BOSTON UNIVERSITY
METROPOLITAN COLLEGE

Generics

By Warren Mansur, edits by Eric Braude

Generics Agenda

1. **Introduction**
2. Generic Methods
3. Generic Classes
4. Generic Bounded Types
5. Wildcard
6. Type Erasure

Call-by-value vs. Template

- … f(int i) { ← called by value (of i)
- Generics parametrizes a *class*
  - (not instances of a class)
  - but a class doesn’t have a “value”
- The best we can do is utilize a *template*
  - Generic class

Generics Agenda

1. Introduction
2. **Generic Methods**
3. Generic Classes
4. Generic Bounded Types
5. Wildcard
6. Type Erasure

```java
public class NotGenericMethods {
  static Integer[] iarr = {2, 5, 8};
  static Double[] darr = {28.67, 5.05, 8.3};
  static String[] sarr = {"Twelve", "Angry", "Men"};
  public static void main(String[] args) {
      showIntArray(iarr);
      showDoubleArray(darr);
      showStringArray(sarr);
  }
  public static void showIntArray(Integer[] a) {
       for (Integer elem: a)
           System.out.print(elem + " ");
       System.out.println();
  }
  public static void showDoubleArray(Double[] a) {
       for (Double elem: a)
           System.out.print(elem + " ");
       System.out.println();
  }
  public static void showStringArray(String[] a) {
       for (String elem: a)
           System.out.print(elem + " ");
       System.out.println();
  }
}
```

Not Generic Methods

Output

```
2 5 8
28.67 5.05 8.3
Twelve Angry Men
```

… but there is commonality we’re not capturing.

Generic Method Alternative

```java
public class GenericMethods {

  static Integer[] iArr = {2, 5, 8};
  static Double[] dArr = {28.67, 5.05, 8.3};
  static String[] sArr = {"Twelve", "Angry", "Men"};

  public static void main(String[] args) {
    showArray(iArr);
    showArray(dArr);
    showArray(sArr);
  }

  public static <T> void showArray(T[] a) {
    for (T elem: a)
      System.out.print(elem + " ");
    System.out.println();
  }
}
```

Output

```
2 5 8
28.67 5.05 8.3
Twelve Angry Men
```

Generics Agenda

1. Introduction

2. Generic Methods

3. Generic Classes

Non-Generic Container Class

```java
public class NonGenericContainer {

  private Object o;
  private String description;

  public NonGenericContainer
       (Object o, String description) {
    this.o = o;
    this.description = description;
  }

  public Object getObject() {
    return o;
  }

  public static void main(String[] args) {…
```

Output

```
wrapper1 object = 3? ==> true
wrapper2 object = nine? ==> true
wrapper3 object = false? ==> true
```

Non-Generic Container Class

```java
public class NonGenericContainer {
  private Object o;
  private String description;
  public NonGenericContainer
       (Object o, String description) {
    this.o = o;
    this.description = description;
  }
  public Object getObject() {
    return o;
  }
  public static void main(String[] args) {
    NonGenericContainer wrapper1 = new NonGenericContainer(3, "three");
    NonGenericContainer wrapper2 = new NonGenericContainer("nine", "nine");
    NonGenericContainer wrapper3 = new NonGenericContainer(false, "false");
    System.out.println("wrapper1 object = 3? ==> " +
       ((int)wrapper1.getObject() == 3)); //must be cast
    System.out.println("wrapper2 object = nine? ==> " +
       (((String)wrapper2.getObject()).equals("nine"))); //must be cast
    System.out.println("wrapper3 object = false? ==> " +
       ((Boolean)wrapper3.getObject() == false)); //mus
  }
}
```

Output

```
wrapper1 object = 3? ==> true
wrapper2 object = nine? ==> true
wrapper3 object = false? ==> true
```

```java
public class NonGenericIntContainer {
  private Integer o;
  private String description;
  public NonGenericIntContainer(Integer o, String description) {
    this.o = o;
    this.description = description;
  }
  public Integer getObject() {
    return o;
  }
}

public class NonGenericStringContainer {
  private String o;
  private String description;
  public NonGenericStringContainer(String o, String description) {
    this.o = o;
    this.description = description;
  }
  public String getObject() {
    return o;
  }
}
```

