OOP with Java

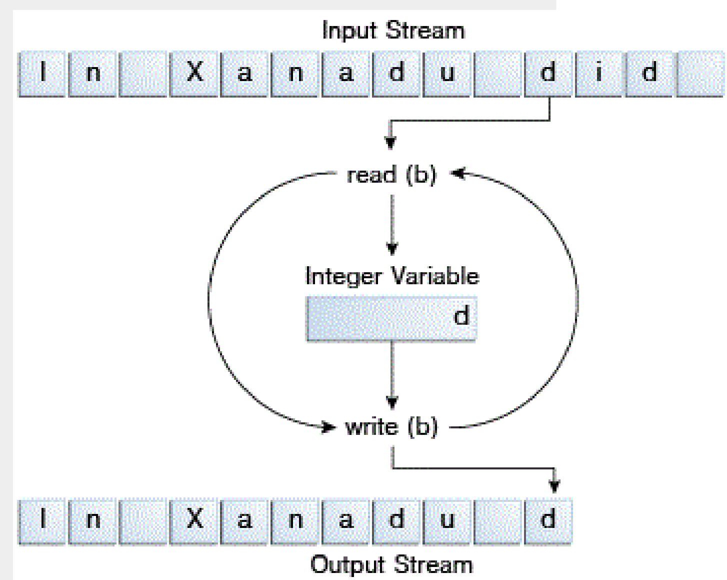
Yuanbin Wu
cs@ecnu

OOP with Java

- 通知
  - Project 7 :6 月 13 日晚 9 点
  - (暂时安排) 下周复习，下下周答疑
  - 考试时间？

- 复习
- I/O 流
  - InputStream/Reader
    - read()
  - OutputStream/Writer
    - write()
  - 抽象： 数据的来源 / 数据的目的地
    - ByteArrayStream, FileStream StringStream ObjectStream

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class CopyBytes {
    public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("xanadu.txt");
            out = new FileOutputStream("outagain.txt");
            int c;
            while ((c = in.read()) != -1)
                out.write(c);
        } finally {
            if (in != null)
                in.close();
            if (out != null)
                out.close();
        }
    }
}
```



- 装饰器
  - FilterInputStream/FilterOutputStream
    - BufferedInputStream
    - DataInputStream

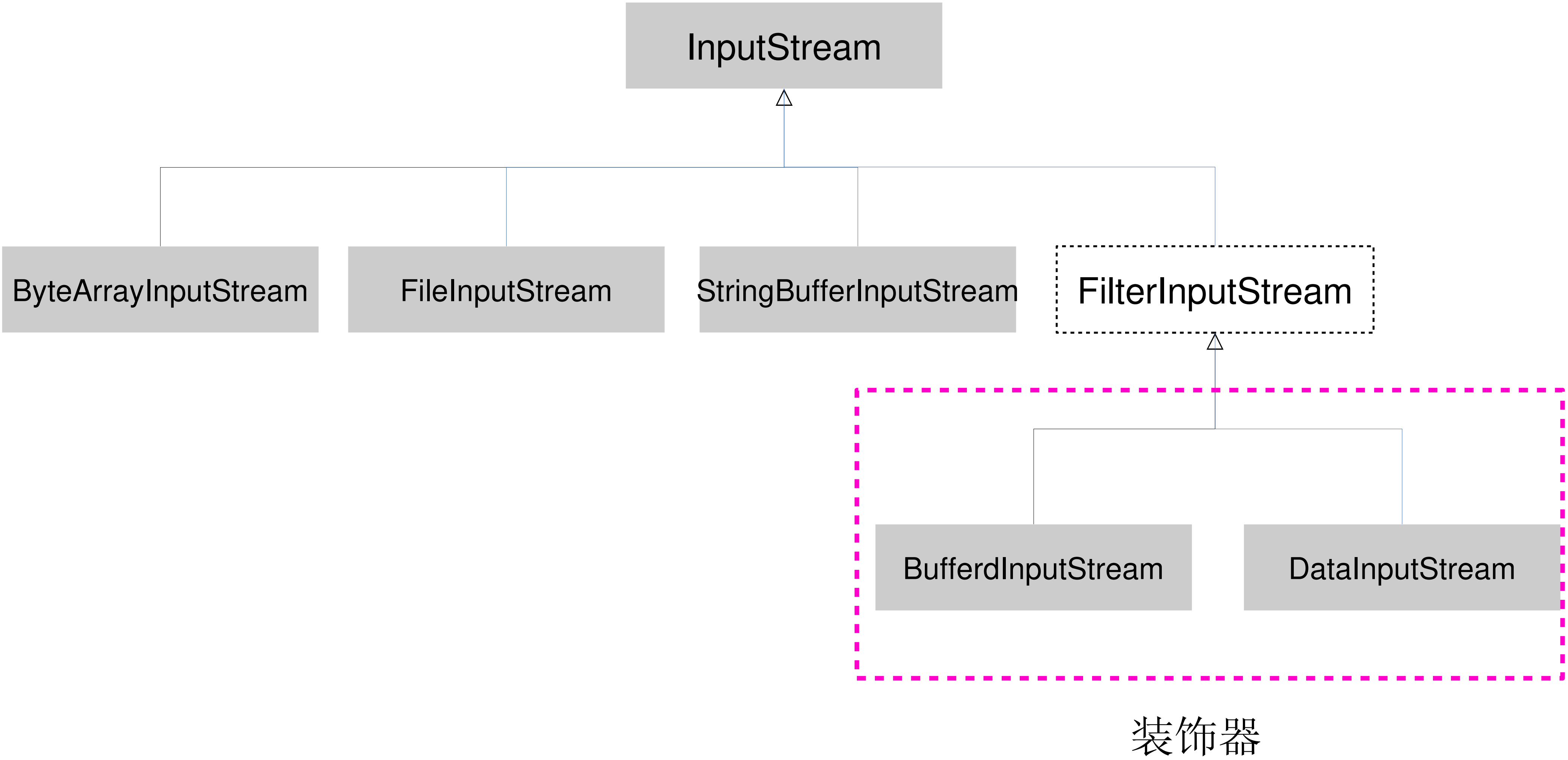
```
FileInputStream fin = new FileInputStream("xanadu.txt");
BufferedInputStream bf = new BufferedInputStream(fin);
DataInputStream din = new DataInputStream(bf);

