Java Generics

Yann-Gaël Guéhéneuc

yann-gael.gueheneuc@polytmtl.ca

Version 1.0.1

2013/04/19

POLYTECHNIQUE
MONTRÉAL

WORLD-CLASS
ENGINEERING

Département de génie informatique et de génie logiciel

Any questions/comments are welcome at yann-gael.gueheneuc@polymtl.ca

Source code available at http://www.ptidej.net/tutorial/javagenerics

Problem

- Sorting lists does not and should not depend on the type of the elements stored in the list

```java
import java.util.List;

public interface ISort {
    public List sort(final List aList);
}
```

Problem

- Sorting lists does not and should not depend on the type of the elements stored in the list

```
import java.util.List;

public interface ISort {
    public List sort(final List aList);
}
```

Problem: elements in the list may not be comparable
Solution: generic typing to enforce elements to be Comparable

Problem

- Sorting lists assumes (and is sure) that the elements stored in the list are comparable

```java
import java.util.List;

public interface ISort<E extends Comparable<E>> {
    public List<E> sort(final List<E> aList);
}
```

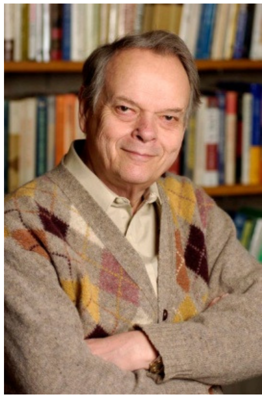
Outline

- History
- Problem
- Special Case
- General Definitions
- Generics Definitions
  - Parametric Polymorphism
  - Other Bounded Parametric Polymorphisms
- When to use generics
- How to use generics
- Caveats with generics
- Reflecting on generics
- Conclusion
- Few references

Outline

- **History**
- Problem
- Special Case
- General Definitions
- Generics Definitions
  - Parametric Polymorphism
  - Other Bounded Parametric Polymorphisms
- When to use generics
- How to use generics
- Caveats with generics
- Reflecting on generics
- Conclusion
- Few references

History



John C. Reynolds
*1935

- 1983: Reynolds formalises the parametricity theorem, called abstraction theorem
  - Functions with similar types have similar properties

John C. Reynolds ; “Types, abstraction, and parametric polymorphism” ; Information Processing ; pp. 513–523, North Holland, 1983.

History

- Parametric polymorphism

$$\text{append: } [a] \times [a] \rightarrow [a]$$

  - Expressiveness
  - Type-safety
    - First implementation in ML in 1989 (1976?)

Robin Milner, Robert Harper, David MacQueen, and Mads Tofte ; “The Definition Of Standard ML” ; The MIT Press, 1997.

History

Explicit parametric polymorphism

- Parametric polymorphism

append: [a] × [a] → [a]

  – Expressiveness

  – Type-safety

    • First implementation in ML in 1989 (1976?)

Robin Milner, Robert Harper, David MacQueen, and Mads Tofte ; “The Definition Of Standard ML” ; The MIT Press, 1997.

History

David Musser
*c.1945

Alexander Stepanov
*1950

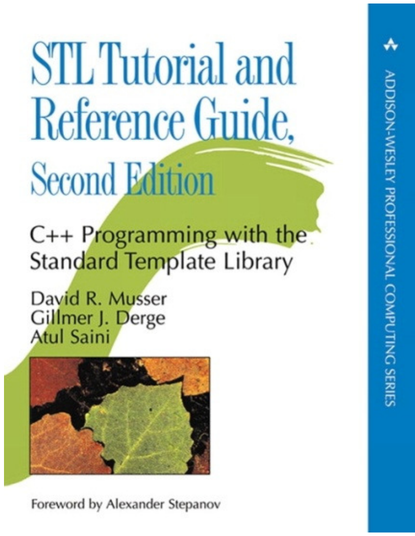
- 1988: David Musser and Alexander Stepanov define the concept of generic programming
  - Abstractions from examples of algorithms and data structure
  - Concept of “concept”

David R. Musser and Alexander A. Stepanov ; “Generic Programming” ; International symposium on Symbolic and Algebraic Computation, pp. 13-25, ACM Press, 1988.

History

> “Generic programming is about abstracting and classifying algorithms and data structures. […] Its goal is the incremental construction of systematic catalogs of useful, efficient and abstract algorithms and data structures.”
>
> —Alexander Stepanov

History



STL Tutorial and Reference Guide, Second Edition

C++ Programming with the Standard Template Library

David R. Musser
Gillmer J. Derge
Atul Saini

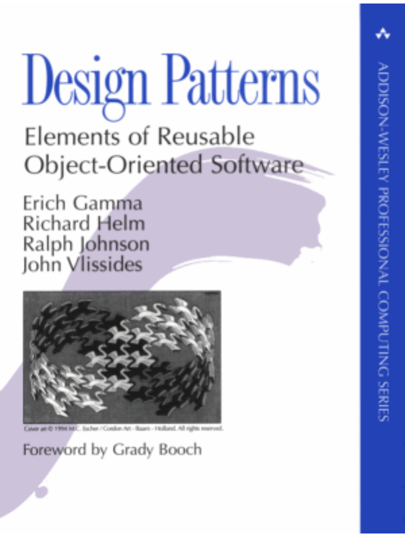
Foreword by Alexander Stepanov

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

- Generic programming
  - Theory of iterators
  - Independent of implementation
    - C++ Standard Template Library (STL)

```
const  ::std::vector<Foo>::iterator theEnd = theContainer.end();
for    ( ::std::vector<Foo>::iterator i = theContainer.begin();
         i != theEnd;
         ++i )
{
   Foo &cur_element = *i;
   // Do something…
}
```

History


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

- 1994: the GoF defines parameterized types

  “Also known as generics (Ada, Eiffel) and templates (C++)”

  “A type that leaves some constituent types unspecified. The unspecified types are supplied as parameters at the point of use.”

History

- 1977–1980: Ada
  - 2005: generic container library
- 1985: Eiffel

  Bertrand Meyer ; Object-Oriented Software Construction ; Prentice Hall, 1988.
- 1991: C++

  http://www.stroustrup.com/hopl2.pdf
  - 1994: STL (under Stepanov’s guidance)
- 2004: Java
  - Type erasure
- 2005: C#
  - Reified generics

Outline

- History
- **Problem**
- Special Case
- General Definitions
- Generics Definitions
  - Parametric Polymorphism
  - Other Bounded Parametric Polymorphisms
- When to use generics
- How to use generics
- Caveats with generics
- Reflecting on generics
- Conclusion
- Few references

Problem

“Implement generic algorithms that work on a collection of different types”

—The Java Tutorials, Oracle

http://docs.oracle.com/javase/tutorial/java/generics/why.html

Problem

- Sorting lists does not and should not depend on the type of the elements stored in the list

```
import java.util.List;

public interface ISort {
    public List sort(final List aList);
}
```

Problem

- Sorting lists does not and should not depend on the type of the elements stored in the list

```
import java.util.List;

public interface ISort {
    public List sort(final List aList);
}
```

Problem: elements in the list may not be comparable
Solution: generic typing to enforce elements to be Comparable

Problem

- Sorting lists does not and should not depend on the type of the elements stored in the list

```java
import java.util.List;

public interface ISort<E extends Comparable<E>> {
    public List<E> sort(final List<E> aList);
}
```

Outline

- History
- Problem
- **Special Case**
- General Definitions
- Generics Definitions
  - Parametric Polymorphism
  - Other Bounded Parametric Polymorphisms
- When to use generics
- How to use generics
- Caveats with generics
- Reflecting on generics
- Conclusion
- Few references

Special Case

```java
package ca.polymtl.ptidej.generics.java;

public class Example1 {
    public static void main(final String[] args) {
        final Object[] arrayOfObjects = new Object[10];
        final String[] arrayOfStrings = new String[20];

        System.out.println(arrayOfObjects.length);
        System.out.println(arrayOfStrings.length);

        System.out.println(arrayOfObjects[0]);
        System.out.println(arrayOfStrings[2]);

        System.out.println(arrayOfObjects.clone());
        System.out.println(arrayOfStrings.toString());
    }
}
```

Special Case

- Array are (often) predefined generic types

```java
final Object[] arrayOfObjects = new Object[10];
final String[] arrayOfStrings = new String[20];
```