One Class Per Type

One Class Per Type

```java
public class NonGenericBooleanContainer {
  private Boolean o;
  private String description;
  public NonGenericBooleanContainer(Boolean o, String description) {
    this.o = o;
    this.description = description;
  }
  public Boolean getObject() {
    return o;
  }
}

public class NonGenericContainerTest {
  public static void main(String[] args) {
    NonGenericIntContainer wrapper1 = new NonGenericIntContainer(3, "three");
    NonGenericStringContainer wrapper2 = new NonGenericStringContainer("nine", "nine");
    NonGenericBooleanContainer wrapper3 = new NonGenericBooleanContainer(false, "false");
    System.out.println("wrapper1 object = 3? ==> " +
        (wrapper1.getObject() == 3)); //avoids casting
    System.out.println("wrapper2 object = nine? ==> " +
        (wrapper2.getObject().equals("nine"))); //avoids casting
    System.out.println("wrapper3 object = false? ==> " +
        (wrapper3.getObject() == false)); //avoids casting
  }
}
```

Output

```
wrapper1 object = 3? ==> true
wrapper2 object = nine? ==> true
wrapper3 object = false? ==> true
```

Solution: Generic Class

```
public class GenericContainer<T> {

  private T o;

  private String description;

  public GenericContainer(T o, String description)
    this.o = o;
    this.description = description;
  }

  public T getObject() {
    return o;
  }
…
}
```

template

Solution: Generic Class

```java
public class GenericContainer<T> {
  private T o;
  private String description;
  public GenericContainer(T o, String description) {
    this.o = o;
    this.description = description;
  }
  public T getObject() {
    return o;
  }
  public static void main(String[] args) {
    GenericContainer<Integer> wrapper1 = new GenericContainer<Integer>(3, "three");
    GenericContainer<String> wrapper2 = new GenericContainer<String>("nine", "nine");
    GenericContainer<Boolean> wrapper3 = new GenericContainer<Boolean>(false, "false");
    System.out.println("wrapper1 object = 3? ==> " +
        (wrapper1.getObject() == 3)); //avoids cast
    System.out.println("wrapper2 object = nine? ==> " +
        (wrapper2.getObject().equals("nine"))); //avoids cast
    System.out.println("wrapper3 object = false? ==> " +
        (wrapper3.getObject() == false)); //avoids cast
  }
}
```

Output

```
wrapper1 object = 3? ==> true
wrapper2 object = nine? ==> true
wrapper3 object = false? ==> true
```

A Bit of History – C++ Templates

```
template <class SomeType>
SomeType GetMin (myType a, myType b) {
 return (a<b?a:b);
}

...
int i,j;
GetMin (i,j);
```

Generics Agenda

1. Introduction
2. Generic Methods
3. Generic Classes
4. **Generic Bounded Types**
5. Wildcard
6. Type Erasure

Recall Inheritance

```java
public class Student {
  public void Identify() {
    System.out.println("I am a student.");
  }
}

public class UndergradStudent extends Student {
  public void Identify() {
    System.out.println("I am an undergrad.");
  }
  public void IdentifyMinor() {
    System.out.println("My minor is basket weaving.");
  }
}

public class GradStudent extends Student {
  public void Identify() {
    System.out.println("I am a graduate.");
  }
  public void IdentifyThesis() {
    System.out.println("My thesis is about chess boxing.");
  }
}
```

