Topics

- background and goals of generic programming
- basics of generic classes = parameterized types
- generic methods for general algorithms
- inheritance rules for generic types
- bounded type parameters
- generic code and the Java Virtual Machine
- restrictions and limitations
- wildcard types and wildcard type capture

Why generic programming

Background

- old version 1.4 Java collections were Object-based and required the use of ugly casts
  - cannot specify the exact type of elements
  - must cast to specific classes when accessing

Java generics

- lets you write code that is safer and easier to read
- is especially useful for general data structures, such as ArrayList
- generic programming = programming with classes and methods parameterized with types

Why generic programming (cont.)

- generic types are a powerful tool to write reusable object-oriented components and libraries
- however, the generic language features are not easy to master and can be misused
  - their full understanding requires the knowledge of the type theory of programming languages
    - especially covariant and contravariant typing
- the following introduces the main aspects of Java generics and their use and limitations
- we mostly inspect illustrative samples of what is and what is not allowed, with some short glimpses inside the JVM implementation
Why generic programming (cont.)

Java generics
• in principle, supports statically-typed data structures
  – early detection of type violations
  • cannot insert a string into ArrayList <Number>
  – also, hides automatically generated casts
• superficially resembles C++ templates
  – C++ templates are factories for ordinary classes and functions
  • a new class is always instantiated for given distinct generic parameters (type or other)
• in Java, generic types are factories for compile-time entities related to types and methods

Definition of a simple generic class

class Pair <T> {
  public T first;
  public T second;
  public Pair (T f, T s) { first = f; second = s; }
  public Pair () { first = null; second = null; }
}

• you instantiate the generic class by substituting actual types for type variables, as: Pair <String>
• you can think the result as a class with a constructor
  public Pair (String f, String s), etc . . .
• you can then use the instantiated generic class as it were a normal class (almost):
  Pair <String> pair = new Pair <String> ("1","2");

Multiple type parameters allowed

• you can have multiple type parameters

class Pair <T, U> {
  public T first;
  public U second;
  public Pair (T x, U y) { first = x; second = y; }
  public Pair () { first = null; second = null; }
}

• to instantiate: Pair <String, Number>

Generic static algorithms

• you can define generic methods both inside ordinary classes and inside generic classes

class Algorithms { // some utility class
  public static <T> T getMiddle (T [ ] a) {
    return a [ a.length / 2 ];
  }
  ...
}

• when calling a generic method, you can specify type
  String s = Algorithms.<String>getMiddle (names);
• but in most cases, the compiler infers the type:
  String s = Algorithms. getMiddle (names);
Inheritance rules for generic types

- Pair<Manager> matches Pair<? extends Employee> => subtype relation (covariant typing)
- Pair<Object> matches Pair<? super Employee> => subtype relation (contravariant typing)
- Pair<Employee> can contain only Employees, but Pair<Object> may be assigned anything (Numbers) => no subtype relation
- also: Pair<T> <= Pair<?> <= Pair (raw)

List<String> sl = new LinkedList<String> ();
List x = sl; // OK
x.add (new Integer (5)); // type safety warning
String str = sl.get (0); // throws ClassCastException.

Comments on inheritance relations

- Pair<Manager> matches Pair<? extends Employee> => subtype relation (covariant typing)
- Pair<Object> matches Pair<? super Employee> => subtype relation (contravariant typing)
- Pair<Employee> can contain only Employees, but Pair<Object> may be assigned anything (Numbers) => no subtype relation
- also: Pair<T> <= Pair<?> <= Pair (raw)

Bounds for type variables

- consider the min algorithm: find the smallest item in a given array of elements
- to compile this, must restrict T to implement the Comparable interface that provides compareTo

public static <T extends Comparable>T min (T [ ] a) {
  // this is almost correct
  if (a.length == 0) throw new InvalidArg.. ();
  T smallest = a [0];
  for (int i = 1; i < a.length; i++)
    if (smallest.compareTo (a [i]) > 0) // T constraint
      smallest = a [i];
  return smallest;
}

Bounds for type variables (cont.)

- however, Comparable is itself a generic interface
- moreover, any supertype of T may have extended it

public static <T extends Object & Comparable <? super T>>
  T min (T [ ] a) { ... } // the more general form
  T smallest = a [0];
  for (int i = 1; i < a.length; i++)
    if (smallest.compareTo (a [i]) > 0) // T constraint
      smallest = a [i];
  return smallest;

- cannot inherit multiple different instantiations of the same generic type (class or interface)
- an inherited generic type is fixed for subtypes, too
Generic code and the JVM

- the JVM has no instantiations of generic types
- a generic type definition is compiled once only, and a corresponding raw type is produced
  - the name of the raw type is the same name but type variables removed
- type variables are erased and replaced by their bounding types (or Object if no bounds); e.g.:
  ```java
class Pair {                 // the raw type for Pair <T>
    public Object first;
    public Object second;
    public Pair (Object f, Object s) { . . }
  }
```
- byte code has some generic info, but objects don't

Generic code and the JVM (cont.)

