CSE 331
Software Design & Implementation

Hal Perkins

Winter 2012

Generics

(Slides by David Notkin and Mike Ernst)

Why we ♥ abstraction

- Hide details
  - Avoid distraction
  - Permit the details to change later
- Give a meaningful name to a concept
- Permit reuse in new contexts
  - Avoid duplication: error-prone, confusing
  - Programmers hate to repeat themselves – “lazy”

Varieties of abstraction

- Abstraction over computation: procedures
  - `int x1, y1, x2, y2;`
  - `Math.sqrt(x1*x1 + y1*y1);`
  - `Math.sqrt(x2*x2 + y2*y2);`
- Abstraction over data: ADTs (classes, interfaces)
  - `Point p1, p2;`
- Abstraction over types: polymorphism (generics)
  - `Point<Integer>, Point<Double>`
  - Applies to both computation and data

Parametric polymorphism

- Ability to write a function or type so that it handles values identically without depending on knowledge of their types
- These are generic functions or generic data types – they take a type as a parameter
  - That is, they allow for substitutability of types under some conditions
  - First introduced in ML language in 1976, although the concept has been around since (at least) LISP
  - Now part of many other languages (Haskell, Java C#, Delphi)
  - C++ templates are similar but lack various features/flexibility
- Parametric polymorphism allows you to write flexible, general code without sacrificing type safety
  - Most commonly used in Java with collections
  - Also used in reflection (seen later)

Type Parameters (Generics)

- `List<Type> name = new ArrayList<Type>();`
- Since Java 1.5, a constructor of **`java.util.ArrayList`** can (and almost always does) specify the type of elements it will contain
  - The type that is passed is called the *type parameter*

```
List<String> names = new ArrayList<String>();
names.add("Boris");
names.add("Natasha");
String spy = names.get(0); // ok element type
Point oops = (Point)names.get(1);
                // error -- need String not Point
```

- Use of the “raw type” **`ArrayList`** (with no type is passed) leads to warnings (which can be controlled by compiler options)

Programs include a group of abstractions

```
interface ListOfNumbers {
  boolean add(Number elt);
  Number get(int index);
}
interface ListOfIntegers {
  boolean add(Integer elt);
  Integer get(int index);
}
```

… and many, many more

```
interface List<E> {
  boolean add(E n);
  E get(int index);
}
```

Declares a new **variable**, called a **formal parameter**

Instantiate by passing an **Integer**:
l.add(7);
myList.add(myInt);

The type of **add** is
Integer → boolean

Declares a new **type variable**, called a **type parameter**

Instantiate by passing a type:
List<Float>
List<List<String>>
List<T>

The type of **List** is
Type → Type

Declaring and instantiating generics

```
// a parameterized (generic) class
public class name<Type> {
```

or

```
public class name<Type, Type, ..., Type> {
```

- Putting the **`Type`** in **`< >`** states that any client that constructs your object must supply one or more type parameters
  - Just like a "regular" method's parameters state that any client invoking it must supply objects of the proper type
- It is essentially a constructor for the generic class
- The rest of the class's code refers to that type by name
  - The convention is to use a 1-letter name such as **`T`** for **`Type`**, **`E`** for **`Element`**, **`N`** for **`Number`**, **`K`** for **`Key`**, **`V`** for **`Value`**, or **`M`** for **`Murder`**
- The type parameter is instantiated by the client. (e.g. **`E`** → **`String`**), essentially invoking the generic class constructor

Using type variables

- Implementation code of the generic class can perform any operation permitted by the type variable

```
interface List1<E extends Object> {
  void m(E arg) {
    arg.asInt();   // compiler error, E might not
                   // support asInt
  }
}

interface List2<E extends Number> {
  void m(E arg) {
    arg.asInt();  // OK, since Number and its
                  // subtypes support asInt
  }
}
```

Invocations by clients are restricted

```
boolean add1(Object elt);
boolean add2(Number elt);
add1(new Date());  // OK
add2(new Date());  // compile-time error

interface List1<E extends Object> {…}
interface List2<E extends Number> {…}
List1<Date>       // OK, Date is a subtype
                  // of Object
List2<Date>       // compile-time error,
                  // Date is not a subtype
                  // of Number
```