Generic Bounded Types

```java
public class StudentContainer<S extends Student> {
  private S student;
  public StudentContainer(S student) {
    this.student = student;
  }
  public S getStudent() {
    return student;
  }
  public static void main(String[] args) {
    StudentContainer<UndergradStudent> UndergradContainer =
        new StudentContainer<UndergradStudent>(new UndergradStudent());
    StudentContainer<GradStudent> GradContainer =
        new StudentContainer<GradStudent>(new GradStudent());

    //No casting required.
    UndergradContainer.getStudent().Identify();
    UndergradContainer.getStudent().IdentifyMinor();
    GradContainer.getStudent().Identify();
    GradContainer.getStudent().IdentifyThesis();

    //Wouldn't compile: Bound mismatch: The type Integer
    // is not a valid substitute for the bounded parameter ...
    // StudentContainer<Integer> Container =
    //   new StudentContainer<Integer>(new Integer(1));
  }
}
```

Output

```
I am an undergrad.
My minor is basket weaving.
I am a graduate.
My thesis is about chess boxing.
```

Contraindication: Standard Inheritance

- ➲ Do not use a generic when the class only need operate on *any* one of the type.
- ➲ Use a generic when the class using the type must operate on *exactly* that type.

Contraindication: Standard Inheritance

```java
public class UndergradStudentAlt extends Student {
  public void Identify() {
    System.out.println("I am an undergrad.");
  }
}
public class GradStudentAlt extends Student {
  public void Identify() {
    System.out.println("I am a graduate.");
  }
}
public class StudentContainerAlt<S extends Student> {
  public S student;
  public StudentContainerAlt(S student) {
    this.student = student;
  }
  public static void main(String[] args) {
    StudentContainerAlt<UndergradStudentAlt> UndergradContainer =
        new StudentContainerAlt<UndergradStudentAlt>(new UndergradStudentAlt());
    StudentContainerAlt<GradStudentAlt> GradContainer =
        new StudentContainerAlt<GradStudentAlt>(new GradStudentAlt());

    //No advantage for using generic in this scenario.
    UndergradContainer.student.Identify();
    GradContainer.student.Identify();
  }
}
```

Generics Agenda

1. Introduction
2. Generic Methods
3. Generic Classes
4. Generic Bounded Types
5. ⇨ **Wildcard**
6. Type Erasure

Generic Wildcard

```java
public class WildcardExample {

  public static void main(String[] args) {
    ArrayList<Integer> IntList =
      new ArrayList<Integer>(Arrays.asList(5,2));
    ArrayList<String> StringList =
      new ArrayList<String>(Arrays.asList("one"));
    ArrayList<Boolean> BoolList =
      new ArrayList<Boolean>(Arrays.asList(true, false));
    printCollection(IntList);
    printCollection(StringList);
    printCollection(BoolList);
  }

  public static void printCollection(ArrayList<?> collection) {
    for (Object o : collection) {
      System.out.println(o);
    }
  }
}
```

Output

```
5
2
one
true
false
```

Bounded Wildcard

```java
public class WildcardExtendsExample {

  public static void main(String[] args) {
    ArrayList<Integer> IntList =
        new ArrayList<Integer>(Arrays.asList(5,2));
    ArrayList<Double> DoubleList =
        new ArrayList<Double>(Arrays.asList(3.5));
    ArrayList<Long> LongList =
        new ArrayList<Long>(Arrays.asList(100L, 1000L));
    addNumberCollection(IntList);
    addNumberCollection(DoubleList);
    addNumberCollection(LongList);

    //Will not compile: The method addNumbercollection(ArrayList<String>) is undefined...
    //addNumbercollection(new ArrayList<String>(Arrays.asList("string")));
  }

  public static void addNumberCollection(ArrayList<? extends Number> collection) {
    Double result = 0.0;
    for (Number n : collection) {
      result += n.doubleValue();
    }
    System.out.println("Result = " + result);
  }
}
```

Output

```
Result = 7.0
Result = 3.5
Result = 1100.0
```

Generics Agenda

1. Introduction

2. Generic Methods

3. Generic Classes

4. Generic Bounded Types

5. Wildcard

⇨ **6. Type Erasure**

Backward Compatibility

- Generics did not exist until Java Version 5 (previously known as 1.5).
- It was essential the new feature did not break backward compatibility with existing JVMs, so *type erasure* was introduced along with generics.
- The Java compiler checks the correctness of code that uses generics at compile time, then erases their use when generating bytecode.