Special Case

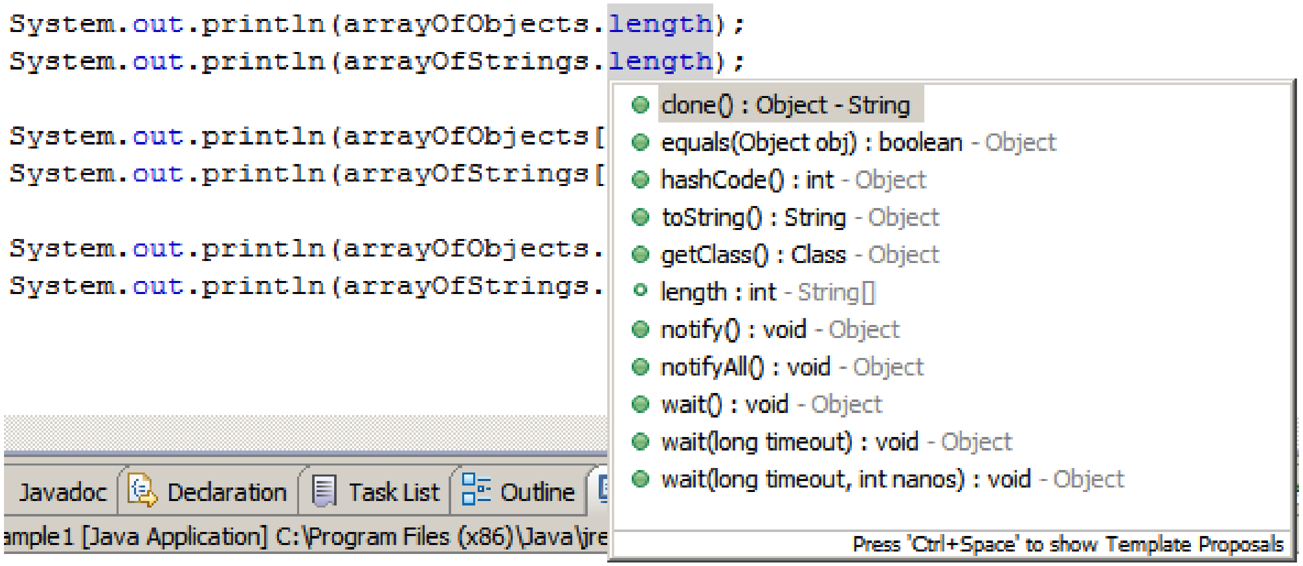
- Array are (often) predefined generic types

```
final Object[] arrayOfObjects = new Object[10];
final String[] arrayOfStrings = new String[20];
```

Any type can go here

Special Case

- Every new array instantiates a new concrete type (or reuse an existing concrete type)



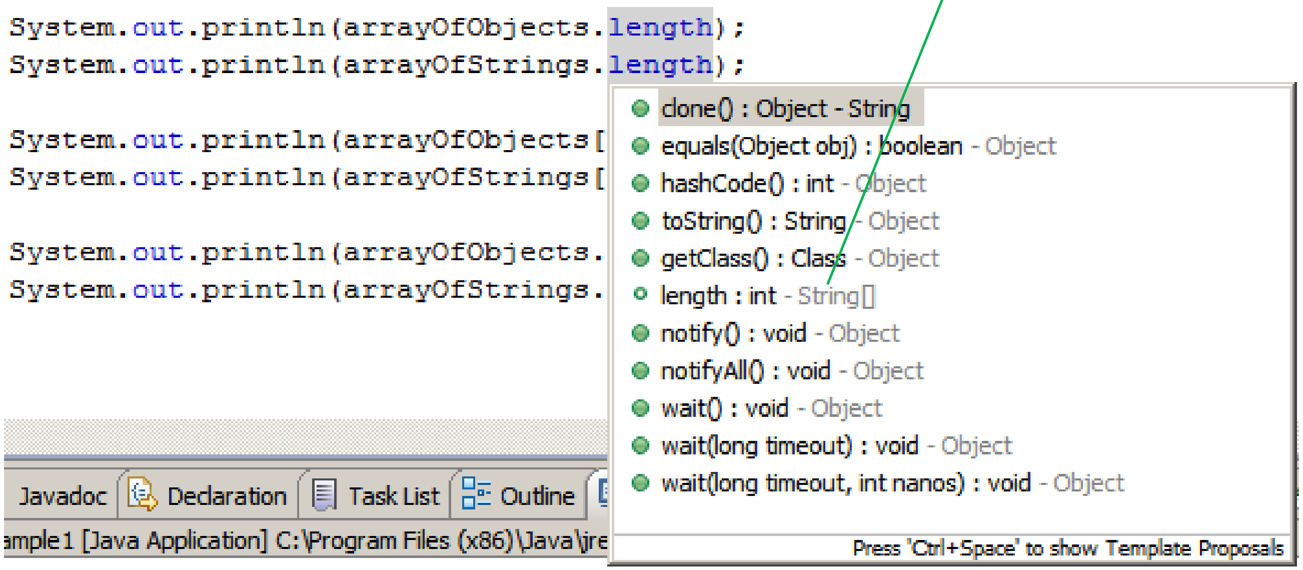
```
System.out.println(arrayOfObjects.length);
System.out.println(arrayOfStrings.length);

System.out.println(arrayOfObjects[
System.out.println(arrayOfStrings[

System.out.println(arrayOfObjects.
System.out.println(arrayOfStrings.
```

clone() : Object - String
equals(Object obj) : boolean - Object
hashCode() : int - Object
toString() : String - Object
getClass() : Class - Object
length : int - String[]
notify() : void - Object
notifyAll() : void - Object
wait() : void - Object
wait(long timeout) : void - Object
wait(long timeout, int nanos) : void - Object
Press 'Ctrl+Space' to show Template Proposals
Javadoc
Declaration
Task List
Outline
ample1 [Java Application] C:\Program Files (x86)\Java\jre

Special Case

New concrete type (pseudo-type in Java)

- Every new array instantiates a new concrete type (or reuse an existing concrete type)



```
System.out.println(arrayOfObjects.length);
System.out.println(arrayOfStrings.length);

System.out.println(arrayOfObjects[
System.out.println(arrayOfStrings[

System.out.println(arrayOfObjects.
System.out.println(arrayOfStrings.
```

clone() : Object - String
equals(Object obj) : boolean - Object
hashCode() : int - Object
toString() : String - Object
getClass() : Class - Object
length : int - String[]
notify() : void - Object
notifyAll() : void - Object
wait() : void - Object
wait(long timeout) : void - Object
wait(long timeout, int nanos) : void - Object
Press 'Ctrl+Space' to show Template Proposals
Javadoc
Declaration
Task List
Outline
ample1 [Java Application] C:\Program Files (x86)\Java\jre

Special Case

- Syntax and semantics built in the compiler

```
System.out.println(arrayOfObjects.length);
System.out.println(arrayOfStrings.length);

System.out.println(arrayOfObjects[0]);
System.out.println(arrayOfStrings[2]);

System.out.println(arrayOfObjects.clone());
System.out.println(arrayOfStrings.toString());
```

Pseudo-field

Special Case

- Syntax and semantics built in the compiler

```
System.out.println(arrayOfObjects.length);
System.out.println(arrayOfStrings.length);

System.out.println(arrayOfObjects[0]);
System.out.println(arrayOfStrings[2]);

System.out.println(arrayOfObjects.clone());
System.out.println(arrayOfStrings.toString());
```

Pseudo-field

Access, a[b]

Special Case

- Syntax and semantics built in the compiler

```
System.out.println(arrayOfObjects.length);
System.out.println(arrayOfStrings.length);

System.out.println(arrayOfObjects[0]);
System.out.println(arrayOfStrings[2]);

System.out.println(arrayOfObjects.clone());
System.out.println(arrayOfStrings.toString());
```

Special Case

- Syntax and semantics built in

Pseudo-field

Access, a[b]

> In the Java programming language arrays are objects (§4.3.1), are dynamically created, and may be assigned to variables of type Object (§4.3.2). All methods of class Object may be invoked on an array.
>
> —JLS

```
System.out.println(arrayOfObjects.length);
System.out.println(arrayOfStrings.length);

System.out.println(arrayOfObjects[0]);
System.out.println(arrayOfStrings[2]);

System.out.println(arrayOfObjects.clone());
System.out.println(arrayOfStrings.toString());
```

Outline

- History
- Problem
- Special Case
- **General Definitions**
- Generics Definitions
  - Parametric Polymorphism
  - Other Bounded Parametric Polymorphisms
- When to use generics
- How to use generics
- Caveats with generics
- Reflecting on generics
- Conclusion
- Few references

General Definitions

- Polymorphism
  - Ad-hoc polymorphism
  - Subtype polymorphism
  - Parametric polymorphism
    - Implicit
    - Explicit

General Definitions

- Ad-hoc polymorphism
  - Method overloading
  - Not a feature of the type system
  - Dispatch mechanism
    - Typically, dispatch depends on the concrete type of the receiver of a method

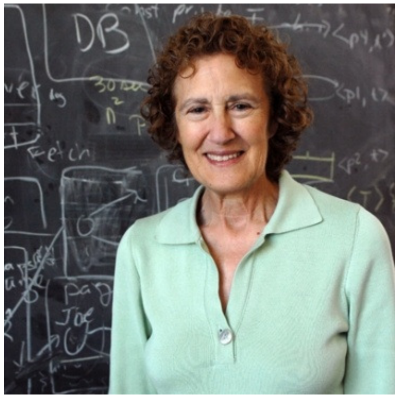
Christopher Strachey ; “Fundamental Concepts in Programming Languages” ; Higher-Order and Symbolic Computation, volume 13, issue 1-2, pp. 11-49, Springer, 2000.

