

Lecture 9: Generics (Ch 21)

Objectives

- ☞ To know the benefits of generics (§ 21.1).
- ☞ To use generic classes and interfaces (§ 21.2).
- ☞ To declare generic classes and interfaces (§ 21.3).
- ☞ To understand why generic types can improve reliability and readability (§ 21.3).
- ☞ To declare and use generic methods and bounded generic types (§ 21.4).
- ☞ To use raw types for backward compatibility (§ 21.5).
- ☞ To know wildcard types and understand why they are necessary (§ 21.6).
- ☞ To convert legacy code using JDK 1.5 generics (§ 21.7).
- ☞ To understand that generic type information is erased by the compiler and all instances of a generic class share the same runtime class file (§ 21.8).
- ☞ To know certain restrictions on generic types caused by type erasure (§ 21.8).
- ☞ To design and implement generic matrix classes (§ 21.9).

Utility Container: Stack/Queue/Tree/Hashtable

- ☞ Let us consider how to implement a stack of “things”, assuming the thing is class ClassA.

```
class Stack{
    ClassA elements[];
    int pos = 0;
    public void push(ClassA a) {
        elements[pos++]=a;
    }
    public ClassA pop(){
        return elements[--pos];
    }
    public void pushAll(Stack s){
        for(//there is element in me)
            s.push(pop());
    }
    public void popAll(Stack s) {
        for(//there is element in s)
            push(s.pop());
    }
}
```



Illustration

Customer won't buy your code

- ➡ It works with ClassAA that extends ClassA.
- ➡ It doesn't work with ClassB and ClassC

The big “O”, java.lang.Object comes to rescue!

```
class Stack{
    Object elements[];
    int pos = 0;
    public void push(Object a) {
        elements[pos++]=a;
    }
    public Object pop(){
        return elements[--pos];
    }
    public void pushAll(Stack s){
        for(//there is element in me)
            s.push(pop());
    }
    public void popAll(Stack s) {
        for(//there is element in s)
            push(s.pop());
    }
}
```



Version 2

Customer bought your code but asked for a refund later

- ☞ Your code works with ClassA and ClassB when “consuming” elements.

```
Stack s = new Stack();  
s.push(new ClassA());  
s.push(new ClassB());
```

- ☞ Your code requires casting when getting them “out of” the stack

```
ClassA a = (ClassA)s.pop();
```



What’s the problem with this way of programming?

Casting Objects

You have already used the casting operator to convert variables of one primitive type to another. *Casting* can also be used to convert an object of one class type to another within an inheritance hierarchy. In the preceding section, the statement

```
m(new Student());
```

assigns the object `new Student()` to a parameter of the `Object` type. This statement is equivalent to:

```
Object o = new Student(); // Implicit casting  
m(o);
```



(From Inheritance Lecture)

The statement `Object o = new Student()`, known as implicit casting, is legal because an instance of `Student` is automatically an instance of `Object`.

Why Casting Is Necessary?

Suppose you want to assign the object reference `o` to a variable of the `Student` type using the following statement:

```
Student b = o;
```

A compilation error would occur. Why does the statement **Object o = new Student()** work and the statement **Student b = o** doesn't?

```
o = new Student();  
o = new Teacher();  
b = o;
```

Enclose the target object type in parentheses and place it before the object to be cast, as follows:

```
Student b = (Student)o; // Explicit casting
```

(From Inheritance Lecture)

Casting from Superclass to Subclass

Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.

```
Apple x = (Apple) fruit;
```

```
Orange x = (Orange) fruit;
```

(From Inheritance Lecture)

Casting is bad

- ☞ Remember the devil: `void*`?
- ☞ If: `Object o = new Object()` and `Apple a = (Apple)o`, you get a runtime error.
- ☞ It was mostly used when you use collection classes such as “LinkedList”.
- ☞ No longer necessary with the use of Generics

(From Inheritance Lecture)

Motivating Example – Old Style


```
List stones = new LinkedList();
stones.add(new Stone(RED));
stones.add(new Stone(GREEN));
stones.add(new Stone(RED));
Stone first = (Stone)stones.get(0);
```

The cast is annoying
but essential!

```
public int countStones(Color color) {
    int tally = 0;
    Iterator it = stones.iterator();
    while (it.hasNext()) {
        Stone stone = (Stone)it.next();
        if (stone.getColor() == color) {
            tally++;
        }
    }
    return tally;
}
```

Why Do You Get a Warning?


```
public class ShowUncheckedWarning {  
    public static void main(String[] args) {  
        java.util.ArrayList list =  
            new java.util.ArrayList();  
        list.add("Java Programming");  
    }  
}
```



To understand the compile warning on this line, you need to learn Java generics.