- Pair <String> and Pair <Employee> use the same bytecode generated as the raw class Pair
- when translating generic expressions, such as
  ```java
  Pair <Employee> buddies = new Pair < . . ;
  Employee buddy = buddies.first;
  ```
  - the compiler uses the raw class and automatically inserts a cast from Object to Employee:
    ```java
    Employee buddy = (Employee)buddies.first;
    ```
- in C++, no such casts are required since class instantiations already use specific types
- if multiple constraints (Object & Comparable . .) then the type parameter is replaced by the first one

Overriding of methods of generic type

- consider a generic class with a non-final method:
  ```java
  class Pair <T> {   // parameter T is erased from code
    public void setSecond (T s) { second = s; } . .
  }
  ```
- to override such type-erased methods, the compiler must generate extra bridge methods:
  ```java
  class DateInterval extends Pair <Date> {
    public void setSecond (Date high) { // override
      if (high.compareTo (first) < 0) throw new . .
      second = high;  // otherwise OK
    }
    public void setSecond (Object s) { // bridge method
      setSecond ((Date)s);     // generated by compiler
    }
  }
  ```

Restrictions and limitations

- in Java, generic types are compile-time entities
  - in C++, instantiations of a class template are compiled separately as source code, and tailored code is produced for each one
- primitive type parameters (Pair <int>) not allowed
  - in C++, both classes and primitive types allowed
- objects in JVM have non-generic classes:
  ```java
  Pair<String> strPair = new Pair<String> . .;
  Pair<Number> numPair = new Pair<Number> . . ;
  b = strPair.getClass () == numPair.getClass ();
  assert b == true;  // both of the raw class Pair
  ```
- but byte-code has reflective info about generics

- in Python, generic types are compile-time entities
  - in C++, instantiations of a class template are compiled separately as source code, and tailored code is produced for each one
- primitive type parameters (Pair <int>) not allowed
  - in C++, both classes and primitive types allowed
- objects in JVM have non-generic classes:
  ```java
  Pair<String> strPair = new Pair<String> . .;
  Pair<Number> numPair = new Pair<Number> . . ;
  b = strPair.getClass () == numPair.getClass ();
  assert b == true;  // both of the raw class Pair
  ```
- but byte-code has reflective info about generics

Restrictions and limitations (cont.)
- instantiations of generic parameter T are not allowed
  ```java
  new T()          // ERROR: whatever T to produce?
  new T[10]
  ```
- arrays of parameterized types are not allowed
  ```java
  new Pair<String>[10];         // ERROR
  ```
  since type erasure removes type information needed for checks of array assignments
- static fields and static methods with type parameters are not allowed
  ```java
  class Singleton<T> {
    private static T singleOne;         // ERROR
  }
  ```
  since after type erasure, one class and one shared static field for all instantiations and their objects

Wildcard types
- note that the raw class `Pair` is not equal `Pair<?>
  ```java
  Pair pair1 = . .;
  pair1.first = new Double(10.0); // WARNING
  Pair<?> pair2 = . .;
  pair2.first = new Double(10.0); // ERROR
  ```
- but some operations have no type constraints:
  ```java
  public static boolean hasNulls(Pair<?> p) {
    return p.first == null || p.second == null;
  }
  ```
  alternatively, you could provide a generic method
  ```java
  public static <T> boolean hasNulls(Pair<T> p)
  ```
  generally, prefer wildcard types (but use generic method with type T when multiple parameters)

Wildcard capture
- the wildcard type `?` cannot be used as a declared type of any variables (as in the previous slide)
  ```java
  Pair<?> p = new Pair<String>("one", "two"); . .
  p.first = p.second; // ERROR: unknown type
  ```
- but, can sometimes use a generic method to capture the wildcard:
  ```java
  public static <T> void rotate(Pair<T> p) {
    T temp = p.first; p.first = p.second;
    p.second = temp;
  }
  ```
  the compile checks that such a capture is legal
  - e.g., the context ensures that T is unambiguous

Collections and algorithms
- goal: design a minimal interface that you need
- e.g., for `max`, implement to take any `Collection`
  ```java
  public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> c) {
    // a hypothetical implementation:
    Iterator<T> it = c.iterator();
    T largest = it.next(); // or throws NoSuchElement
    while (it.hasNext()) {
      T val = it.next();
      if (largest.compareTo(val) < 0) largest = val;
    }
    return largest;
  }
  ```