General Definitions

- Ad-hoc polymorphism
  - A name may have more than one meaning
    - It may refer to more than one algorithm
  - The choice of the algorithm is context-dependent but know at compile-time

    (Early binding when compared to the following subtype polymorphism)

General Definitions



Barbara Liskov
*1939

- Subtype polymorphism
  - – Liskov substitution principle
    - **Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T**

(Late binding when compared to the previous ad hoc polymorphism)

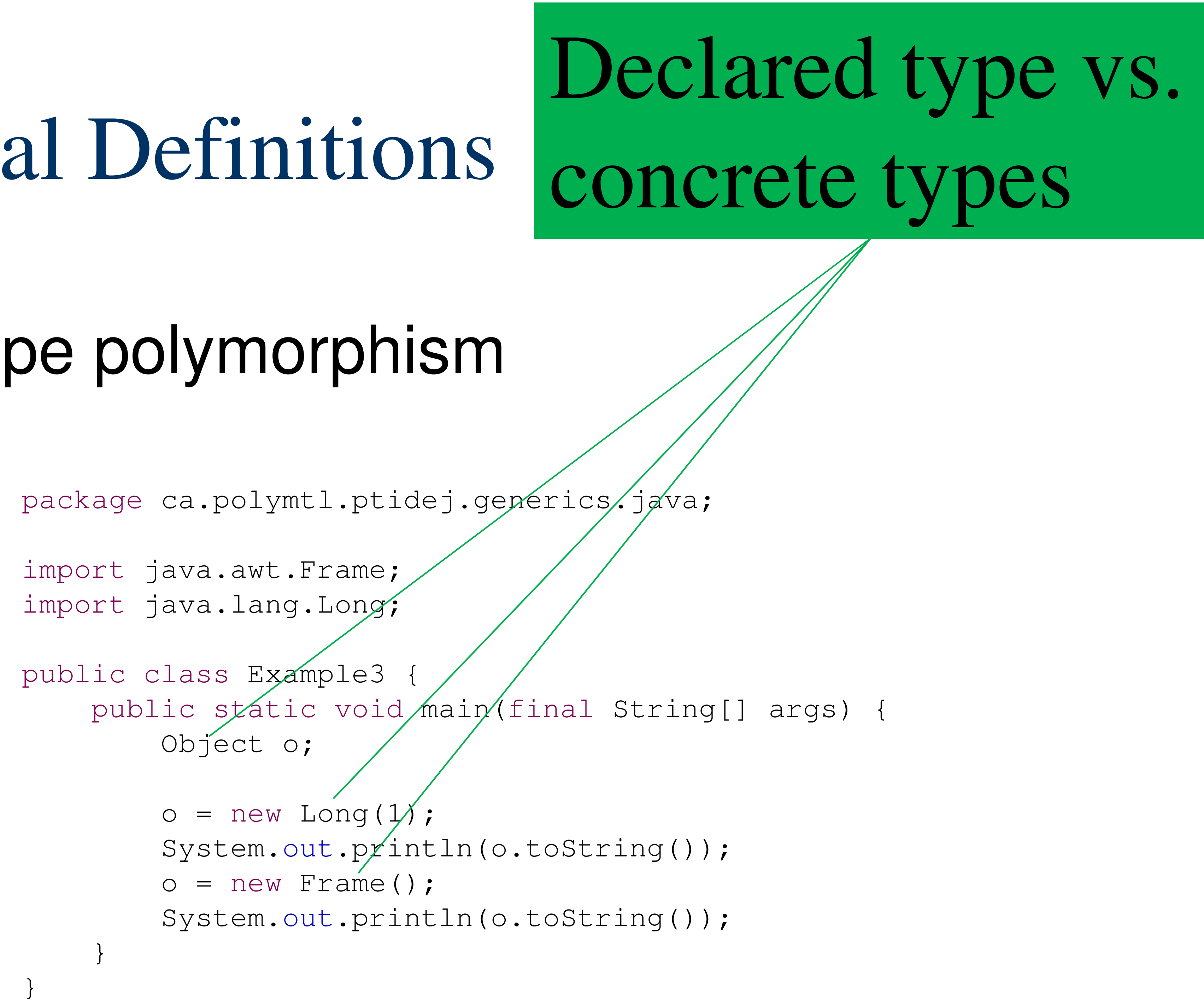
General Definitions

- Subtype polymorphism

```
package ca.polymtl.ptidej.generics.java;

import java.awt.Frame;
import java.lang.Long;

public class Example3 {
    public static void main(final String[] args) {
        Object o;

        o = new Long(1);
        System.out.println(o.toString());
        o = new Frame();
        System.out.println(o.toString());
    }
}
```

General Definitions

- Subtype polymorphism



Declared type vs. concrete types

```
package ca.polymtl.ptidej.generics.java;

import java.awt.Frame;
import java.lang.Long;

public class Example3 {
    public static void main(final String[] args) {
        Object o;

        o = new Long(1);
        System.out.println(o.toString());
        o = new Frame();
        System.out.println(o.toString());
    }
}
```

General Definitions

- Parametric polymorphism

```
public class NonGenericBox {
    private Object object;

    public void set(final Object object) {
        this.object = object;
    }
    public Object get() {
        return this.object;
    }
}

public void useOfNonGenericBox() {
    final NonGenericBox aNonGenericBox = new NonGenericBox();
    aNonGenericBox.set(new String());
    final String myString = (String) aNonGenericBox.get();
    System.out.println(myString);
}
```

General Definitions

Must cast to ask compiler to allow the assignment

- Parametric polymorphism

```
public class NonGenericBox {
    private Object object;

    public void set(final Object object) {
        this.object = object;
    }
    public Object get() {
        return this.object;
    }
}

public void useOfNonGenericBox() {
    final NonGenericBox aNonGenericBox = new NonGenericBox();
    aNonGenericBox.set(new String());
    final String myString = (String) aNonGenericBox.get();
    System.out.println(myString);
}
```

General Definitions

- Parametric polymorphism

```
public class ~~Non~~GenericBox<T> {
    private T object;

    public void set(final T object) {
        this.object = object;
    }
    public T get() {
        return this.object;
    }
}

public void useOfNonGenericBox() {
    final ~~Non~~GenericBox<String> a~~Non~~GenericBox = new ~~Non~~GenericBox<String>();
    a~~Non~~GenericBox.set(new String());
    final String myString = ~~(String)~~ a~~Non~~GenericBox.get();
    System.out.println(myString);
}
```

⚠ We use Java vocabulary in the following

Type parameter

General Definitions

- Parametric polymorphism

```
public class ~~Non~~GenericBox<T> {
    private T object;

    public void set(final T object) {
        this.object = object;
    }
    public T get() {
        return this.object;
    }
}

public void useOfNonGenericBox() {
    final ~~Non~~GenericBox<String> a~~Non~~GenericBox = new ~~Non~~GenericBox<String>();
    a~~Non~~GenericBox.set(new String());
    final String myString = ~~(String)~~ a~~Non~~GenericBox.get();
    System.out.println(myString);
}
```

⚠ We use Java vocabulary in the following

Type variable

Type parameter

General Definitions

- Parametric polymorphism

```
public class ~~Non~~GenericBox<T> {
    private T object;

    public void set(final T object) {
        this.object = object;
    }
    public T get() {
        return this.object;
    }
}

public void useOfNonGenericBox() {
    final ~~Non~~GenericBox<String> a~~Non~~GenericBox = new ~~Non~~GenericBox<String>();
    a~~Non~~GenericBox.set(new String());
    final String myString = ~~(String)~~ a~~Non~~GenericBox.get();
    System.out.println(myString);
}
```

⚠ We use Java vocabulary in the following

Type variable

Type parameter

Generic type declaration

General Definitions

- Parametric polymorphism

```
public class ~~Non~~GenericBox<T> {
    private T object;

    public void set(final T object) {
        this.object = object;
    }
    public T get() {
        return this.object;
    }
}

public void useOfNonGenericBox() {
    final ~~Non~~GenericBox<String> a~~Non~~GenericBox = new ~~Non~~GenericBox<String>();
    a~~Non~~GenericBox.set(new String());
    final String myString = ~~(String)~~ a~~Non~~GenericBox.get();
    System.out.println(myString);
}
```

⚠ We use Java vocabulary in the following

General Definitions

Type variable

Type parameter

Generic type declaration

Parameterised methods

- Parametric polymorphism

```
public class ~~Non~~GenericBox<T> {
    private T object;

    public void set(final T object) {
        this.object = object;
    }
    public T get() {
        return this.object;
    }
}

public void useOfNonGenericBox() {
    final ~~Non~~GenericBox<String> a~~Non~~GenericBox = new ~~Non~~GenericBox<String>();
    a~~Non~~GenericBox.set(new String());
    final String myString = ~~(String)~~ a~~Non~~GenericBox.get();
    System.out.println(myString);
}
```

⚠ We use Java vocabulary in the following

General Definitions

Type variable

Type parameter

Generic type declaration

Parameterised methods

Type argument

- Parametric polymorphism

```
public class ~~Non~~GenericBox<T> {
    private T object;

    public void set(final T object) {
        this.object = object;
    }
    public T get() {
        return this.object;
    }
}

public void useOfNonGenericBox() {
    final ~~Non~~GenericBox<String> a~~Non~~GenericBox = new ~~Non~~GenericBox<String>();
    a~~Non~~GenericBox.set(new String());
    final String myString = ~~(String)~~ a~~Non~~GenericBox.get();
    System.out.println(myString);
}
```

⚠ We use Java vocabulary in the following

General Definitions

- Parametric polymorphism

```
public class GenericBox<T> {
    private T t;

    public void set(final T t) {
        this.t = t;
    }
    public T get() {
        return this.t;
    }
}

public void useOfGenericBox() {
    final GenericBox<String> aGenericBox = new GenericBox<String>();
    aGenericBox.set(new String());
    final String myString = aGenericBox.get();
    System.out.println(myString);
}
```

General Definitions

- Parametric polymorphism

```java
package ca.polymtl.ptidej.generics.java;

public class Example4 {
    public static void main(final String[] args) {
        System.out.println(Util.<String>compare("a", "b"));
        System.out.println(Util.<String>compare(new String(""), new Long(1)));
        System.out.println(Util.compare(new String(""), new Long(1)));
    }
}

public class Util {
    public static <T> boolean compare(T t1, T t2) {
        return t1.equals(t2);
    }
}
```

