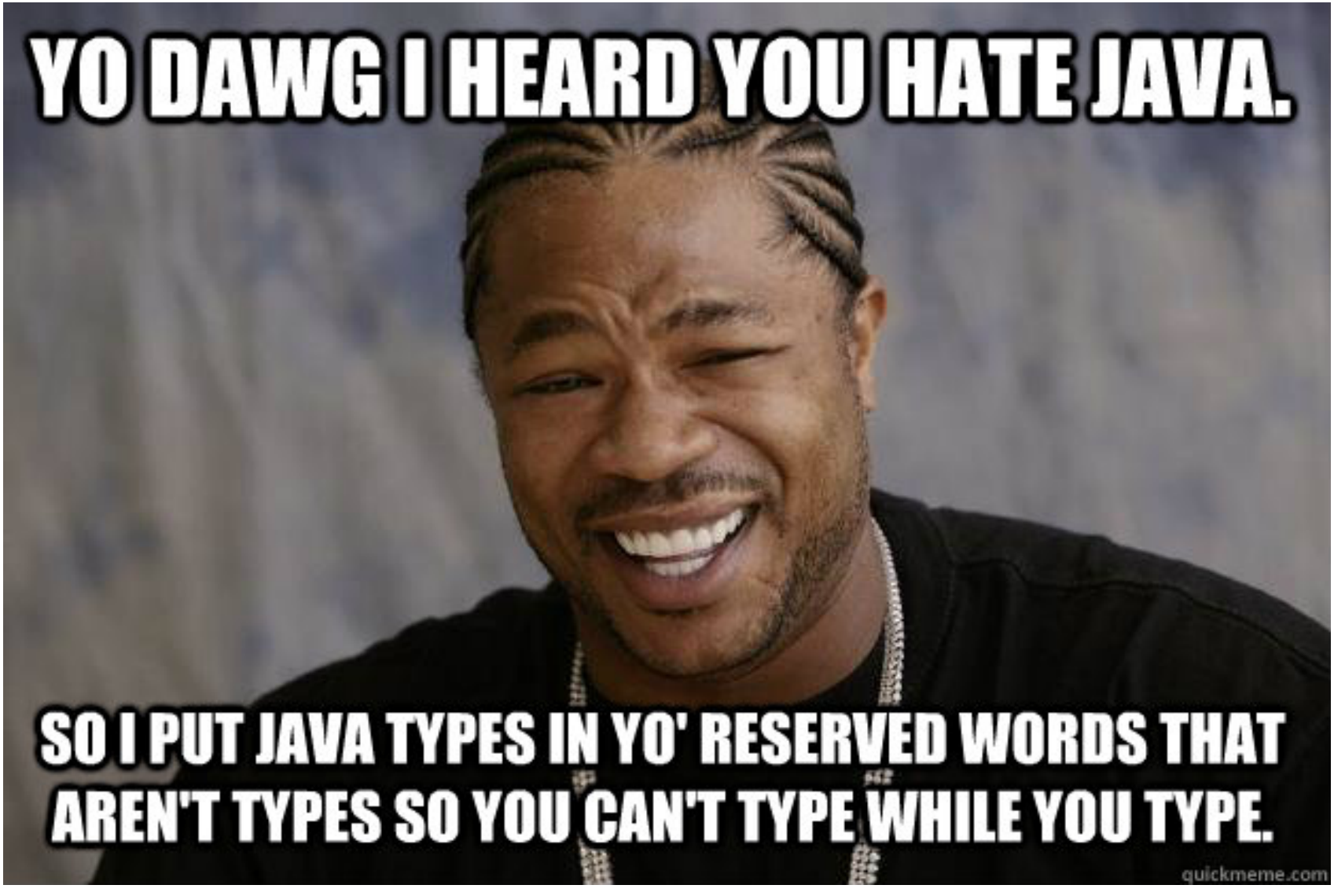
Pros

- Enhanced type safety
- Less code duplication,
- Less clutter from casting,
- Compile time errors more often

Cons

- Cannot instantiate generic types
- Cannot assign list of sub type to pointer to array of super type

Conclusion


YO DAWG I HEARD YOU HATE JAVA.
SO I PUT JAVA TYPES IN YO' RESERVED WORDS THAT AREN'T TYPES SO YOU CAN'T TYPE WHILE YOU TYPE.
quickmeme.com