din.read();  din.readInt();  din.readDouble();
```

```
FileOutputStream fout = new FileOutputStream("xanadu.txt");
BufferedOutputStream bf = new BufferedOutputStream(fout);
DataOutputStream dout = new DataOutputStream(bf);

dout.write(1);  dout.writeInt(10);  dout.writeDouble(3.14);
```

- 复习



InputStream
ByteArrayInputStream
FileInputStream
StringBufferInputStream
FilterInputStream
BufferdInputStream
DataInputStream
装饰器

OOP with Java

- 运行时类型信息
- 泛型

RTTI

- 运行时类型信息
  - RunTime Type Information (RTTI)
  - Class 类
    - 每一个类都包含一个 Class 类的对象
    - 该对象包含该类的信息
      - 类名字
      - 类的有哪些方法
      - 类的有哪些成员

RTTI

- 每个类的静态成员
  - 所有对象共享一个 Class 对象
- 每个对象调用 getClass() 获得 Class 类的对象
  - String s = “hello”; Class c = s.getClass();
- 每个类通过 .class 获得 Class 类的对象
  - Class c = String.class

RTTI

- Class 类的方法
  - getName()
  - getInterfaces()
  - getSuperclass()
  - newInstance()

```java
import static net.mindview.util.Print.*;
interface HasBatteries { }
interface Waterproof { }
interface Shoots {}
class Toy {
    Toy() {}
    Toy(int i) {}
}

class FancyToy extends Toy implements
                HasBatteries, Waterproof, Shoots {
        FancyToy() { super(1); }
}
```

```java
public class ToyTest {
    static void printInfo(Class cc) {
        print("Class name: " + cc.getName() +
            " is interface? [" + cc.isInterface() + "]");
        print("Simple name: " + cc.getSimpleName());
        print("Canonical name : " + cc.getCanonicalName());
    }
    public static void main(String[] args) {
        Class c = null;
        try { c = Class.forName("typeinfo.toys.FancyToy");
        } catch(ClassNotFoundException e) {
                print("Can't find FancyToy");System.exit(1);
        }
        printInfo(c);
        for(Class face : c.getInterfaces())
            printInfo(face);
        Class up = c.getSuperclass();
        Object obj = null;
        try {
        // Requires default constructor:
        obj = up.newInstance();
        } catch(InstantiationException e) {
            print("Cannot instantiate"); System.exit(1);
        } catch(IllegalAccessException e) {
            print("Cannot access"); System.exit(1);
        }
        printInfo(obj.getClass());
    }
}
```

RTTI

- RTTI 的用途
  - 给定 Object 引用，判断它的类型