General Definitions

- Parametric polymorphism

```java
package ca.polymtl.ptidej.generics.java;

public class Example4 {
    public static void main(final String[] args) {
        System.out.println(Util.<String>compare("a", "b"));
        System.out.println(Util.<String>compare(new String(""), new Long(1)));
        System.out.println(Util.compare(new String(""), new Long(1)));
    }
}

public class Util {
    public static <T> boolean compare(T t1, T t2) {
        return t1.equals(t2);
    }
}
```

Generic method

General Definitions

- Parametric polymorphism

```java
package ca.polymtl.ptidej.generics.java;

public class Example4 {
    public static void main(final String[] args) {
        System.out.println(Util.<String>compare("a", "b"));
        System.out.println(Util.<String>compare(new String(""), new Long(1)));
        System.out.println(Util.compare(new String(""), new Long(1)));
    }
}

public class Util {
    public static <T> boolean compare(T t1, T t2) {
        return t1.equals(t2);
    }
}
```

Explicit calls

Generic method

General Definitions

- Parametric polymorphism

```
package ca.polymtl.ptidej.generics.java;

public class Example4 {
    public static void main(final String[] args) {
        System.out.println(Util.<String>compare("a", "b"));
        System.out.println(Util.<String>compare(new String(""), new Long(1)));
        System.out.println(Util.compare(new String(""), new Long(1)));
    }
}

public class Util {
    public static <T> boolean compare(T t1, T t2) {
        return t1.equals(t2);
    }
}
```

Explicit calls

Implicit call

Generic method

Outline

- History
- Problem
- Special Case
- General Definitions
- **Generics Definitions**
  - Parametric Polymorphism
  - Other Bounded Parametric Polymorphisms
- When to use generics
- How to use generics
- Caveats with generics
- Reflecting on generics
- Conclusion
- Few references

Generics Definitions

> “A *generic type* is a generic class or interface that is parameterized over types.”
>
> —The Java Tutorials, Oracle

Generics Definitions

- Java generics are one implementation of parametric polymorphism
  - Type erasure

- Type parameters can be constrained
  - Lower bounds
  - Upper bounds

  to obtain bounded type parameters

Outline

- History
- Problem
- Special Case
- General Definitions
- Generics Definitions
  - **Parametric Polymorphism**
  - Other Bounded Parametric Polymorphisms
- When to use generics
- How to use generics
- Caveats with generics
- Reflecting on generics
- Conclusion
- Few references

Generics Definitions

- Parametric polymorphism
  - Predicative
    - ML
  - Impredicative
    - System F
    - C++, Java 1.5
  - Bounded
    - C++ in one way, Java 1.5 in another

Martín Abadi, Luca Cardelli, Pierre-Louis Curien ; “Formal Parametric Polymorphism” ; SRC research report, issue 109, Digital, Systems Research Center, 1993.

Generics Definitions

- Predicative parametric polymorphism
  - A type T containing a type variable α may not be used in such a way that α is instantiated to a polymorphic type

Generics Definitions

- Predicative parametric polymorphism
  - A type T containing a type variable α may not be used in such a way that α is instantiated to a polymorphic type

```java
final GenericBox<String> aGenericBox = new GenericBox<String>();
aGenericBox.set(new String());
```

```java
final GenericBox<List<String>> aGenericBox = new GenericBox<List<String>>();
aGenericBox.set(new String());
```

Generics Definitions

- Predicative parametric polymorphism
  - A type T containing a type variable α may not be used in such a way that α is instantiated to a polymorphic type

```
final GenericBox<String> aGenericBox = new GenericBox<String>();
aGenericBox.set(new String());
```

```
final GenericBox<List<String>> aGenericBox = new GenericBox<List<String>>();
aGenericBox.set(new String());
```

Generics Definitions

- Impredicative parametric polymorphism
  - Example 1
  - Example 2

Generics Definitions

- Impredicative parametric polymorphism
  - Example 1

```
final GenericBox<List<String>> aGenericBox = new GenericBox<List<String>>();
aGenericBox.set(new String());
```

  - Example 2

Generics Definitions

- Impredicative parametric polymorphism
  - Example 1

```java
final GenericBox<List<String>> aGenericBox = new GenericBox<List<String>>();
aGenericBox.set(new String());
```

  - Example 2

```java
import java.util.List;

public interface ISort<E extends Comparable<E>> {
    public List<E> sort(final List<E> aList);
}
```

Generics Definitions

- Bounded parametric polymorphism

```java
import java.util.List;

public interface ISort<E extends Comparable<E>> {
    public List<E> sort(final List<E> aList);
}
```

The type `E` of the list elements must implement the interface `Comparable`

Generics Definitions

- Bounded parametric polymorphism

> “Bounded genericity is less about limiting the types accepted by [a] generic class […] and more about giving the generic class a more complete information on its generic type T […] to validate the call to its methods at compile time.”
>
> —paercebal

http://stackoverflow.com/questions/6803100/achieving-bounded-genericity-in-c/6803124

Generics Definitions

```
public class Example5 {
    public static void main(final String[] args) {
        final Sort<A> sort = new Sort<A>();
        final List<A> listOfAs = new ArrayList<A>();
        sort.sort(listOfAs);
        System.out.println();
    }
}
class Sort<E extends Comparable<E>> {
    public List<E> sort(final List<E> aList) {
        return // TO DO
    }
}
class A implements Comparable<A> {
    public int compareTo(final A o) {
        return // TO DO
    }
}
class B implements Comparable<B> {
    public int compareTo(final B o) {
        return // TO DO
    }
}
```

Generics Definitions

```
public class Example5 {
    public static void main(final String[] args) {
        final Sort<A> sort = new Sort<A>();
        final List<A> listOfAs = new ArrayList<A>();
        sort.sort(listOfAs);
        System.out.println();
    }
}
class Sort<E extends Comparable<E>> {
    public List<E> sort(final List<E> aList) {
        return // TO DO
    }
}
class A implements Comparable<A> {
    public int compareTo(final A o) {
        return // TO DO
    }
}
class B implements Comparable<B> {
    public int compareTo(final B o) {
        return // TO DO
    }
}
```

Must be comparable (with itself)

Outline

- History
- Problem
- Special Case
- General Definitions
- Generics Definitions
  - Parametric Polymorphism
  - **Other Bounded Parametric Polymorphisms**
- When to use generics
- How to use generics
- Caveats with generics
- Reflecting on generics
- Conclusion
- Few references

Generics Definitions

- Other bounded parametric polymorphisms

Java

C++

Generics Definitions

- Other bounded parametric polymorphisms

> “This feature is provided as-is and where-used by the compiler: in a way similar to duck typing, but resolved at compile-time. [Compilation succeeds] only if the generic type class [declares] the [expected method].”
>
> —paercebal

http://stackoverflow.com/questions/6803100/achieving-bounded-genericity-in-c/6803124

Generics Definitions

```
class X {
    public:
    virtual void kewl_method() { /* etc. */ }
};
class Y: public X {
    public:
    virtual void kewl_method() { /* etc. */ }
};
class Z {
    public:
    virtual void kewl_method() { /* etc. */ }
};
class K {
    public:
    virtual void wazaa() { /* etc. */ }
};

template<typename T>
class A {
    public:
    void foo() {
        T t;
        t.kewl_method();
    }
};
```

Generics Definitions

```
class X {
    public:
    virtual void kewl_method() { /* etc. */ }
};
class Y: public X {
    public:
    virtual void kewl_method() { /* etc. */ }
};
class Z {
    public:
    virtual void kewl_method() { /* etc. */ }
};
class K {
    public:
    virtual void wazaa() { /* etc. */ }
};

template<typename T>
class A {
    public:
    void foo() {
        T t;
        t.kewl_method();
    }
};
```

No common type

Generics Definitions

```
class X {
    public:
    virtual void kewl_method() { /* etc. */ }
};
class Y: public X {
    public:
    virtual void kewl_method() { /* etc. */ }
};
class Z {
    public:
    virtual void kewl_method() { /* etc. */ }
};
class K {
    public:
    virtual void wazaa() { /* etc. */ }
};

template<typename T>
class A {
    public:
    void foo() {
        T t;
        t.kewl_method();
    }
};
```

Common API

Generics Definitions

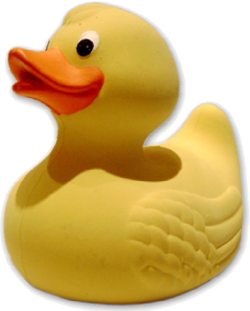
```
int main()
{
  // A's constraint is : implements kewl_method
  A<X> x ; x.foo() ;
  // OK: x implements kewl_method

  A<Y> y ; y.foo() ;
  // OK: y derives from X

  A<Z> z ; z.foo() ;
  // OK: z implements kewl_method

  A<K> k ; k.foo() ;
  // NOT OK : K won't compile: /main.cpp error:
  // ‘class K’ has no member named ‘kewl_method’

  return 0;
}
```

Generics Definitions

```
int main()
{
    // A's constraint is : implements kewl_method
    A<X> x ; x.foo() ;
    // OK: x implements kewl_method

    A<Y> y ; y.foo() ;
    // OK: y derives from X

    A<Z> z ; z.foo() ;
    // OK: z implements kewl_method

    A<K> k ; k.foo() ;
    // NOT OK : K won't compile: /main.cpp error:
    // ‘class K’ has no member named ‘kewl_method’

    return 0;
}
```