Type variables are types

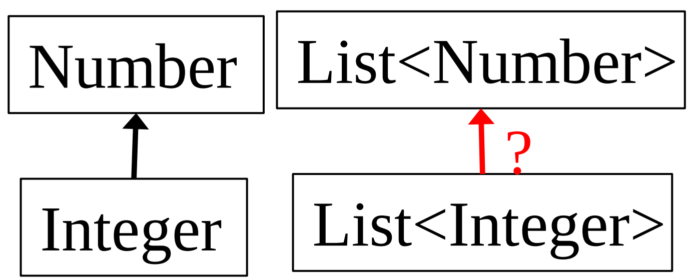
Declaration

```
class NewSet<T> implements Set<T> {
    // rep invariant:
    //    non-null, contains no duplicates
    List<T> theRep;
}
```

Use

Generics and subtyping

- **Integer** is a subtype of **Number**
- Is **List<Integer>** a subtype of **List<Number>**?
- Use subtyping rules (stronger, weaker) to find out



`List<Number>` and `List<Integer>`

```
interface List<Number> {
  boolean add(Number elt);
  Number get(int index);
}
interface List<Integer> {
  boolean add(Integer elt);
  Integer get(int index);
}
```

Java subtyping is covariant (less tolerant) with respect to generics

But covariant subtyping is restrictive

```
interface Set<E> {
  // Adds all of the elements in c to this set
  // if they're not already present (optional operation)
  void addAll(Set<E> c);
}
```

Problem 1:

```
Set<Number> s;
List<Number> l;
s.addAll(l);
```

```
interface Set<E> {
  void addAll(Collection<E> c);
}
```

Problem 2:

```
Set<Number> s;
List<Integer> l;
s.addAll(l);
```

```
interface Set<E> {
  void addAll(Collection<? extends E> c);
}
```

A “Collection of unknown”type that extends E

Using wildcards

```
class HashSet<E> implements Set<E> {
  void addAll(Collection<? extends E> c) {
    // What can this code assume about c?
    // What operations can this code invoke on c?
    ...
  }
}
```

- Wildcards are written at declarations, not uses
  - The use defines the **?** as a parameter when it the type is instantiated
- A missing **extends** clause means **extends Object**
- There is also “**? super E**”

Wildcards

- `?` indicates a wild-card type parameter, one that can be any type
  `List<?> list = new List<?>();   // anything`

- Difference between `List<?>` and `List<Object>`
  - `?` can become any particular type; `Object` is just one such type
  - `List<Object>` is restrictive; wouldn't take a `List<String>`

- Difference between `List<Foo>` and `List<? extends Foo>`
  - The latter binds to a particular `Foo` subtype and allows ONLY that
    - Ex: `List<? extends Animal>` might store only `Giraffes` but not `Zebras`
  - The former allows anything that is a subtype of `Foo` in the same list
    - Ex: `List<Animal>` could store both `Giraffes` and `Zebras`

Another example

```
public class Graph<N> implements Iterable<N> {
  private final Map<N, Set<N>> node2neighbors;
  public Graph(Set<N> nodes, Set<Tuple<N,N>> edges) {
    ...
  }
}

public interface Path<N, P extends Path<N,P>>
  extends Iterable<N>, Comparable<Path<?, ?>> {
  public Iterator<N> iterator();
}
```

Bounded type parameters

- **`<Type extends SuperType>`**
  An upper bound; accepts the given supertype or any of its subtypes
  Works for multiple superclass/interfaces with **`&`**
  `<Type extends ClassA & InterfaceB & InterfaceC & ...>`

- **`<Type super SuperType>`**
  - A lower bound; accepts the given supertype or any of its supertypes

- Example

```
// tree set works for any comparable type
public class TreeSet<T extends Comparable<T>> {
     ...
}
```

Complex bounded types

- ```
  public static <T extends Comparable<T>>
      T max(Collection<T> c)
  ```
  - Find max value in any collection (if the elements can be compared)

- ```
  public static <T> void copy(
     List<T2 super T> dst, List<T3 extends T> src)
  ```
  - Copy all elements from **`src`** to **`dst`**
  - **`dst`** must be able to safely store anything that could be in **`src`**
  - This means that all elements of **`src`** must be of **`dst`**'s element type or a subtype

- ```
  public static <T extends Comparable<T2 super T>>
      void sort(List<T> list)
  ```
  - Sort any list whose elements can be compared to the same type or a broader type
  ```

Get/Put Principle

- Should you insert wildcards everywhere, and if so, **`extends`** or **`super`** or neither?
- Get/Put principle
  - Use **`? extends T`** when you *get* values from something
  - Use **`? super T`** when you *put* values into something
  - Use neither (just **`T`**, not **`?`**) if you do both