Fix the Warning

```
public class ShowUncheckedWarning {  
    public static void main(String[] args) {  
        java.util.ArrayList<String> list =  
            new java.util.ArrayList<String>();  
        list.add("Java Programming");  
    }  
}
```



No compile warning on this line.

What is Generics?

- ➡ *Generics* is the capability to parameterize types.
- ➡ You can define a class or a method with generic types that can be substituted using concrete types by the compiler.
- ➡ For example, you may define a generic stack class that stores the elements of a generic type. From this generic class, you may create a stack object for holding strings and a stack object for holding numbers. Here, strings and numbers are concrete types that replace the generic type.

Remember Cake Molds?

Generics are sort of like components of a mold.

Generics are actually not classes. They need to assemble other classes to become classes



Old way:

A container of classes of uniform shapes



New way:

A container of classes of specific shapes

Why Generics?

☞ The key benefit of generics

- Enable errors to be detected at compile time rather than at runtime.
- A generic class or method permits you to specify allowable types of objects that the class or method may work with.
- If you attempt to use the class or method with an incompatible object, the compile error occurs.

Generic Type

```
package java.lang;

public interface Comparable {
    public int compareTo(Object o)
}
```

(a) Prior to JDK 1.5

```
package java.lang;

public interface Comparable<T> {
    public int compareTo(T o)
}
```

(b) JDK 1.5

Runtime error

```
Comparable c = new Date();
System.out.println(c.compareTo("red"));
```

(a) Prior to JDK 1.5

```
Comparable<Date> c = new Date();
System.out.println(c.compareTo("red"));
```

(b) JDK 1.5

Generic Instantiation

Improves reliability

Compile error

Generic Motivation

_interfaces.ComparableRectangle, ComparableRectangleWithGeneric

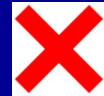
Stack in generics

```
class Stack<T>{
    T elements[];
    int pos = 0;
    public void push(T a) {
        elements[pos++]=a;
    }
    public T pop(){
        return elements[--pos];
    }
    public void pushAll(Stack<T> s){
        for(//there is element in me)
            s.push(pop());
    }
    public void popAll(Stack<T> s) {
        for(//there is element in s)
            push(s.pop());
    }
}
```

```
Stack<ClassA> s1 = new Stack<ClassA>();
s1.push(new ClassA());
ClassA a = s1.pop();
```

```
Stack<ClassB> s2 = new Stack<ClassB>();
s2.push(new ClassB());
ClassB b = s2.pop();
```

```
s2.push(new ClassA());
```



```
ClassB b1 = (ClassB) s1.pop();
```



Version 3

Generic ArrayList in JDK 1.5

java.util.ArrayList

```
+ArrayList()
+add(o: Object) : void
+add(index: int, o: Object) : void
+clear(): void
+contains(o: Object): boolean
+get(index: int) : Object
+indexOf(o: Object) : int
+isEmpty(): boolean
+lastIndexOf(o: Object) : int
+remove(o: Object): boolean
+size(): int
+remove(index: int) : boolean
+set(index: int, o: Object) : Object
```

(a) ArrayList before JDK 1.5

java.util.ArrayList<E>

```
+ArrayList()
+add(o: E) : void
+add(index: int, o: E) : void
+clear(): void
+contains(o: Object): boolean
+get(index: int) : E
+indexOf(o: Object) : int
+isEmpty(): boolean
+lastIndexOf(o: Object) : int
+remove(o: Object): boolean
+size(): int
+remove(index: int) : boolean
+set(index: int, o: E) : E
```

(b) ArrayList in JDK 1.5

No Casting Needed

```
ArrayList<Double> list = new ArrayList<Double>();
```

```
list.add(5.5); // 5.5 is automatically converted to new Double(5.5)
```

```
list.add(3.0); // 3.0 is automatically converted to new Double(3.0)
```

```
Double doubleObject = list.get(0); // No casting is needed
```

```
double d = list.get(1); // Automatically converted to double
```

Generic Types Restriction

- ➡ Generic types must be reference types.