Unbounded Class Type Erasure

```java
//Before the class is compiled.
public class ClassTypeErasure<T> {
  private T member;
  public ClassTypeErasure(T member) {
    this.member = member;
  }
  public T getMember() {
    return member;
  }
}

//After compilation, all references to T are replaced with Object.
public class ClassTypeErasure {
  private Object member;
  public ClassTypeErasure(Object member) {
    this.member = member;
  }
  public Object getMember() {
    return member;
  }
}
```

Bounded Class Type Erasure

```
//Before the class is compiled.
public class BoundedClassTypeErasure<T extends Number> {
  private T member;
  public BoundedClassTypeErasure(T member) {
    this.member = member;
  }
  public T getMember() {
    return member;
  }
}

//After compilation, all references to T are replaced with
Number.
public class BoundedClassTypeErasure {
  private Number member;
  public BoundedClassTypeErasure(Number member) {
    this.member = member;
  }
  public Number getMember() {
    return member;
  }
}
```

Method Type Erasure

```java
//Before compile
public class MethodTypeErasure {

  //T is checked at compiletime.
  public static <T> void PrintIt(T param) {
    System.out.println(param);
  }

  //T is checked at compiletime.
  public static <T extends Number> void PrintNumberPlusOne(T param) {
    System.out.println(param.doubleValue() + 1);
  }
}

//After compile
public class MethodTypeErasure {

  //T is replaced with object after compile.
  public static void PrintIt(Object param) {
    System.out.println(param);
  }

  //T is replaced with Number after compile.
  public static void PrintNumberPlusOne(Number param) {
    System.out.println(param.doubleValue() + 1);
  }
}
```

Generic Limitation: Still One Class

➲ A generic class is does not exist at runtime; only the base class exists.

➲ All instances of a generic class are instances of the same base class.

```
ArrayList<String> list1 = new ArrayList<String>();
ArrayList<Number> list2 = new ArrayList<Number>();

//list1 instanceof ArrayList is true
//list2 instanceof ArrayList is true
//list1 instanceof ArrayList<String> does not compile.
//list2 instanceof ArrayList<Number> does not compile.

//At runtime, there is no ArrayList<String> class, only an ArrayList class.
```

Generic Limitation: *new* No-No

- A generic class does not exist at runtime, so it’s not possible to create a new instance of it.

```
public class ContainerWithNew<T> {

  public T createItem() {
    //Cannot do this. Compiler reports "Cannot instantiate the type T".
    return new T();
  }

  public T[] createItemArray() {
    //Cannot do this. Compiler reports "Cannot create a generic array of T".
    return new T[5];
  }
}
```

Generics Summary

- ➲Type checks occur at compile time rather than runtime (huge)!
- ➲The need for verbose casting is eliminated.
- ➲Programmers may implement more general algorithms rather than repeating the same code over and over for each type.
- ➲It is easier to learn a single class than a suite of classes that accomplish the same thing.
- ➲Generics can be defined at both the class and method level.

Type Erasure Summary

- ➲Type erasure ensures backward compatibility by checking correctness at compile time then erasing the generics.
- ➲Therefore only one class exists at runtime.
- ➲It’s not possible to use instanceof on generic classes, nor it is possible to create new objects of the generic directly.

When to Use Generic Summary

- ➲ Use a generic when those who use the class or method must know the exact type being used at compiletime.
- ➲ Use a generic when the class or method can operate on many types, but it’s not necessary for the class or method to know exactly which of these types its operating on.
- ➲ Use a generic when logically you are creating a class template or method template that works across many types (to avoid redefining it over and over again for each type).
- ➲ *Do not* define a generic class or method when standard inheritance will do; that is, when those who use the class do not need to tie it down to a specific type at compiletime.