“Static” duct typing

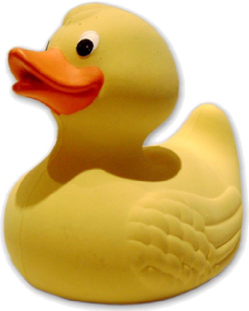
Generics Definitions



- Duck typing
  - Dynamically-typed languages: Smalltalk
  - Statically-typed language: C++

> “When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.”
>
> —Alex Martelli or James W. Riley

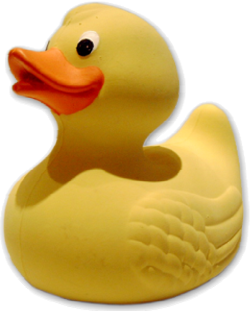
Generics Definitions



- Dynamically-typed languages: Smalltalk

```
Object subclass: #D
         instanceVariableNames: ''
         classVariableNames: ''
         poolDictionaries: ''
         category: 'CSE3009'.

D compile: 'needAFooMethod: anObjectWithaFooMethod
         "Example of duck typing"
         anObjectWithaFooMethod foo.'.
```

Generics Definitions

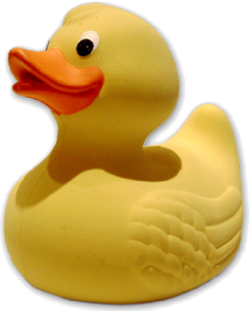


- Dynamically-typed languages: Smalltalk

```
Object subclass: #D
         instanceVariableNames: ''
         classVariableNames: ''
         poolDictionaries: ''
         category: 'CSE3009'.

D compile: 'needAFooMethod: anObjectWithaFooMethod
         "Example of duck typing"
         anObjectWithaFooMethod foo.'.
```

Any object with a `foo` method will do

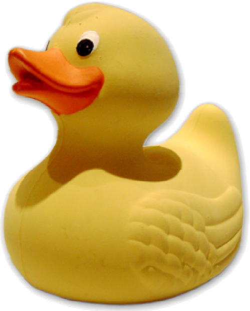
Generics Definitions



- Dynamically-typed languages: Smalltalk

```
SMUtilities subclass: #D1
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'CSE3009'.

D1 compile: 'foo
        Transcript show: ''D1'' ; cr.'.

PointArray variableWordSubclass: #D2
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'CSE3009'.

D2 compile: 'foo
        Transcript show: ''D2'' ; cr.'.
```

Generics Definitions

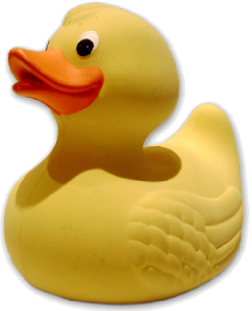


- Dynamically-typed languages: Smalltalk

```
SMUtilities subclass: #D1
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'CSE3009'.

D1 compile: 'foo
        Transcript show: ''D1'' ; cr.'.

PointArray variableWordSubclass: #D2
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'CSE3009'.

D2 compile: 'foo
        Transcript show: ''D2'' ; cr.'.
```

Two unrelated classes

Generics Definitions



- Dynamically-typed languages: Smalltalk

```
d := D new.
d needAFooMethod: (D1 new).
d needAFooMethod: (D2 new).
```

```
D1
D2
```

Outline

- History
- Problem
- Special Case
- General Definitions
- Generics Definitions
  - Parametric Polymorphism
  - Other Bounded Parametric Polymorphisms
- **When to use generics**
- How to use generics
- Caveats with generics
- Reflecting on generics
- Conclusion
- Few references

When to Use Generics

- Scenario 1: you want to enforce type safety for containers and remove the need for typecasts when using these containers

```
public final class Example1 {
    public static void main(final String[] args) {
        final List untypedList = new ArrayList();
        untypedList.add(new String());
        final Integer i = (Integer) untypedList.get(0);

        final List<String> typedList = new ArrayList<String>();
        typedList.add(new String());
        final Integer i = (Integer) typedList.get(0);
    }
}
```

Does not compile

When to Use Generics

- Scenario 2: you want to build generic algorithms that work on several types of (possible unrelated) things

```java
import java.util.List;

public interface ISort<E extends Comparable<E>> {
    public List<E> sort(final List<E> aList);
}
```

Outline

- History
- Problem
- Special Case
- General Definitions
- Generics Definitions
  - Parametric Polymorphism
  - Other Bounded Parametric Polymorphisms
- When to use generics
- **How to use generics**
- Caveats with generics
- Reflecting on generics
- Conclusion
- Few references

How to Use Generics

- Lots of resources
- Lots of discussions

- First step http://docs.oracle.com/javase/tutorial/java/generics/index.html
- Then, http://stackoverflow.com/search?q=%22java+generics%22
  - 1,323 results as of 2013/04/14

How to Use Generics

- Typed containers, before

```java
import java.util.ArrayList;
import java.util.List;

public final class Example1Before {
    public static void main(final String[] args) {
        final List untypedList = new ArrayList();
        untypedList.add(new String());
        final Integer i = (Integer) untypedList.get(0);
    }
}
```

How to Use Generics

- Typed containers, what happens?

```java
import java.util.ArrayList;
import java.util.List;

public final class Example1Before {
    public static void main(final String[] args) {
        final List untypedList = new ArrayList();
        untypedList.add(new String());
        final Integer i = (Integer) untypedList.get(0);
    }
}
```

How to Use Generics

- Typed containers, what happens?

```java
import java.util.ArrayList;
import java.util.List;

public final class Example1Before {
    public static void main(final String[] args) {
        final List untypedList = new ArrayList();
        untypedList.add(new String());
        final Integer i = (Integer) untypedList.get(0);
    }
}
```

```
Exception in thread "main" java.lang.ClassCastException:
    java.lang.String cannot be cast to java.lang.Integer
    at ca.polymtl.ptidej.generics.java.Example1Before.main(Example1Before.java:29)
```

How to Use Generics

- Typed containers, another look

```java
import java.util.ArrayList;
import java.util.List;

public final class Example1Before {
    public static void main(final String[] args) {
        final List untypedList = new ArrayList();
        untypedList.add(new String());
        final Integer i = (Integer) untypedList.get(0);
    }
}
```

How to Use Generics

- Typed containers, another look

```
import java.util.ArrayList;
import java.util.List;

public final class Example1Before {
    public static void main(final String[] args) {
        final List untypedList = new ArrayList();
        untypedList.add(new String());
        final Integer i = (Integer) untypedList.get(0);
    }
}
```

`List` and `ArrayList` are raw types, compiler cannot typecheck

How to Use Generics

- Typed containers, solution

```java
import java.util.ArrayList;
import java.util.List;

public final class Example1Before {
    public static void main(final String[] args) {
        final List<String> typedList = new ArrayList<String>();
        typedList.add(new String());
        final Integer i = (Integer) typedList.get(0);
    }
}
```

How to Use Generics

- Typed containers, solution

```
import java.util.ArrayList;
import java.util.List;

public final class Example1Before {
    public static void main(final String[] args) {
        final List<String> typedList = new ArrayList<String>();
        typedList.add(new String());
        final Integer i = (Integer) typedList.get(0);
    }
}
```

Does not compile because `String` and `Interger` are not compatible

How to Use Generics

- Family of algorithms, before

```
public interface Enumeration {
    /**
     * Tests if this enumeration contains more elements.
     *
     * @return  <code>true</code> if and only if this enumeration object
     *           contains at least one more element to provide;
     *          <code>false</code> otherwise.
     */
    boolean hasMoreElements();

    /**
     * Returns the next element of this enumeration if this enumeration
     * object has at least one more element to provide.
     *
     * @return     the next element of this enumeration.
     * @exception  NoSuchElementException  if no more elements exist.
     */
    Object nextElement();
}
```

How to Use Generics

- Family of algorithms, what happens?

```
public interface Enumeration {
    /**
     * Tests if this enumeration contains more elements.
     *
     * @return  <code>true</code> if and only if this enumeration object
     *           contains at least one more element to provide;
     *          <code>false</code> otherwise.
     */
    boolean hasMoreElements();

    /**
     * Returns the next element of this enumeration if this enumeration
     * object has at least one more element to provide.
     *
     * @return     the next element of this enumeration.
     * @exception  NoSuchElementException  if no more elements exist.
     */
    Object nextElement();
}
```

How to Use Generics

Forces clients to use `Object`

- Family of algorithms, what happens?

```
public interface Enumeration {
    /**
     * Tests if this enumeration contains more elements.
     *
     * @return  <code>true</code> if and only if this enumeration object
     *           contains at least one more element to provide;
     *          <code>false</code> otherwise.
     */
    boolean hasMoreElements();

    /**
     * Returns the next element of this enumeration if this enumeration
     * object has at least one more element to provide.
     *
     * @return     the next element of this enumeration.
     * @exception  NoSuchElementException  if no more elements exist.
     */
    Object nextElement();
}
```

How to Use Generics

- Family of algorithms, another look

```
public interface Enumeration {
    /**
     * Tests if this enumeration contains more elements.
     *
     * @return  <code>true</code> if and only if this enumeration object
     *           contains at least one more element to provide;
     *          <code>false</code> otherwise.
     */
    boolean hasMoreElements();

    /**
     * Returns the next element of this enumeration if this enumeration
     * object has at least one more element to provide.
     *
     * @return     the next element of this enumeration.
     * @exception  NoSuchElementException  if no more elements exist.
     */
    Object nextElement();
}
```