- ➡ The following statement is wrong:

```
ArrayList<int> ints = new ArrayList<int>();
```

- ➡ Use wrapper class types:

```
ArrayList<Integer> ints = ...
```

Declaring Generic Classes and Interfaces

GenericStack<E>
-list: java.util.ArrayList<E>
+GenericStack() +getSize(): int +peek(): E +pop(): E +push(o: E): E +isEmpty(): boolean

An array list to store elements.

Creates an empty stack.

Returns the number of elements in this stack.

Returns the top element in this stack.

Returns and removes the top element in this stack.

Adds a new element to the top of this stack.

Returns true if the stack is empty.

GenericStack

Multiple Type Parameter

☞ Our coffee shop lunch menu

- Meat + Starch + Soup
- (Pork/Beef/Chicken)+(Rice/Italy noodle/Fries)+(Cream/Vegetable soup)

```
class Lunch<M,St,S> {  
    M meat_  
    St starch_  
    S soup_  
    public Lunch(M meat, St starch, S soup){  
        meat_ = meat;  
        starch_=starch;  
        soup_=soup;  
    }  
}
```

```
Beef b = new Beef();  
Rice r = new Rice();  
MushroomSoup s = new MushroomSoup();  
Lunch<Beef, Rice, MushroomSoup> l = new Lunch<Beef,  
Rice, MushroomSoup>(b,r,s);
```

Problem 1:

Incorrect Parameter Types

```
Node b = new Node();  
TreeNode r = new TreeNode();  
BTreeNode s = new BTreeNode();  
Lunch<Node, TreeNode, BTreeNode> l = new  
Lunch<Node, TreeNode, BTreeNode>(b,r,s);
```

We want to more properly say the following:

```
Lunch<subtypes of Meat, subtypes of Starch, subtypes of Soup>
```

Bounded Generic Types

```
public class Meat { ... }
public class Starch { ... }
public class Soup { ... }
public class Beef extends Meat { ... }
public class CreamSoup extends Soup { ... }
class Lunch <M extends Meat, St extends Starch, S extends
Soup> {
    M meat_;
    St starch_;
    S soup_;
    public Lunch(M meat, St starch, S soup){
        meat_ = meat;
        starch_=starch;
        soup_=soup;
    }
}
```


Generic Methods

```
public static <E> void print(E[] list) {  
    for (int i = 0; i < list.length; i++)  
        System.out.print(list[i] + " ");  
    System.out.println();  
}
```

```
public static void print(Object[] list) {  
    for (int i = 0; i < list.length; i++)  
        System.out.print(list[i] + " ");  
    System.out.println();  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Integer[] integers = {1,2,3,4,5};  
        String[] strings = {"London", "Paris", "New York", "Austin"};  
        Object[] objects = {"London", "Paris", "New York", "Austin", 1};  
        Test.<Integer>print(integers);  
        Test.<String>print(strings);  
        Test.print(objects);  
        Test.print1(integers);  
        Test.print1(strings);  
        Test.print1(objects);  
    }  
    public static <E> void print(E[] list) {  
        for (int i = 0; i < list.length; i++)  
            System.out.print(list[i] + " ");  
        System.out.println();  
    }  
    public static void print1(Object[] list) {  
        for (int i = 0; i < list.length; i++)  
            System.out.print(list[i] + " ");  
        System.out.println();  
    }  
}
```

GenericTest

Bounded Generic Type

```
public static void main(String[] args ) {  
    Rectangle rectangle = new Rectangle(2, 2);  
    Circle circle = new Circle(2);  
    System.out.println("Same area? " + equalArea(rectangle, circle));  
}
```

```
public static <E> boolean  
    equalArea(E object1, E object2) {  
    return object1.getArea() == object2.getArea();  
}
```



Difference?

```
public static <E extends GeometricObject> boolean  
    equalArea(E object1, E object2) {  
    return object1.getArea() == object2.getArea();  
}
```

BoundedTypeDemo

Bounded Generic Type

```
public static void main(String[] args ) {  
    Rectangle rectangle = new Rectangle(2, 2);  
    Circle circle = new Circle(2);  
    System.out.println("Same area? " + equalArea(rectangle, circle));  
}
```

```
public static boolean equalArea(GeometricObject o1, GeometricObject o2) {  
    return o1.getArea() == o2.getArea();  
}
```

```
public static <E extends GeometricObject> boolean  
    equalArea(E object1, E object2) {  
    return object1.getArea() == object2.getArea();  
}
```



Difference?

