Generics with Type Bounds

Lecture 27

Generic Methods

- Like classes, methods can be generic
  ```java
class ArrayOps { // ordinary nongeneric class
  static <T> T midpoint(T[] A);  
  <T> int nonNullLength(T[] A);
  }
```
- Scope of type parameter limited to method
- Instantiation with a specific parameter type not needed when invoking method
  - Parameter type is inferred from arguments
    ```java
    String s = ArrayOps.midpoint(args);
    Date d = ArrayOps.midpoint(timeline);
    int c = arrayWorker.nonNullLength(args);
    (Can also use return type, when assigned)
    But explicit type invocation is legal too
    ```
    ```java
    i = MathUtilities.<Integer>max(42, 34);
    ```

Example: Generic Methods

```java
class ArrayOps {
  public static <T> T midpoint(T[] A) {
    assert A.length >= 1;
    return A[A.length/2];
  }
  public <T> int nonNullLength(T[] A) {
    int count = 0;
    for (T t : A)
      if (t != null) count++;
    return count;
  }
  public static void main(String[] args) {
    ArrayOps arrayWorker = new ArrayOps();
    String s1 = ArrayOps.midpoint(args);
    String s2 = ArrayOps.<String>midpoint(args);
    int x = arrayWorker.nonNullLength(args);
    int y = arrayWorker.<String>nonNullLength(args);
  }
}
```

Example: Type Bounds

```java
class Filter {
  static <T>
    T max(T t1, T t2) {
    return (t1.compareTo(t2) <= 0 ? t2 : t1);
  }
}
```

BigNatural nat1 = ...
BigNatural nat2 = ...
System.out.println(Filter.max(nat1, nat2));

Question: Why not This Way?

```java
class Filter {
  static <T>
    Comparable<T> max(Comparable<T> t1, 
                    Comparable<T> t2) {
    return (t1.compareTo(t2) <= 0 ? t2 : t1);
  }
}
```

BigNatural nat1 = ...
BigNatural nat2 = ...
System.out.println(Filter.max(nat1, nat2));
Example: Type Bounds

```java
class Filter {
    static <T extends Comparable<T>>
    T max(T t1, T t2) {
        return (t1.compareTo(t2) <= 0 ? t2 : t1);
    }
}
BigNatural nat1 = ...;
BigNatural nat2 = ...;
System.out.println(Filter.max(nat1, nat2));
```

Arrays and Inheritance

- Consider 3 types: Student, Person, Object
  - Student extends Person, Person extends Object
- Subtyping: A Student "is a" Person
  - A Student can do everything a Person can do
    - Client would rather have Student to use
    - Implementer would rather write Person
  - Code expecting a Person, can be given a Student
    - boolean older (int age, Person p);
- Question: A Student[] "is a" Person[]?
  - Can a Student[] do everything a Person[] can do?
  - Can code expecting a Person[] be given a Student[] instead?
    - boolean allOlder(int age, Person[] ps);

Arrays and Co/Contra-Variance

- Student[] is a Person[], Person[] is an Object[]
  - boolean allOlder (int age, Person[] ps) {
    for (Person p : ps)
      if (p.getAge() < age) result = false;
    return result;
  }
- Student[] is a Person[]
  - Consequence: We live dangerously
    - If the wrong type of object is assigned to an array element, ArrayStoreException is thrown

Strawman 1: Covariance

- Student[] is a Person[], Person[] is an Object[]
  - void populateClass(Student[] roster) {
    for (int i=0; i<roster.length; i++)
      roster[i] = new Student();
  }
- Student[] is a Person[]
  - Student[] roster = ...
  - //assert: roster contains only Students
  - clobberFirst(roster);
  - //trouble: Dynamic type of roster[0] is Infant
  - roster[0].grantDegree();

Strawman 2: Contravariance

- Person[] is a Student[], Person[] is a Object[]
  - void populateClass(Student[] roster) {
    for (int i=0; i<roster.length; i++)
      roster[i] = new Student();
  }
- Person[] is a Student[]
  - Person[] ps = ...
  - graduate(ps);

Java’s Choice

- Neither is right!
  - A Student[] can not do everything a Person[] can do!
    - e.g. it can not contain an Infant
  - A Person[] can not do everything a Student[] can do!
    - e.g. it can not calculate a max GPA
- Java’s choice: Covariance
  - Student[] is a Person[]
  - Consequence: We live dangerously
    - If the wrong type of object is assigned to an array element, ArrayStoreException is thrown
Generics and Wildcards

- Wildcard ?: Refers to stack of any kind
  
  Stack<?>

- Example
  
  boolean largeSize(int limit, Stack<?> s) {
    if (s.size() > limit) return true;
    else return false;
  }

- Subtyping: Every Stack is a Stack<?>
  
  Stack<String> args = ...;
  Stack<People> crew = ...;
  flag = largeSize(3, args);  //ok
  flag = largeSize(32, crew); //ok

Generics and Inheritance

- Is a Stack<Student> a Stack<Person>?
  
  Can a Stack<Student> do everything a Stack<Person> can do?
  Can code expecting a Stack<Person> be given a Stack<Student> instead?

- Java’s choice:
  
  No!
  For a generic class G, there is no implicit subtyping relationship between G<A> and G<B>
  Neither covariance nor contravariance
  Regardless of any subtyping relationship between A and B

Generics: Co/Contra-variance

- Similar to arrays
  
  Sometimes covariance is ok
  Sometimes contravariance is ok

- Consider code written for Stack<Person>
  
  boolean someMethod(Stack<Person> s);

- Questions:
  
  Can a Stack<Student> be passed in instead?
  Can a Stack<Object> be passed in instead?

- Answer:
  
  It depends on what client code does with s!
  Some code works fine for Stack<Student>
  Some code works fine for Stack<Object>

Both Forms

- Example 1: Getting from stack
  
  int firstAge(Stack<Person> s) {
    Person p = s.pop();
    return p.getAge();
  }

  Works when argument is a Stack<Student>
  Does not work when given a Stack<Object>

- Example 2: Putting into stack
  
  void addChild(Stack<Person> s) {
    s.push(new Person(3));
  }

  Works when argument is a Stack<Object>
  Does not work when given a Stack<Student>

Upper Type Bounds: Covariance

- Combine wildcard with type bound
  
  Stack<? extends Person>

  Person is an upper bound on type parameter

- Reflects covariant relationship
  
  int firstAge(Stack<? extends Person> s) { Person p = s.pop(); return p.getAge(); }

- Use when code “gets” from generic

List<? extends Number> figures = new ArrayList<Number>(); // OK
figures = new ArrayList<Integer>(); // OK
figures = new ArrayList<Object>(); // compiler error

Lower Type Bounds: Contravariance

- Combine wildcard with type bound
  
  Stack<? super Person>

  Person is a lower bound on type parameter

- Reflects contravariant relationship
  
  void addChild(Stack<? super Person> s) {
    s.push(new Person(3));
  }

- Use when client code "puts" to generic

List<? super Number> figures = new ArrayList<Number>(); // OK
figures = new ArrayList<Object>(); // OK
figures = new ArrayList<Integer>(); // compiler error
figure = 'Wrong type'
Summary

- Generic methods
  - Type parameter applied to individual methods

- Inheritance and arrays
  - Java arrays are covariant in their base type
  - This is not type safe (wrong stores cause exception)

- Inheritance and generics: type bounds
  - Use upper type bound when getting
  - Use lower type bound when putting
  - Use exact type when doing both