How to Use G

Clients must know the type of the next element

- Family of algorithms, another look

```
public interface Enumeration {
    /**
     * Tests if this enumeration contains more elements.
     *
     * @return  <code>true</code> if and only if this enumeration object
     *           contains at least one more element to provide;
     *          <code>false</code> otherwise.
     */
    boolean hasMoreElements();

    /**
     * Returns the next element of this enumeration if this enumeration
     * object has at least one more element to provide.
     *
     * @return     the next element of this enumeration.
     * @exception  NoSuchElementException  if no more elements exist.
     */
    Object nextElement();
}
```

How to Use Generics

- Family of algorithms, solution

```
public interface Enumeration<E> {
    /**
     * Tests if this enumeration contains more elements.
     *
     * @return  <code>true</code> if and only if this enumeration object
     *           contains at least one more element to provide;
     *          <code>false</code> otherwise.
     */
    boolean hasMoreElements();

    /**
     * Returns the next element of this enumeration if this enumeration
     * object has at least one more element to provide.
     *
     * @return     the next element of this enumeration.
     * @exception  NoSuchElementException  if no more elements exist.
     */
    E nextElement();
}
```

How to Use Generics

- Family of algorithms, solution

```
public interface Enumeration<E> {
    /**
     * Tests if this enumeration contains more elements.
     *
     * @return  <code>true</code> if and only if this enumeration object
     *           contains at least one more element to provide;
     *          <code>false</code> otherwise.
     */
    boolean hasMoreElements();

    /**
     * Returns the next element of this enumeration if this enumeration
     * object has at least one more element to provide.
     *
     * @return     the next element of this enumeration.
     * @exception  NoSuchElementException  if no more elements exist.
     */
    E nextElement();
}
```

How to Use G

Clients can specify the type of the next element

- Family of algorithms, solution

```
public interface Enumeration<E> {
    /**
     * Tests if this enumeration contains more elements.
     *
     * @return  <code>true</code> if and only if this enumeration object
     *           contains at least one more element to provide;
     *          <code>false</code> otherwise.
     */
    boolean hasMoreElements();

    /**
     * Returns the next element of this enumeration if this enumeration
     * object has at least one more element to provide.
     *
     * @return     the next element of this enumeration.
     * @exception  NoSuchElementException  if no more elements exist.
     */
    E nextElement();
}
```

Outline

- History
- Problem
- Special Case
- General Definitions
- Generics Definitions
  - Parametric Polymorphism
  - Other Bounded Parametric Polymorphisms
- When to use generics
- How to use generics
- **Caveats with generics**
- Reflecting on generics
- Conclusion
- Few references

Caveats with Generics

- `int`s and `Integer`s, before

```
public interface List extends Collection {
    ...
    boolean add(Object o);
    boolean remove(Object o);
    Object remove(int index);
    ...
}
```

Caveats with Generics

- `int`s and `Integer`s, now

```
public interface List<E> extends Collection<E> {
    ...
    boolean add(E e);
    boolean remove(Object o);
    E remove(int index);
    ...
}
```

Caveats with Generics

- `int`s and `Integer`s, now

```
public interface List<E> extends Collection<E> {
     ...
     boolean add(E e);
     boolean remove(Object o);
     E remove(int index);
     ...
}
```

Caveats with Generics

- `int`s and `Integer`s, what happens?

```java
import java.util.ArrayList;
import java.util.List;

public class Autoboxing {
    public static void main(String[] args) {
        final List<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(new Integer(2));

        list.remove(1);
        list.remove(new Integer(1));

        System.out.println(list.size());
    }
}
```

Caveats with Generi

Autoboxing from `int` to `Integer`

- `int`s and `Integer`s, what happens?

```
import java.util.ArrayList;
import java.util.List;

public class Autoboxing {
    public static void main(String[] args) {
        final List<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(new Integer(2));

        list.remove(1);
        list.remove(new Integer(1));

        System.out.println(list.size());
    }
}
```

Exact parameter matching takes over autoboxing

eneri

Autoboxing from `int` to `Integer`

- `int`s and `Integer`s, what happens?

```
import java.util.ArrayList;
import java.util.List;

public class Autoboxing {
    public static void main(String[] args) {
        final List<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(new Integer(2));

        list.remove(1);
        list.remove(new Integer(1));

        System.out.println(list.size());
    }
}
```

Exact parameter matching takes over autoboxing

eneri

Autoboxing from `int` to `Integer`

- ints and Integers, what happens?

```
import java.util.ArrayList;
import java.util.List;

public class Autoboxing {
    public static void main(String[] args) {
        final List<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(new Integer(2));

        list.remove(1);
        list.remove(new Integer(1));

        System.out.println(list.size());
    }
}
```

0

Caveats with Generics

- Use of `clone()`, before

```java
import java.util.ArrayList;

public class CloningBefore {
    public static void main(final String[] args) {
        final ArrayList list1 = new ArrayList();
        list1.add(new Integer(1));
        list1.add(new Integer(2));

        final ArrayList list2 = (ArrayList) list1.clone();
        System.out.println(list2);
    }
}
```

http://stackoverflow.com/questions/3941850/java-how-to-use-clone-and-what-about-the-cast-check

Caveats with Generics

No complains for the compiler

- Use of `clone()`, before

```java
import java.util.ArrayList;

public class CloningBefore {
    public static void main(final String[] args) {
        final ArrayList list1 = new ArrayList();
        list1.add(new Integer(1));
        list1.add(new Integer(2));

        final ArrayList list2 = (ArrayList) list1.clone();
        System.out.println(list2);
    }
}
```

http://stackoverflow.com/questions/3941850/java-how-to-use-clone-and-what-about-the-cast-check

Caveats with Generics

- Use of `clone()`, now

```
import java.util.ArrayList;

public class CloningNow {
    public static void main(final String[] args) {
        final ArrayList<Integer> list1 = new ArrayList<Integer>();
        list1.add(1);
        list1.add(new Integer(2));

        final ArrayList<Integer> list2 = (ArrayList<Integer>) list1.clone();
        System.out.println(list2);
    }
}
```

Caveats with Generics

- Use of `clone()`, now

```
import java.util.ArrayList;

public class CloningNow {
    public static void main(final String[] args) {
        final ArrayList<Integer> list1 = new ArrayList<Integer>();
        list1.add(1);
        list1.add(new Integer(2));

        final ArrayList<Integer> list2 = (ArrayList<Integer>) list1.clone();
        System.out.println(list2);
    }
}
```

Type safety: Unchecked cast from Object to ArrayList<Integer>

- Use of clone(), now

```
import java.util.ArrayList;

public class CloningNow {
    public static void main(final String[] args) {
        final ArrayList<Integer> list1 = new ArrayList<Integer>();
        list1.add(1);
        list1.add(new Integer(2));

        final ArrayList<Integer> list2 = (ArrayList<Integer>) list1.clone();
        System.out.println(list2);
    }
}
```

Caveats with Generics

- Use of `clone()`, what happens?
  - Compiler is now “stricter”
  - Compiler warns of a type-unsafe operation

Caveats with Generics

- Use of `clone()`, solution
  - Use copy-constructor

```
import java.util.ArrayList;

public class CloningSolution {
    public static void main(final String[] args) {
        final ArrayList<Integer> list1 = new ArrayList<Integer>();
        list1.add(1);
        list1.add(new Integer(2));

        final ArrayList<Integer> list2 = new ArrayList<Integer>(list1);
        System.out.println(list2);
    }
}
```

to obtain type-safety and remove any warning

Caveats with Generics

- Use of `clone()`, solution
  - Use copy-constructor

```
import java.util.ArrayList;

public class CloningSolution {
    public static void main(final String[] args) {
        final ArrayList<Integer> list1 = new ArrayList<Integer>();
        list1.add(1);
        list1.add(new Integer(2));

        final ArrayList<Integer> list2 = new ArrayList<Integer>(list1);
        System.out.println(list2);
    }
}
```

to obtain type-safety and remove any warning

Caveats with Generics

- Instantiating a type variable, problem

```
public class InstantiatingTypeParameterProblem<T> {
    public static void main(final String[] args) {
        ...
    }
    public T getInstanceOfT (){
        // Neither lines work:
        return new T();
        return T.newInstance();
    }
    ...
}
```

Caveats with Generi

Cannot instantiate the type T

- Instantiating a type variable, problem

```
public class InstantiatingTypeParameterProblem<T> {
    public static void main(final String[] args) {
        ...
    }
    public T getInstanceOfT (){
        // Neither lines work:
        return new T();
        return T.newInstance();
    }
    ...
}
```

Caveats with Generi

Cannot instantiate the type `T`

- Instantiating a type variable, problem

```
public class InstantiatingTypeParameterProblem<T> {
    public static void main(final String[] args) {
        ...
    }
    public T getInstanceOfT (){
        // Neither lines work:
        return new T();
        return T.newInstance();
    }
    ...
}
```

The method `newInstance()` is undefined for the type `T`

Caveats with Generics

- Instantiating a type variable, what happens?

```
public class InstantiatingTypeParameterProblem<T> {
    public static void main(final String[] args) {
        ...
    }
    public T getInstanceOfT (){
        // Neither lines work:
        return new T();
        return T.newInstance();
    }
    ...
}
```

- The type parameter `T` is erased at compile-time, the VM cannot use it at run-time

Caveats with Generics

- Instantiating a type variable, solution #1
  - Pass the class of `T` as parameter

```
public class InstantiatingTypeParameterSolution1<T> {
    public static void main(final String[] args) {
        ...
    }
    public T getInstanceOfT(final Class<T> classOfT) {
        return classOfT.newInstance();
    }
    ...
}
```