Problem 2:

Inheritance of Generic Types


➡ ClassA extends ClassB

➡ Subtype

- ClassA is a subtype of ClassB
- Stack<ClassA> is NOT a subtype of Stack<ClassB>

➡ Are we losing polymorphism?

- This creates trouble for us
- Customer requests a refund because they want more flexibility



Define
Subclasses of
Generic types
not required!

Are we losing polymorphism?


```
class Stack<T>{  
    public void pushAll(Stack<T> s){  
        for(//there is element in me)  
            s.push(pop());  
    }  
    public void popAll(Stack<T> s) {  
        for(//there is element in s)  
            push(s.pop());  
    }  
}
```



*//Let us say ClassB is a
//subclass of ClassA
Stack<ClassA> s1 = new Stack<ClassA>();*

Stack<ClassB> s2 = new Stack<ClassB>();

s2.pushAll(s1); 
*//should be OK since ClassB is also a
//type of ClassA. s1 that handles the
//super type can handle subtypes*

s1.popAll(s2); 
*//should be OK since ClassA is a super
//type of ClassB. Again, s1 should be
//able to take subtypes of ClassA.*

It seems that parameter s should behave polymorphically

We need “wild card”

- ➡ In defining a generic type
 - “E extends ClassA” represents all subtypes of ClassA
- ➡ In defining a container variable that behaves polymorphically
 - “? extends” represent any sub types.
 - “? super” represent any super types

Support Polymorphism



Versi
on 4

```
class Stack<T>{  
    public void pushAll(Stack<? super T> s){  
        for(//there is element in me)  
            s.push(pop());  
    }  
    public void popAll(Stack<? extends T> s) {  
        for(//there is element in s)  
            push(s.pop());  
    }  
}
```

```
Stack<ClassA> s1 = new Stack<ClassA>();  
Stack<ClassB> s2 = new Stack<ClassB>();  
s2.pushAll(s1);  
s1.popAll(s2);
```


Wildcards

Why wildcards are necessary? One more example.

WildCardDemo1

?	unbounded wildcard
? extends T	bounded wildcard
? super T	lower bound wildcard:

WildCardDemo2

WildCardDemo3

Raw Type and Backward Compatibility

// raw type

ArrayList list = new ArrayList();

This is roughly analogous to:

ArrayList<Object> list = new ArrayList<Object>();

Or more accurately can be thought of:

ArrayList<?> list = new ArrayList<?>(); // wildcard type

Raw Type is Unsafe

```
// Max.java: Find a maximum object
public class Max {
    /** Return the maximum between two objects */
    public static Comparable max(Comparable o1, Comparable o2) {
        if (o1.compareTo(o2) > 0)
            return o1;
        else
            return o2;
    }
}
```

Runtime Error:

Max.max("Welcome", 23);

Make it Safe

```
// Max1.java: Find a maximum object
public class Max1 {
    /** Return the maximum between two objects */
    public static <E extends Comparable<E>> E max(E o1, E o2) {
        if (o1.compareTo(o2) > 0)
            return o1;
        else
            return o2;
    }
}
```

Compile time error:
Max.max("Welcome", 23);

Erasure on Generics

Generics are implemented using an approach called *type erasure*.

```
public static <T> int count(T[] anArray, T elem) {  
    int cnt = 0;  
    for (T e : anArray)  
        if (e.equals(elem))  
            ++cnt;  
    return cnt;  
}
```

```
public static int count(Object[] anArray, Object elem) {  
    int cnt = 0;  
    for (Object e : anArray)  
        if (e.equals(elem))  
            ++cnt;  
    return cnt;  
}
```

