Today’s Topics

What are Generics
- And why would you want to use them

Generics and Typing
- Univariance
- Wildcards
- Generic methods

How are they implemented (and why you should care)
- erasure and its consequences

© Robert Miller 2009
WHAT ARE GENERICS
You have used generics already

```java
List<String> ls = new ArrayList<String>();
ls.add("Hello");
ls.add("World");
String hello = ls.get(0);
```

<String> is a type parameter for the list

What does this type parameter buy us?

- At runtime, the list of String is just like a list of Number
- But at compile time, the type parameter helps the compiler check that the list is used correctly
public interface List {
    void add(Object x);
    Object get();
    Iterator iterator();
}

public interface Iterator{
    Object next();
    boolean hasNext();
}

List ls = new ArrayList(); // List should contain strings
ls.add("Hello");
ls.add(new Integer(5));
String hello = (String) ls.get(0);
Why Generics

For type checking purposes, you want to have

- List<Integer>, List<String>, List<Foo>, ...

But you can’t possibly maintain a separate list for each type

Prehistoric Java settled for a single list of objects

- After all, everything is an object
- This isn’t ideal

List ls = new ArrayList(); // List should contain strings
ls.add("Hello");
ls.add(new Integer(5));
String hello = (String) ls.get(0);
Generics = parameterized types

```java
public interface List<E>{
    void add(E x);
    E get();
    Iterator<E> iterator();
}
```

```java
public interface Iterator<E>{
    E next();
    boolean hasNext();
}
```

Only one interface/class to maintain

Can be instantiated with an arbitrary type

- List<Integer>, List<String>, List<Foo>
Benefits of generics

List<String> ls = new ArrayList<String>();
ls.add("Hello");
ls.add(new Integer(5)); // This is a compile time error
String hello = ls.get(0);

Comments are replaced by statically checked type parameters
Compiler enforces type safety
No need for messy typecasts
GENERICS AND TYPING
Subtyping

What does it mean for A to be a subtype of B?

- A can be used anywhere in place of B

So far we have seen two mechanisms for introducing subtyping

- **Interfaces:** class A implements B{ ... }
- **Subclasses:** class A extends B{ ... }

Ex:

```java
void rideVehicle(Vehicle v);

class Car extends Vehicle{ ... }
class Spaceship extends Vehicle{ ... }

Car c = new Car();
Spaceship sh = new Spaceship();

rideVehicle(c);
rideVehicle(sh);
```
Subtyping

Trick Question:

Should List<Car> be a subtype of List<Vehicle>?

class Car extends Vehicle{ ... }
class Spaceship extends Vehicle{ ... }

void rideVehicles(List<Vehicle> lv);

List<Car> lcar = ... 
List<Spaceship> lship = ...

rideVehicles(lcar); // is this legal?
rideVehicles(lship); // should it be?
Subtyping

Trick Question:

Should List<Car> be a subtype of List<Vehicle>?

void addVehicles(List<Vehicle> lv){
    lv.add( new Car() );
    lv.add( new Spaceship() );
}

List<Car> lcar = ... 
List<Spaceship> lship = ...

addVehicles(lcar);  // is this legal ?
addVehicles(lship);  // should it be ?
Subtyping

Just because Car is a subtype of Vehicle doesn’t mean List<Car> should be a subtype of List<Vehicle>

- This helps maintain the static guarantees of generic classes
- It can also be a pain when trying to write reusable code

```java
void addVehicles(List<Vehicle> lv){
    lv.add( new Car() );
    lv.add( new Spaceship() );
}

List<Car> lcar = …
List<Spaceship> lship = …

addVehicles(lcar);  // illegal call
addVehicles(lship); // illegal call
```

© Robert Miller 2009
Subtyping

Just because Car is a subtype of Vehicle doesn’t mean List<Car> should be a subtype of List<Vehicle>

- This helps maintain the static guarantees of generic classes
- It can also be a pain when trying to write reusable code

```java
void printVehicles(List<Vehicle> lv){
    for(Vehicle v : lv){
        v.printMe();
    }
}

List<Car> lcar = ...
List<Spaceship> lship = ...

printVehicles(lcar); // illegal call
printVehicles(lship); // illegal call
```

