Java Generics

Object Oriented Programming (236703)
Winter 2015-6
The Dark Ages: Before Java 5

- Java supported only inclusion polymorphism
  - Relying on a common superclass
- Every class is a subclass of java.lang.Object
  - So there's always at least one common superclass
- Collections were actually collections of Objects
  - You can put anything into a collection
  - When you extract something, its static type is Object
  - As safe as any dynamically typed languages
    - But Java is statically typed, and should be safer!
Generics – Before

List list = new ArrayList();
list.add(new Integer(0));
Integer x = (Integer) list.get(0); // downcast required
list.add("abc");
Integer y = (Integer) list.get(1); // Runtime exception

▲ No type safety – collections can hold a mixture of unrelated objects
▲ No internal documentation (i.e., source describing itself) – the declaration does not indicate the use
▲ Still legal Java code
   ▲ But yields a compilation warning (explained later)
Java 5 and Beyond

- Parametric polymorphism introduced in Java 5 (2004)
  - Most significant enhancement since Java's birth
- May resemble C++ templates, but:
  - Implemented differently – “Compile once and for all”
    - No executable blowup
  - Type constraints are explicit – better error messages
  - Less power (e.g., cannot inherit from a type parameter)
  - Parameters can only be classes or interfaces (not primitive types)
Generics – After

List<Integer> list = new ArrayList<Integer>();
// From Java 7:   = new ArrayList<>();
list.add(new Integer(0));
Integer x = list.get(0); // cast not needed
list.add("abc"); // compiler error: expected integer

- Type safe – the collection in use can only hold integers
  - Errors are detected at compile time
- Internally documented – the type indicates it stores integers
Methods Can Be Generic Too

```java
public static <T> List<T> dup(T t, int n) {
    List<T> result = new ArrayList<>();
    for (int i = 0; i < n; ++i)
        result.add(t);
    return result;
}

// Implicit instantiation (common):
List<String> l = dup("abc", 2);

// Explicit instantiation (uncommon):
List<String> m = DupClass.<String>dup("abc", 2);
```
public class Cell<T> {
    private T value;
    public T get() { return value; }
    public void set(T t) { value = t; }
    // T is at least an Object, so it supports toString()
    public String toString() { return value.toString(); }
}
Using Cell<T>

public static void main(String[] args) {
    Cell<Integer> ci = new Cell<>();
    ci.set(new Integer(5));
    System.out.println(ci.get());

    int n = ci.get(); // auto-unboxing
    ci.set(n * n); // auto-boxing

    Cell<Number> cn = new Cell<>();
    cn.set(ci.get());
}

Type Parameters with Upper Bounds

```java
public class NumberCell<T extends Number> {
    // value, get, set, toString()...
    
    // T is at least a Number, 
    // so it supports intValue()
    public int sum(int x) {
        return value.intValue() + x;
    }
}
```

- The `extend` keyword specifies an upper bound for `T`
  - Can be used with both classes and interfaces
- Can have multiple bounds (one class & many interfaces):
  - `T extends MyClass & MyInterface`
Bounds → Constraints

- Inside a generic class: methods can safely assume the type argument is at least the upper bound
  - No need for explicit type checks
  - Errors are detected when the generic class is compiled
    - (comparing to an unbound class that is intended to be used with specific types, and uses casts)

- Users of the class: no surprises
  - If the type argument doesn’t fit, the code will not compile

- Clear separation between internal and external errors
  - Internal: presented to class author when creating the class
  - External: presented to class user when using the class
**Cell<Integer> Is Not A Cell<Object>**

static void assign(Cell<Object> co, Object o) {
    co.set(o);
}

void main(String[] args) {
    Cell<Integer> ci = new Cell<Integer>();
    assign(ci, new Integer(10));
    // what would happen if the above were legal?
    assign(ci, "abc");
    Integer n = ci.get();
    System.out.println(n.intValue());
}

Integer is an Object ⇒ Cell<Integer> is a Cell<Object>!
Generics vs. Arrays

- Reminder: Java arrays are covariant
  - `Object[] arr = new String[10];`
  - Type checks deferred to runtime
    - `arr[0] = new Object()` → exception

- Java generic classes are by default no-variant
  - `Cell<Object> c = new Cell<Integer>();` yields a compilation error
  - Safety over flexibility