Erasure on Generics

- ➡ Generics are a wrapper over `java.lang.Object` to ensure backwards compatibility
- ➡ Generics are implemented using an approach called *type erasure*.
 - Unbound variable replaced by “Object”
 - Bounded variable “T extends ClassA”, “? super ClassB” replaced by bound, “ClassA” or “ClassB”.
- ➡ The types used to instantiate generics serve as “comments”.
- ➡ What generics really is:
 - Compile time checks
 - Hints to generate the casting correctly.
 - Enforcing the usage of the same type

Compile Time Checking

For example, the compiler checks whether generics is used correctly for the following code in (a) and translates it into the equivalent code in (b) for runtime use. The code in (b) uses the raw type.

```
ArrayList<String> list = new ArrayList<String>();  
list.add("Oklahoma");  
String state = list.get(0);
```

(a)

```
ArrayList list = new ArrayList();  
list.add("Oklahoma");  
String state = (String)(list.get(0));
```

(b)

Checks and Hints

```
class Parent<T> {  
    T compute(T t){return t;}  
}
```

```
p = new Parent<String>();  
String s = p.compute( "§");  
String s = p.compute(circle);
```



```
class Parent{  
    Object compute(Object t){  
        return t;    }  
}
```

```
p = new Parent();  
String s = (String) p.compute( "§");  
String s = (String) p.compute(circle);
```

Generics tells us p can only deal with Strings . This information is stored in the compiler not in the code

Type Enforcement

```
class Parent<T extends Geom> {  
    T compute(T t1, T t2){return t1.same(t2);}  
}
```

```
p = new Parent<Circle>();  
String s = p.compute(circle, triangle);
```

```
class Parent{  
    Geom compute(Geom t1, Geom t2){  
        return t1.same(t2);    }  
}
```

```
p = new Parent();  
Geom = (Circle) p.compute((Circle)circle, (Circle)triangle);
```

Generics tells us these two parameters have to be the same type, not just subclass of Geom. This information is stored in the compiler not in the code

Method Overriding

- ➡ Apply the erasure concept before polymorphic matching



```
class T {  
  1: <E> void m(E e) { }  
}
```

```
class X extends T {  
  2: void m(Object o) { }  
}
```

```
T t = new X();
```

```
t.<Object>m(new Object()); dispatch to 2
```

```
t.<String>m(new String()); dispatch to 2
```

The generic T in X is type erased with “Object”. T becomes :

```
class T {  
  void m (Object e) { }  
}
```

Method Overriding

☞ Apply the erasure concept before polymorphic matching



```
class T {  
  1: <E> void m(E e) { }  
}
```

```
class X extends T {  
  2: void m(String o) { }  
}
```

```
T t = new X();
```

```
t.<Object>m(new Object()); dispatch to 1
```

```
t.<String>m(new String()); dispatch to 1
```

T will become:
class T {
 void m(Object o){ }
}, hence, m is overloaded,
not overridden.

Method Overriding

- ➡ Apply the erasure concept before polymorphic matching

```
class T {  
  1: void m(String e) { }  
}
```

```
class X<E> extends T {  
  2: void m(E o) { }  
}
```

```
T t = new X<String>();
```

```
t.m(new String());
```

dispatch to 1



X will become:
class X extends T {
 void m(Object o){ }
}, hence, m is overloaded,
not overridden.

Method Overriding

➡ Overriding works in this case

```
class T<E> {  
  1: void m(E e) { }  
}
```

```
class X extends T<String> {  
  2: void m(String o) { }  
}
```

```
class X {  
  void m(Object o){  
    m((String)o);  
  }  
  void m(String o){}  
}
```

```
T<String> t = new X();
```

```
t.m(new Object());
```

```
t.m(new String());
```

```
T t = new X();
```

```
t.m(new Object());
```



compile error

dispatch to 2

Special treatment by Java compiler

Important Facts

It is important to note that a generic class is shared by all its instances regardless of its actual generic type.

```
GenericStack<String> stack1 = new GenericStack<String>();  
GenericStack<Integer> stack2 = new GenericStack<Integer>();
```

Although GenericStack<String> and GenericStack<Integer> are two types, but there is only one class GenericStack loaded into the JVM.