```
String s = new String(“hello”);
Object o = s;
Class c = o.getClass().getName();
String t = (String)s;
```

泛型

- 容器
  - new ArrayList();

```
class Apple {
    private static long counter;
    private final long id = counter++;
    public long id() { return id; }
}
class Orange { }
```

```
public class ApplesAndOrangesWithoutGenerics {
public static void main(String[] args) {
    ArrayList apples = new ArrayList();
    for(int i = 0; i < 3; i++)
        apples.add(new Apple());

    // Not prevented from adding an Orange to apples:
    apples.add(new Orange());
    for(int i = 0; i < apples.size(); i++)
        ((Apple)apples.get(i)).id();
        // Orange is detected only at run time
    }
}
```

泛型

- 类型安全的容器
  - new ArrayList<Apple>();

```
public class ApplesAndOrangesWithGenerics {
public static void main(String[] args) {
    ArrayList<Apple> apples = new ArrayList<Apple>();
    for(int i = 0; i < 3; i++)
        apples.add(new Apple());

    // Compile error!
    // apples.add(new Orange());
    for(int i = 0; i < apples.size(); i++)
        apples.get(i).id();
    }
    for(Apple c: apples)
        System.out.println(c.id());
}
```

泛型

- 不同类型间的相互替代？
  - 多态

```
Class A { }



void f (A arg) {
    // …
}

f(new A());
```

```
Class A { }
Class B{}



void f (A arg) {
    // …
}

f(new B());
```

```
Class A { }
Class B{}
Class C extends A{}

void f (A arg) {
    // …
}

f(new C());
```

泛型

- 不同类型间的相互替代？
  - 泛型：更灵活的方式

基本概念

- 泛型 (generics)
  - 在定义类的成员和方法时，类型为可变参数

```
class MyType {
    ? member = null;
}
```

```
void f (? arg) {
    // …
}
```

基本概念

- 语法

```
class MyType {
    ? member = null;
}
```

```
class MyType<T> {
    T member = null;
}
```

```
void f (? arg) {
    // …
}
```

```
<T> void f (T arg) {
    // …
}
```

Example: A simple Holder class

```
class Apple { }

public class Holder Apple {
    private Apple a;
    public Holder(Apple a) { this.a = a; }
    public void set(Apple a) { this.a = a; }
    public Apple get() { return a; }

    public static void main(String[] args) {
        Holder h = new Holder(new Apple());
        Apple a = h.get();
        // h.set("Not an Apple"); // Error
        // h.set(1); // Error
    }
}
```

Example: A simple Holder class

```
class Apple { }

public class Holder<T> {
    private T a;
    public Holder(T a) { this.a = a; }
    public void set(T a) { this.a = a; }
    public T get() { return a; }

    public static void main(String[] args) {
        Holder<String> hs = new Holder<String>(new String(“hello”));
        String s = hs.get();

        Holder<Integer> hi = new Holder<Integer>(1);
        int I = hi.get();

        Holder<Apple> h = new Holder<Apple>(new Apple());
        Apple a = h.get();
        // h.set("Not an Apple"); // Error
        // h.set(1); // Error
    }
}
```

Example: A simple Holder class

```java
class Apple { }

public class Holder<T> {
    private T a;
    public Holder(T a) { this.a = a; }
    public void set(T a) { this.a = a; }
    public T get() { return a; }

    public static void main(String[] args) {
        Holder<Apple> h = new Holder<Apple>(
              new Apple());
        Apple a = h.get();
    }
}
```

```java
class Apple { }

public class Holder {
    private Object a;
    public Holder(Object a) { this.a = a; }
    public void set(Object a) { this.a = a; }
    public Object get() { return a; }

    public static void main(String[] args) {
        Holder h = new Holder(new Apple());
        Apple a = (Apple)h.get();
    }
}
```

基本概念

- 参数化类型
  - Much like the more familiar formal parameters used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types

Example: A tuple library

- Tuple
  - 不可变数组（数组元素不可变）
  - 用途：
    - 函数返回多个值
    - Data transfer object

Example: A tuple library

