

## Java Generics



## Example

- You can use it with different types

```
Person a = new Person("Al", "A", new Integer(123));  
Person b = new Person("Pat", "B", "s32");
```

- You may need casts..

```
Integer id = (Integer) a.getID();
```

- ..that may be dangerous

```
Integer id = (Integer) b.getID();
```

ClassCastException  
at run-time

## Problems

- Often you need the same behavior for different kind of classes
  - ♦ Use Object references to accommodate any object type
  - ♦ Use Generic classes and Method
- The use of Object references induces cumbersome code

## Generic class

```
public class Person<T> {  
    String first;  
    String last;  
    T ID;  
  
    Person(String first,String last,T ID){  
        this.first = first;  
        this.last = last;  
        this.ID = ID;  
    }  
  
    T getID(){ return ID; }  
}
```

## Example

- We may need to represent ID of persons in different forms

```
public class Person {  
    String first;String last; Object ID;  
    Person(String f, String l, Object ID){  
        this.first = f;  
        this.last = l;  
        this.ID = ID;  
    }  
}
```

## Generics use

- Declaration is longer but...

```
Person<Integer> a = new Person<Integer>  
    ("Al", "A", new Integer(123));  
Person<String> b = new Person<String>  
    ("Pat", "B", "s32");
```

- ..use is more compact and safer

```
Integer id1 = a.getID();  
String id2 = b.getID();  
Integer ids = b.getID();
```

Compiler error:  
type mismatch

## Generic type declaration

- Syntax:

```
(class/interface) Name <P1 {,P2}>
```

- Parameters:

- ♦ uppercase letter
- ♦ usually: T(ype), E(lement), K(ey), V(alue)

## Example

- Using a list of Integers

- ♦ Without generics ( ArrayList list )

```
list.add(0, new Integer(42));  
int n= ((Integer)list.get(0)).intValue();
```

- ♦ With generics ( ArrayList<Integer> list )

```
list.add(0, new Integer(42));  
int n= ((Integer)list.get(0)).intValue();
```

- ♦ + autoboxing ( ArrayList<Integer> list )

```
list.add(0, new Integer(42));  
int total = list.get(0).intValue();
```

## Generic collections

- All collection interfaces and classes have been redefined as Generics
- Use of generics lead to code that is
  - ♦ safer
  - ♦ more compact
  - ♦ easier to understand
  - ♦ equally performing

## Example

- A class representing a point with different precisions

```
public class Point<T> {  
    T x;  
    T y;  
    public Point(T x, T y){  
        this.x = x;  
        this.y = y;  
    }  
}
```

## Generic list – excerpt

```
public interface List<E>{  
    void add(E x);  
    Iterator<E> iterator();  
}  
public interface Iterator<E>{  
    E next();  
    boolean hasNext();  
}
```

## Example

- Computing distance from origin:

```
public double distance(){  
return Math.sqrt(  
    x.doubleValue()*x.doubleValue()  
    + y.doubleValue()*y.doubleValue());  
}
```

method undefined for type T

- We need to bind T:

```
public class Point<T extends Number> {..}
```

## Bounded types

- Allow to express constraints when *defining* generic types

```
class C<T extends B1 { & B2 } >
```

- ♦ class C can be instantiated only with types derived from B1 (and B2 etc.)

## Bounded wildcard – example

- We need a generic sum function

```
List<Integer> list=new LinkedList<Integer>();  
sum(list);
```

- `double sum(List<Number> list)`
  - ♦ not applicable to `List<Integer>`
- `double sum(List<T extends Number> list)`
  - ♦ not a valid syntax, not defining a new type
- `double sum(List<? extends Number> list)`
  - ♦ we need to use a **bounded wildcard**

## Generics subtyping

- We must be careful about inheritance when generic types are involved
  - ♦ `String` is a subtype of `Object`
  - ♦ `List<String>` is **not** subtype of `List<Object>`

```
List<String> ls = new ArrayList<String>();  
List<Object> lo = ls;  
lo.add(new Object());  
String s = ls.get(0);
```

if this were legal then...

.. we could end up assigning an Object to a String reference

## Wildcards

- Allow to express (lack of) constraints when *using* generic types
- `G<?>`
  - ♦ G of unknown, unbounded
- `G<? extends B>`
  - ♦ upper bound: only sub-types of B
    - Including B
- `G<? super D>`
  - ♦ lower bound: only super-types of D
    - Including D

## Wildcard – example

- An attempt to have a generic method:

```
void printAll(Collection<Object> c) {  
    for (Object e: c)  
        System.out.println(e);  
}
```

- ♦ won't work with e.g. `Collection<String>`

- We ought to use a **wildcard**:

```
void printAll(Collection<?> c) { .. }
```

pronounced:  
Collection of unknown

## Lower bound – example

- A sorted collection should contain elements that can be ordered
  - ♦ An element E can be ordered if it directly implements `Comparable<E>`

```
interface SortedCollection  
    <E extends Comparable<E>>
```

- ♦ but also if any of its super-classes implements the relative `Comparable`

```
interface SortedCollection  
    <E extends Comparable<? super E>>
```

## Generic method declaration

- Syntax:  
`modifiers <T> ret_type name(pars)`
- pars can be:
  - ♦ as usual
  - ♦ T
  - ♦ type<T>

## Type erasure

- Classes corresponding to generic types are generated by **type erasure**
  - ♦ The erasure of a generic class is a **raw type**
    - where any reference to the parameters is substituted with the parameter erasure
  - ♦ Erasure of a parameter is the erasure of its first constraint
    - If no constraint then erasure is Object
  - ♦ The erasure of a non-generic type is the type itself

## Generic methods

- A generic method can be declared both in a common or generic class

```
<T> T method(T t)
<N extends Number> N method(List<N> t)
<N extends Number> void method(List<N> t)
void method(List<? extends Number> t)
```

Wildcards are more compact when a type parameter is used only once

## Type erasure – examples

- In: <T>
  - ♦ T ==> Object
- In: <T extends Number>
  - ♦ T ==> Number
- In: <T extends Number & Comparable>
  - ♦ T ==> Number

## Generics classes

- There is only one class generated (by the compiler) for each generic type declaration

```
Person<Integer> a = new Person<Integer>
    ("A1", "A", new Integer(123));
Person<String> b = new Person<String>
    ("Pat", "B", "s32");
boolean same=(a.getClass()==b.getClass());
```

believe it or not  
same is *true*

## Type erasure – consequences I

- Compiler makes control only when a generic type is used, not within it.
- Whenever a generic or a parameter is used a cast is added to its erasure
- `instanceOf` and `.class` cannot be used on generic types
  - ♦ valid for G<?> equivalent to the raw type

## Type erasure – consequences II

- It is not possible to instantiate an object of the generic's parameter type

```
class G<T> {  
    T[] toArray(){  
        T[] res = new T[n];  
        T t = new T();  
    }  
}
```

The compiler cannot instantiate these objects

- ♦ It is not possible to substitute the erasure in an instantiation statement

## Type erasure- consequences III

- Overload and override must be considered after type erasure

```
class Base<T> {  
    void m(int x){}  
    void m(Object t){}  
    void m(T t){}  
    void m(String s){}  
    <N extends Number> void m(N x){}  
    void m(List<?> l){}  
}
```

## Generics and inheritance

- Inheritance together with generic types leads to several possibilities
- It is not possible to implement two generic interfaces instantiated with different types

```
class Value implements Comparable<Value>  
class ExtValue extends Value  
    implements Comparable<ExtValue>
```