Restrictions on Generics

- ➡ Restriction 1: Cannot Create an Instance of a Generic Type. (i.e., `new E()`).
- ➡ Restriction 2: Generic Array Creation is Not Allowed. (i.e., `new E[100]`).
- ➡ Restriction 3: A Generic Type Parameter of a Class is Not Allowed in a Static Context.
- ➡ Restriction 4: Exception Classes Cannot be Generic.

Not allowed in a static context

```
public class Test<E> {  
    public static void m (E o1) { // Illegal  
    }  
    public static E o1; // Illegal  
    static {  
        E o2; // Illegal  
    }  
}
```


Not allowed in exception classes

```
public class MyException<T> extends Exception {}
```

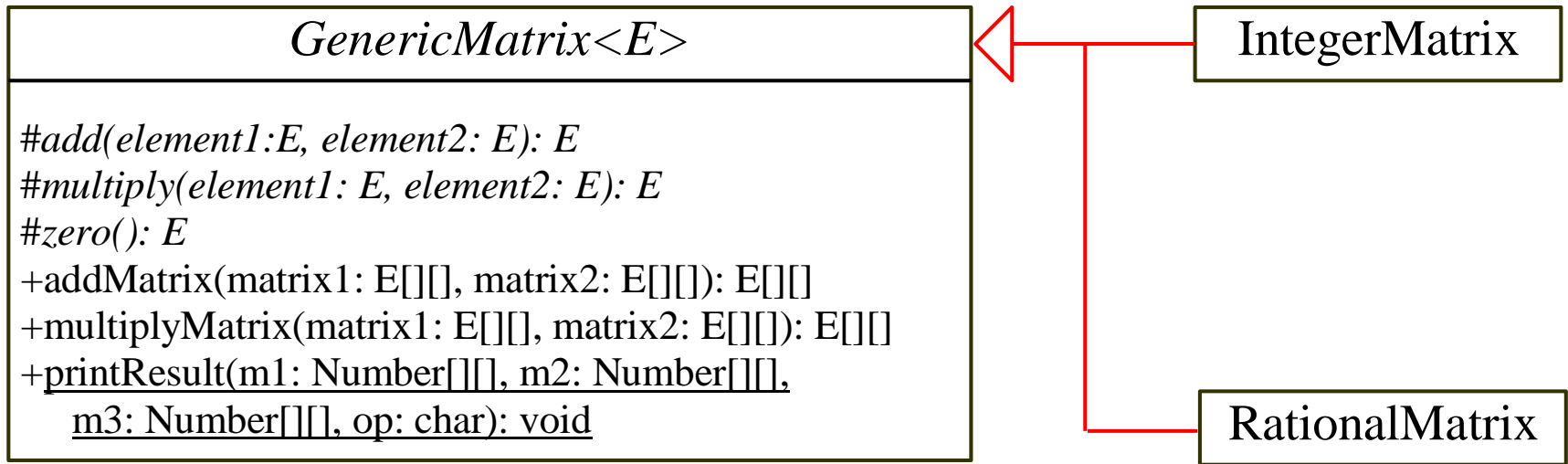
```
try {  
    ...  
}  
catch (MyException <Integer> ex) {  
    ...  
}  
catch (MyException<Circle> ex) {  
    ...  
}
```

Designing Generic Matrix Classes

- ➡ Objective: This example gives a generic class for matrix arithmetic. This class implements matrix addition and multiplication common for all types of matrices.

GenericMatrix

UML Diagram



Source Code

- ➡ Objective: This example gives two programs that utilize the GenericMatrix class for integer matrix arithmetic and rational matrix arithmetic.

IntegerMatrix

TestIntegerMatrix

RationalMatrix

TestRationalMatrix

Subclassing Generics

1. Child generic / Parent regular → Just another generic class

```
class Parent {}  
class Child<T> extends Parent {}
```

2. Child regular / Parent generic → Automatically parameterize parent using Object

```
class Parent<T> {}  
class Child extends Parent {} or  
class Child extends Parent<String> {}
```

Subclassing Generics

3. Child generic / Parent generic → Can either force homogeneous types or use automatic parameterization as in Case 2

```
class Parent<T> {}
```

```
class Child<T, E> extends Parent<E> {}
```

```
or class Child<T, E> extends Parent<String> {}
```

```
or class Child<T, E> extends Parent {}
```

4. Child regular / Parent regular → the old way

```
class Parent {}
```

```
class Child extends Parent {}
```