```
public static <T> void copy(
     List<? super T> dst,
     List<? extends T> src)
```

Reminder: what's the point?

- To decrease the chance that programmers make mistakes about types during execution
- More complicated declarations and instantiations, along with added compile-time checking is the cost
- Generics usually clarify the implementation
  - sometimes ugly: wildcards, arrays, instantiation
- Generics always make the client code prettier and safer

```
interface Map {
  Object put(Object key, Object value);
  equals(Object other);
}

interface Map<Key,Value> {
  Value put(Key key, Value value);
  equals(Object other);
}
```

Example: a generic interface

```
// Represents a list of values
public interface List<E> {
    public void add(E value);
    public void add(int index, E value);
    public E get(int index);
    public int indexOf(E value);
    public boolean isEmpty();
    public void remove(int index);
    public void set(int index, E value);
    public int size();
}

public class ArrayList<E> implements List<E> { ...

public class LinkedList<E> implements List<E> { ...
```

Generic methods

```
public static <Type> returnType name(params) {
```

- When you want to make just a single (often static) method generic in a class, precede its return type by type parameter(s)

```
public class Collections {
   ...
  public static <T> void copy(List<T> dst, List<T> src) {
    for (T t : src) {
      dst.add(t);
    }
  }
}
```

Type erasure

- All generic types become type **`Object`** once compiled
  - Big reason: backward compatibility with old byte code
  - So, at runtime, all generic instantiations have the same type

```
List<String>  lst1 = new ArrayList<String>();
List<Integer> lst2 = new ArrayList<Integer>();
lst1.getClass() == lst2.getClass()  // true
```

- You cannot use **`instanceof`** to discover a type parameter

```
Collection<?> cs = new ArrayList<String>();
if (cs instanceof Collection<String>) {
  // illegal
}
```

Generics and casting

- Casting to generic type results in a warning

```
List<?> lg = new ArrayList<String>();  // ok
List<String> ls = (List<String>) lg;   // warn
```

- The compiler gives an unchecked warning, since this isn't something the runtime system is going to check for you
- Usually, if you think you need to do this, you're wrong
  - (Unless you’re implementing things like **`ArrayList`** – and then be sure you understand why you’re getting the warning)

- The same is true of type variables:

```
public static <T> T badCast(T t, Object o) {
  return (T) o;   // unchecked warning
}
```

Generics and arrays

```
public class Foo<T> {
    private T aField;                          // ok
    private T[] anArray;                       // ok


    public Foo(T param) {
        aField = new T();                      // error
        anArray = new T[10];                   // error
    }
}
```

- You cannot create objects or arrays of a parameterized type

Generics/arrays: a hack

```
public class Foo<T> {
    private T aField;                          // ok
    private T[] anArray;                       // ok

    @SuppressWarnings("unchecked")
    public Foo(T param) {
        aField = param;                          // ok
        T[] a2 = (T[]) (new Object[10]); // ok
    }
}
```

- You *can* create variables of that type, accept them as parameters, return them, or create arrays by casting **`Object[]`**
  - Casting to generic types is not type-safe, so it generates a warning
  - You almost surely don't need this in common situations!

Comparing generic objects

```
public class ArrayList<E> {
    ...
    public int indexOf(E value) {
        for (int i = 0; i < size; i++) {
        // if (elementData[i] == value) {
            if (elementData[i].equals(value)) {
                return i;
            }
        }
        return -1;
    }
}
```

- When testing objects of type **`E`** for equality, must use **`equals`**

Tips when writing a generic class

- Start by writing a concrete instantiation
  - It's often easier to reason about a concrete instance than an abstraction of that instance
- Get it correct (testing, reasoning, etc.)
- Consider writing a second concrete version
  - It's still often easier to reason about a concrete instance than an abstraction of that instance
- Generalize it by adding type parameters
  - Think about which types are the same & different
  - Not all ints are the same, nor are all Strings
  - The compiler will help you find errors
- Eventually, it will be easier to write the code generically from the start
  - but maybe not yet