- Not to be confused:
  - `List<String> ls = new ArrayList<String>();` is legal!
A Better Version Of Assign()

```java
static<T> void assign(Cell<T> co, T o) {
    // previously: (Cell<Object> co, Object o)
    co.set(o);
}

void main() {
    Cell<Integer> ci = new Cell<Integer>();
    assign(ci, new Integer(10)); // Now it works
}
```
The Wildcard: <?>

```java
static boolean isNull(Cell<?> c) {
    return c.get() == null;
}
static public void main(String[] args) {
    isNull(new Cell<Integer>);
}
```

- The wildcard provides variance
- Applicable to type use, not type definition!
  - Variables, parameters, return values and base classes
  - No class Cell<?> { ... }

14
Upper-bound $\rightarrow$ Covariance

```java
static boolean isNull(Cell<? extends Number> c) {
    Number n = c.get();
    c.set(new Number(5));  // ERROR!
}
```

- An upper-bound wildcard allows getting values from the variable
- Passing arguments to methods is forbidden
  - Cell<? Extends Number> can hold a Cell<Integer>, Cell<Double> and even Cell<Number>
  - One exception: null can safely be passed
public class Cell<T> {
    ...
    public void copyTo(Cell<? super T> c) {
        c.set(this.value);
    }
}

Cell<Integer> ci = new Cell<>();
Cell<Number> cn = new Cell<>();
ci.copyTo(cn);  // Integer → Number

Now, no returning – just passing in arguments

Lower bound only legal (and sensible) with wildcards
Type Erasure

- Compiling a Generic class: Cell<T>

  1. Check type correctness
  2. Perform **Type Erasure**: replace T with its upper bound (Object by default). The result is the **Raw Type**.
  3. Compile to byte code

```java
class Cell<T extends Number>
{
    T value;
    T get() { ... }
    void set(T t) { ... }
    String toString() { ... }
}

class Cell  // raw type
{
    Number value;
    Number get() { ... }
    void set(Number t) { ... }
    String toString() { ... }
}
```
Instantiation Procedure

- Compiling an instantiation (type): `Cell<String> c;`
  1. Replace the instantiated type with the raw type
  2. “Annotate” **generic types** with the type arguments
     - Fields, method arguments and return type, base class/interface
     - Not regular annotation, but available at runtime using reflection

```java
class SomeClass {
    public Cell<Integer> cell;
}
```

```java
class SomeClass {
    public Cell cell;
}
```
Ensuring Type Safety

Compiling a field access or a message send:

1. Obtain the “annotation” of the receiver or method
2. Check actual method parameters against the actual type parameters
3. Downcast return type to the actual type parameter if needed

```java
void main(String[] args) {
    SomeClass sc = ...;
    sc.cell.set(1);
    Integer i = sc.cell.get();
}
```

```java
void main(String[] args) {
    // sc.cell.set <- Integer?
    sc.cell.set(...)
    Integer i =
        (Integer)sc.cell.get();
}
```
Type Erasure And Reflection

- Past erasure, the **class** and **objects** do not contain type arguments
- Some program elements do keep the type arguments:
  - Fields, parameters, return values, base classes/interfaces
  - (as long as they are instantiated – List<String> and not List<T>)

```java
class MyClass { public List<String> myList; }

public static void main(String[] args) throws Exception {
    Field field = MyClass.class.getField("myList");
    ParameterizedType type =
        (ParameterizedType)field.getGenericType();
    Type[] argTypes = type.getActualTypeArguments();
    for (Type argType : argTypes)
        System.out.println("argClass = " + argType);
}
```

argClass = class java.lang.String
List<Integer> list = new ArrayList<>();
Object o = list;
List<Object> fake = (List<Object>) o;
    // Warning: Unchecked Conversion
fake.add("abc");
Integer i = list.get(0);
    // Runtime error: cast from String to Integer

- Unchecked conversion warning
  - Issued when assigning a raw type into an instantiated type
    - The compiler cannot verify type safety
  - Indicates a danger of type errors at run time – avoid!
- Assigning instantiated type into a raw type – ok
In Java – Virtual <3 Generic

- Reminder: in C++, a virtual function cannot be a template
  - Each instantiation creates a separate function, making it practically impossible to create the vtable
- In Java, a virtual (i.e., not final) method can be generic
  - Type erasure → a single method instance
  - Only one vtable entry is required
Type Erasure – Pros and Cons

- Benefit of Erasure:
  - Binary compatibility with older libraries: List<String> is translated to type List (raw type)
    - Legacy code using "pre-generics" types (e.g. containers) still usable

- Drawback of Erasure
  - Generic objects carry no type information
    - List<Integer> and List<String> refer to the raw List
  - No type information inside a generic class/method
    - Type variables cannot be used in new expressions
    - Overload resolution cannot rely on type argument