Caveats with Generics

- Instantiating a type variable, solution #2
  - Pass a factory of `T` as parameter

```
interface Factory<T> {
    T getInstance();
}
class Something {
    public static class FactoryOfSomething implements Factory<Something> {
        public Something getInstance() {
            return new Something();
        }
    }
}
public class InstantiatingTypeParameterSolution2<T> {
    public static void main(final String[] args) {
        ...
    }
    public T getInstanceOfT(final Factory<T> factory) {
        return factory.getInstance();
    }
    ...
}
```

Caveats with Generics

- Instantiating a type variable, solution #3
  - Prevent type erasure by specialising an interesting class

```
public class InstantiatingTypeParameterSolution3 extends GenericClass<String> {
    public static void main(final String[] args) {
        final InstantiatingTypeParameterSolution3 i =
            new InstantiatingTypeParameterSolution3();
        i.foo();
    }
    public void foo() {
        final Object s = this.getInstanceOfT();
        System.out.println(s.getClass());
    }
}
```

Caveats with Generics

Type argument and subclassing

- Instantiating a type variable, solution #3
  - Prevent type erasure by specialising an interesting class

```
public class InstantiatingTypeParameterSolution3 extends GenericClass<String> {
    public static void main(final String[] args) {
        final InstantiatingTypeParameterSolution3 i =
            new InstantiatingTypeParameterSolution3();
        i.foo();
    }
    public void foo() {
        final Object s = this.getInstanceOfT();
        System.out.println(s.getClass());
    }
}
```

Caveats with Generics

- Instantiating a type variable, solution #3
  - – Prevent type erasure by specialising an interesting class

```
import java.lang.reflect.ParameterizedType;

abstract class GenericClass<T> {
    public T getInstanceOfT() {
        final ParameterizedType pt =
            (ParameterizedType) this.getClass().getGenericSuperclass();
        final String parameterClassName =
            pt.getActualTypeArguments()[0].toString().split("\\s")[1];
        T parameter = (T) Class.forName(parameterClassName).newInstance();
        return parameter;
    }
}
```

Caveats with

The superclass is generic, the subclass specialises it

- Instantiating a type variable, solution #3
  - Prevent type erasure by specialising an interesting class

```java
import java.lang.reflect.ParameterizedType;

abstract class GenericClass<T> {
    public T getInstanceOfT() {
        final ParameterizedType pt =
            (ParameterizedType) this.getClass().getGenericSuperclass();
        final String parameterClassName =
            pt.getActualTypeArguments()[0].toString().split("\\s")[1];
        T parameter = (T) Class.forName(parameterClassName).newInstance();
        return parameter;
    }
}
```

Caveats with Generics

- Implicit generic methods
  - As with explicit generic methods, use `Object` in the generated bytecodes

```java
public final class Example4 {
    public static void main(final String[] args) {
        System.out.println(Util4.<String> compare("a", "b"));
        // The following line, as expected, produces a type mismatch error
        // System.out.println(Util.<String> compare(new String(""), new Long(1)));
        System.out.println(Util4.compare(new String(""), new Long(1)));
    }
}
final class Util4 {
    public static <T> boolean compare(final T t1, final T t2) {
        return t1.equals(t2);
    }
}
```

Caveats with Generics

- Implicit generic methods
  - As with explicit generic methods, use `Object` in the generated bytecodes

```
// Method descriptor #15 ([Ljava/lang/String;)V
// Stack: 7, Locals: 1
public static void main(java.lang.String[] args);
   …
   14  invokevirtual ca.polymtl.ptidej.generics.java.Util44.compare(java.lang.Object, java.lang.Object) : boolean [29]
   …
   47  invokevirtual ca.polymtl.ptidej.generics.java.Util44.compare(java.lang.Object, java.lang.Object) : boolean [29]
   …
```

to ensure backward-compatibility with pre-generic Java code

Caveats with Generics

- Multiple bounds

“A type variable with multiple bounds is a subtype of all the types listed in the bound. If one of the bounds is a class, it must be specified first.”

—The Java Tutorials, Oracle

Caveats with Generics

- Multiple bounds

```
class Example8A {
}
interface Example8B {
}
interface Example8C {
}
class Example8D<T extends Example8A & Example8B & Example8C> {
}
class Example8Test1 extends Example8A implements Example8B, Example8C {
}
class Example8Test2 extends Example8A {
}
public class Example8 {
    public static void main(final String[] args) {
        final Example8D<Example8Test1> d1 = new Example8D<Example8Test1>();
        final Example8D<Example8Test2> d2 = new Example8D<Example8Test2>();
    }
}
```

Bound mismatch: The type `Test2` is not a valid substitute for the bounded parameter `<T extends ...>`

- Multiple bounds

```
class Example8A {
}
interface Example8B {
}
interface Example8C {
}
class Example8D<T extends Example8A & Example8B & Example8C> {
}
class Example8Test1 extends Example8A implements Example8B, Example8C {
}
class Example8Test2 extends Example8A {
}
public class Example8 {
    public static void main(final String[] args) {
        final Example8D<Example8Test1> d1 = new Example8D<Example8Test1>();
        final Example8D<Example8Test2> d2 = new Example8D<Example8Test2>();
    }
}
```

Caveats with Generics

- Upper- and lower-bounded wildcards
  - Type parameters can be constrained to be
    - Any subtype of a type, `extends`
    - Any supertype of a type, `super`
  - Useful with collections of items

```
import java.util.List;

public interface ISort<E extends Comparable<E>> {
    public List<E> sort(final List<E> aList);
}
```

Caveats with Generics

- PECS
  - Collections that produce `extends`
  - Collections that consume `super`

Always from the point of view of the collection

http://stackoverflow.com/questions/2723397/java-generics-what-is-pecs

Caveats with Generics

- PECS
  - Collections that produce `extends`
    - They produce elements of some types
    - These types must be “topped” to tell the client that it can safely expect to receive `Somthing`
    - Any item from the collection is a `Somthing` (in the sense of Liskov’s substitution)

```
Collection<? extends Something>
```

Caveats with Generics

- PECS
  - Collections that consume `super`
    - They consume elements of some types
    - These types must be “bottomed” to tell the client that it can safely put `Something`
    - Any item in the collection is “at most” `Something` (in the sense of Liskov’s substitution)

```
Collection<? super Something>
```

Caveats with Generics

- PECS
  - Collections that produce and consume must just use one type parameter
    - Not legal to combine `extends` and `super`

```
Collection<Something>
```

Caveats with Generics

- Ambiguity between parameterised types

```java
public class Example9 {
    public static String f(List<String> list) {
        System.out.println("strings");
        return null;
    }
    public static Integer f(List<Integer> list) {
        System.out.println("numbers");
        return null;
    }
    public static void main(String[] args) {
        f(Arrays.asList("asdf"));
        f(Arrays.asList(123));
    }
}
```

http://stackoverflow.com/questions/2723397/java-generics-what-is-pecs

Legality depends on compiler

- Eclipse 3.5 says **yes**
- Eclipse 3.6 says **no**
- Intellij 9 says **yes**
- Sun javac 1.6.0_20 says **yes**
- GCJ 4.4.3 says **yes**
- GWT compiler says **yes**
- Crowd says **no**

- Ambiguity between parameterised types

```
public class Example9 {
    public static String f(List<String> list) {
        System.out.println("strings");
        return null;
    }
    public static Integer f(List<Integer> list) {
        System.out.println("numbers");
        return null;
    }
    public static void main(String[] args) {
        f(Arrays.asList("asdf"));
        f(Arrays.asList(123));
    }
}
```

http://stackoverflow.com/questions/2723397/java-generics-what-is-pecs

Outline

- History
- Problem
- Special Case
- General Definitions
- Generics Definitions
  - Parametric Polymorphism
  - Other Bounded Parametric Polymorphisms
- When to use generics
- How to use generics
- Caveats with generics
- **Reflecting on generics**
- Conclusion
- Few references

Reflecting on Generics

- Java generics use type erasure
  - (Most) Type parameters / arguments are erased at compile-time and exist at run-time only as annotations
  - Ensure backward-compatibility with pre-generic Java code
  - Limit access to type parameters / arguments using reflection

Caveats with Generics

- Type-safe use of `getClass()`

```
class Example11A {
}
public class Example11 {
    public static void main(final String[] args) {
        final Example11A anA1 = new Example11A();
        final Class<Example11A> anA1Class =
            (Class<Example11A>) anA1.getClass();
        System.out.println(anA1Class);

        final Example11A anA2 = new Example11A();
        final Class<? extends Example11A> anA2Class = anA2.getClass();
        System.out.println(anA2Class);
    }
}
```

http://stackoverflow.com/questions/11060491/is-there-a-clean-way-to-assign-the-class-of-a-generic-type-to-a-variable

Type safety: Unchecked cast from `Class<capture#1-of ? extends Example11A>` to `Class<Example11A>`

```
class Example11A {
}
public class Example11 {
    public static void main(final String[] args) {
        final Example11A anA1 = new Example11A();
        final Class<Example11A> anA1Class =
            (Class<Example11A>) anA1.getClass();
        System.out.println(anA1Class);

        final Example11A anA2 = new Example11A();
        final Class<? extends Example11A> anA2Class = anA2.getClass();
        System.out.println(anA2Class);
    }
}
```

http://stackoverflow.com/questions/11060491/is-there-a-clean-way-to-assign-the-class-of-a-generic-type-to-a-variable