© Robert Miller 2009
Wildcards

The symbol ? in the type parameter is a wildcard

- List<?> is a supertype of List<X> for any X.
- List<? extends Y> is a supertype of List<X> if Y is a supertype of X

```java
void printVehicles(List<? extends Vehicle> lv){
    for(Vehicle v : lv){
        v.printMe();
    }
}

List<Car> lcar = ...  
List<Spaceship> lship = ...

printVehicles(lcar);
printVehicles(lship);
```
Wildcards

What should the type parameter for the list below be?

```java
void addCars(List<           > lv){
    lv.add(new Car());
    lv.add(new Car());
}

List<Vehicle> lveh = ... ;
List<Car> lcar = ... ;

addCars(lveh);
addCars(lcar);
```
Wildcards

Express the fact that any superclass of cars will be acceptable

```java
void addCars(List<? super Cars> lv){
    lv.add(new Car());
    lv.add(new Car());
}
```

List<Vehicle> lvveh = ... ;
List<Car> lcarn = ... ;

addCars(lvveh);
addCars(lcarn);

© Robert Miller 2009
Wildcards

General rule:
- Use Container<? extends A> when you want to read from the container
- Use Container<? super A> when you want to write to the container
Generic Methods

Problem:

➢ Write a method that copies elements from one list to another

```
static void copyList(List<?> out, List<?> in){
    ...
}
Will this work?
```
Generic Methods

Generic methods allow you to establish relationships between type parameters

```java
static <T> void copyList(List<T> out, List<T> in){
    for(T e : in){
        out.add(e);
    }
}
```
Generic Methods

With generic methods you can establish relationships between type parameters

```java
static <T> void copyList(List<? super T> out, List<? extends T> in){
    for(T e : in){
        out.add(e);
    }
}
```
Generic Methods

What’s the difference

```java
void printVehicles(List<? extends Vehicle> lv){
    for(Vehicle v : lv){
        v.printMe();
    }
}

<T extends Vehicle> void printVehicles(List<T> lv){
    for(Vehicle v : lv){
        v.printMe();
    }
}
```

© Robert Miller 2009
HOW ARE THEY IMPLEMENTED
Erasure

Java implements generics through Type erasure

What does that mean?

- It means it uses the type parameters to do type checking and then removes them
- The bytecode knows nothing about generics

Erasure makes the implementation simple

- easy to do separate compilation
- easy to have backwards compatibility

But, it leads to some important limitations in the use of generics
Can’t construct new objects of type T

Is the following code legal?

```java
<T> void defaultPopulate(List<T> lv, int n){
    for(int i=0; i<n; ++i){
        lv.add( new T() );
    }
}
List<Car> lc;
defaultPopulate(lc, 5);
```

Why not?
No arrays of generic classes

What’s wrong with this?

```java
List<Car>[] carListArr = new ArrayList<Car>[5];
```

You need to know that in java

```java
if A is a subtype of B,
    A[] is a subtype of B[]
```

```java
Foo[] fooArr = new Foo[5];
Object[] oArr = fooArr;
```

```java
oArr[0] = new Boo();
```

```java
Foo x = fooArr[0];
```

© Robert Miller 2009
No arrays of generic classes

What’s wrong with this?

List<Car>[] carListArr = new ArrayList<Car>[5] ;

Object[] objList = carListArr;

List<Spaceship> shList = new ArrayList<Spaceship>();
shList.add(new Spaceship());
objList[0] = shList;

List<Car> cList = carListArr[0];
Car c = cList.get(0);
No arrays of generic classes

What's wrong with this?

```java
List<?> carListArr = new ArrayList<?>[5];
Object[] objList = carListArr;
List<Spaceship> shList = new ArrayList<Spaceship>();
shList.add(new Spaceship());
objList[0] = shList;
List<?> clist = carListArr[0];
Car c = (Car) clist.get(0);
```
Beware of instanceof and casting

What’s wrong with the code below?

```java
List<Spaceship> shList = new ArrayList<Spaceship>();
Object lst = shList;
if(lst instanceof List<Spaceship>){
    List<Spaceship> sl2 = (List<Spaceship>) lst;
}
```
Conclusions

- Generics are a powerful mechanism to make your code more reusable.
- You should be aware of subtle subtyping rules.
- Writing good generic code can be tricky.