Type safety: Unchecked cast from `Class<capture#1-of ? extends Example11A>` to `Class<Example11A>`

```
class Example11A {
}
public class Example11 {
    public static void main(final String[] args) {
        final Example11A anA1 = new Example11A();
        final Class<Example11A> anA1Class =
            (Class<Example11A>) anA1.getClass();
        System.out.println(anA1Class);

        final Example11A anA2 = new Example11A();
        final Class<? extends Example11A> anA2Class = anA2.getClass();
        System.out.println(anA2Class);
    }
}
```

No warning

http://stackoverflow.com/questions/11060491/is-there-a-clean-way-to-assign-the-class-of-a-generic-ty

Caveats with Generics

- Type-safe use of `getClass()`

```
class MyList extends ArrayList<Integer> {
}
public class Example11 {
    public static void main(final String[] args) {
        final List<Integer> list1 = new ArrayList<Integer>();
        final Class<List<Integer>> list1Class =
            (Class<List<Integer>>) list1.getClass();
        System.out.println(list1Class);

        final MyList list2 = new MyList();
        Class<? extends List<? extends Integer>> list2Class = list2.getClass();
        System.out.println(list2Class);
    }
}
```

http://stackoverflow.com/questions/11060491/is-there-a-clean-way-to-assign-the-class-of-a-generic-type-to-a-variable

Type safety: Unchecked cast from `Class<capture#4-of ? extends List>` to `Class<List<Integer>>`

- Type-safe use of `getClass()`

```
class MyList extends ArrayList<Integer> {
}
public class Example11 {
    public static void main(final String[] args) {
        final List<Integer> list1 = new ArrayList<Integer>();
        final Class<List<Integer>> list1Class =
            (Class<List<Integer>>) list1.getClass();
        System.out.println(list1Class);

        final MyList list2 = new MyList();
        Class<? extends List<? extends Integer>> list2Class = list2.getClass();
        System.out.println(list2Class);
    }
}
```

http://stackoverflow.com/questions/11060491/is-there-a-clean-way-to-assign-the-class-of-a-generic-type-to-a-variable

Type safety: Unchecked cast from `Class<capture#4-of ? extends List>` to `Class<List<Integer>>`

- Type-safe use of `getClass()`

```
class MyList extends ArrayList<Integer> {
}
public class Example11 {
    public static void main(final String[] args) {
        final List<Integer> list1 = new ArrayList<Integer>();
        final Class<List<Integer>> list1Class =
            (Class<List<Integer>>) list1.getClass();
        System.out.println(list1Class);

        final MyList list2 = new MyList();
        Class<? extends List<? extends Integer>> list2Class = list2.getClass();
        System.out.println(list2Class);
    }
}
```

No warning

http://stackoverflow.com/questions/11060491/is-there-a-clean-way-to-assign-the-class-of-a-generic-ty

Caveats with Generics

- Use of `newInstance()`

```
class Example10A {
}
public class Example10 {
    public static void main(final String[] args) {
        final Class<Example10A> clazz1 = Example10A.class;
        final Example10A anA1 = clazz1.newInstance();
        System.out.println(anA1);

        final Class<?> clazz2 = Class.forName(
            "ca.polymtl.ptidej.generics.java.Example9A");
        final Example10A anA2 = (Example10A) clazz2.newInstance();
        System.out.println(anA2);
    }
}
```

http://stackoverflow.com/questions/2592642/type-safety-unchecked-cast-from-object

Caveats with Generics

- Obtaining the type of a type parameter
  - Due to type erasure
    - Type parameters are kept as annotations
    - Type arguments disappear

  Except for anonymous/local classes!

http://stackoverflow.com/questions/1901164/get-type-of-a-generic-parameter-in-java-with-reflection

Caveats with Generics

- Obtaining the type of a type parameter

```java
public final class Voodoo0 extends TestCase {
    public static void chill(final List<?> aListWithSomeType) {
        CommonTest.assertNotEqualAsExpected(
            aListWithSomeType,
            SpiderManVoodoo0.class);
    }
    public static void main(String... args) {
        Voodoo0.chill(new ArrayList<SpiderManVoodoo0>());
    }
    public void test() {
        Voodoo0.main(new String[0]);
    }
}
class SpiderManVoodoo0 {
}
```

Caveats with Generics

- Obtaining the type of a type parameter

```
public final class Voodoo0 extends TestCase {
    public static void chill(final List<?> aListWithSomeType) {
        CommonTest.assertNotEqualAsExpected(
            aListWithSomeType,
            SpiderManVoodoo0.class);
    }
    public static void main(String... args) {
        Voodoo0.chill(new ArrayList<SpiderManVoodoo0>());
    }
    public void test() {
        Voodoo0.main(new String[0]);
    }
}
class SpiderManVoodoo0 {
}
```

✗

Caveats with Generics

- Obtaining the type of a type parameter

```java
public final class Voodoo1 extends TestCase {
    public static void chill(final List<?> aListWithSomeType) {
        CommonTest.assertNotEqualAsExpected(
            aListWithSomeType,
            SpiderManVoodoo1.class);
    }
    public static void main(String... args) {
        Voodoo1.chill(new ArrayList<SpiderManVoodoo1>() {});
    }
    public void test() {
        Voodoo1.main(new String[0]);
    }
}
class SpiderManVoodoo1 {
}
```

Caveats with Generics

- Obtaining the type of a type parameter

```java
public final class Voodoo1 extends TestCase {
    public static void chill(final List<?> aListWithSomeType) {
        CommonTest.assertNotEqualAsExpected(
            aListWithSomeType,
            SpiderManVoodoo1.class);
    }
    public static void main(String... args) {
        Voodoo1.chill(new ArrayList<SpiderManVoodoo1>() {});
    }
    public void test() {
        Voodoo1.main(new String[0]);
    }
}
class SpiderManVoodoo1 {
}
```

✓

Caveats with Generics

- Obtaining the type of a type parameter

```
public final class Voodoo1 extends TestCase {
    public static void chill(final List<?> aListWithSomeType) {
        CommonTest.assertNotEqualAsExpected(
            aListWithSomeType,
            SpiderManVoodoo1.class);
    }
    public static void main(String... args) {
        Voodoo1.chill(new ArrayList<SpiderManVoodoo1>() {});
    }
    public void test() {
        Voodoo1.main(new String[0]);
    }
}
class SpiderManVoodoo1 {
}
```

✓

Caveats with

Anonymous/local class stores types information

- Obtaining the type of a type parameter

```
public final class Voodoo1 extends TestCase {
    public static void chill(final List<?> aListWithSomeType) {
        CommonTest.assertNotEqualAsExpected(
            aListWithSomeType,
            SpiderManVoodoo1.class);
    }
    public static void main(String... args) {
        Voodoo1.chill(new ArrayList<SpiderManVoodoo1>() {});
    }
    public void test() {
        Voodoo1.main(new String[0]);
    }
}
class SpiderManVoodoo1 {
}
```

✓

Caveats with Generics

- Obtaining the type of a type parameter

```
// Compiled from Voodoo1.java (version 1.7 : 51.0, super bit)
// Signature: Ljava/util/ArrayList<Lca/polymtl/ptidej/generics/java/erasure/SpiderManVoodoo1;>;
class ca.polymtl.ptidej.generics.java.erasure.Voodoo1$1 extends java.util.ArrayList {

  ...

  // Method descriptor #11 ()V
  // Stack: 1, Locals: 1
  Voodoo1$1();
    0  aload_0 [this]
    1  invokespecial java.util.ArrayList() [13]
    4  return
      Line numbers:
        [pc: 0, line: 38]
        [pc: 4, line: 1]
      Local variable table:
        [pc: 0, pc: 5] local: this index: 0 type: new ....java.erasure.Voodoo1(){}

  ...
}
```

Outline

- History
- Problem
- Special Case
- General Definitions
- Generics Definitions
  - Parametric Polymorphism
  - Other Bounded Parametric Polymorphisms
- When to use generics
- How to use generics
- Caveats with generics
- Reflecting on generics
- **Conclusion**
- Few references

Conclusion

- Java generics

  "Implement generic algorithms that work on a collection of different types"

  —The Java Tutorials, Oracle

Conclusion

- Scenario 1: you want to enforce type safety for containers and remove the need for typecasts when using these containers
- Scenario 2: you want to build generic algorithms that work on several types of (possible unrelated) things

Conclusion

- Easy to use in simple cases
- Several caveats, though

- Can be very tricky is corner cases
  - Use them sparingly and purposefully

Outline

- History
- Problem
- Special Case
- General Definitions
- Generics Definitions
  - Parametric Polymorphism
  - Other Bounded Parametric Polymorphisms
- When to use generics
- How to use generics
- Caveats with generics
- Reflecting on generics
- Conclusion
- **Few references**

Outline

- In no particular order
  - http://en.wikipedia.org/wiki/Generics_in_Java
  - http://www.angelikalanger.com/GenericsFAQ/FAQSections/TechnicalDetails.html#FAQ502
  - http://www.uio.no/studier/emner/matnat/ifi/INF3110/h05/lysark/Types.pdf
  - http://www.slideshare.net/SFilipp/java-puzzle-167104
  - http://www.jquantlib.org/index.php/Using_TypeTokens_to_retrieve_generic_parameters#Anonymous_classes
  - http://www.clear.rice.edu/comp310/JavaResources/ generics/
  - http://gafter.blogspot.kr/2006/12/super-type-tokens.html