```
public class TwoTuple<A,B> {
    public final A first;
    public final B second;
    public TwoTuple(A a, B b) { first = a; second = b; }
    public String toString() {
        return "(" + first + ", " + second + ")";
    }
}
```

Example: A tuple library

```
public class ThreeTuple<A,B,C> extends TwoTuple<A,B> {
    public final C third;
    public ThreeTuple(A a, B b, C c) {
        super(a, b);
        third = c;
    }
    public String toString() {return "(" + first + ", " + second + ", " +
third +")"; }
}
```

```
public class FourTuple<A,B,C,D> extends ThreeTuple<A,B,C> {
    public final D four;
    public ThreeTuple(A a, B b, C c, D d) {
        super(a, b, c);
        third = d;
    }
    public String toString() {return "(" + first + ", " + second + ", " +
third + "," + four + ")"; }
}
```

Example: A tuple library

```
class Apple { }
class Orange { }

public class TupleTest {
    static TwoTuple<String,Integer> f() {
        return new TwoTuple<String,Integer>("hi", 47); // Autoboxing
    }
    static ThreeTuple<Apple,String,Integer> g() {
        return new ThreeTuple<Apple, String, Integer>(new Apple(), "hi", 47);
    }
    static FourTuple<Orange,Apple,String,Integer> h() {
        return new FourTuple<Orange,Apple,String,Integer>(new Orange(), new Apple(),
"hi", 47);
    }
    public static void main(String[] args) {
        TwoTuple<String,Integer> ttsi = f();
        System.out.println(ttsi);
        // ttsi.first = "there"; // Compile error: final
        System.out.println(g());
        System.out.println(h());
        }
}
```

泛型接口

- 带有类型参数的接口

```
public interface Generator<T> { T next(); }
```

```
public class Coffee {
    private static long counter = 0;
    private final long id = counter++;
    public String toString() {
        return getClass().getSimpleName() + " " + id;
    }
}

public class Latte extends Coffee {}
public class Mocha extends Coffee {}
public class Cappuccino extends Coffee {}
public class Americano extends Coffee {}
public class Breve extends Coffee {}
```

```java
import java.util.*;
public class CoffeeGenerator implements Generator<Coffee> {

    private Class[] types = { Latte.class, Mocha.class,
        Cappuccino.class, Americano.class, Breve.class, };

    private static Random rand = new Random(47);
    private int size = 0;
    public CoffeeGenerator() {}
    public CoffeeGenerator(int sz) { size = sz; }

    public Coffee next() {
        try {
            return (Coffee) types[rand.nextInt(types.length)].newInstance();
            // Report programmer errors at run time:
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
    }
    public static void main(String[] args) {
        CoffeeGenerator gen = new CoffeeGenerator();
        for(int i = 0; i < 5; i++)
            System.out.println(gen.next());
    }
}
```

```
// public class CoffeeGenerator implements Generator<Coffee>

public class Fibonacci implements Generator<Integer> {
    private int count = 0;
    public Integer next() { return fib(count++); }
    private int fib(int n) {
        if(n < 2) return 1;
        return fib(n-2) + fib(n-1);
    }
    public static void main(String[] args) {
        Fibonacci gen = new Fibonacci();
        for(int i = 0; i < 18; i++)
            System.out.print(gen.next() + " ");
    }
}
```

泛型方法

- 方法的参数，返回值带有类型参数
- 语法

```
public <T> void f(T x)  {  //… }
```

<T> 说明该方法带有类型参数 T

<T> 需要放在返回值说明之前

泛型方法

```
public class GenericMethods {
    public <T> void f(T x) {
        System.out.println(x.getClass().getName());
    }
    public static void main(String[] args) {
        GenericMethods gm = new GenericMethods();
        gm.f("");   gm.f(1);   gm.f(1.0); gm.f(1.0F);      gm.f('c');  gm.f(gm);
    }
}
```

方法的类型参数可以与类的类型参数无关

```
public class GenericMethods<E> {
    E member;
    public <T> void f(T x) {
        System.out.println(x.getClass().getName());
    }
}
```

类型推理

- 编译器推理出类型变量的值

```
public class New {
    public static <K, V>  Map<K, V>() {
        return new HashMap<K, V>();
    }
    public static void main(String []args){
        Map<String, List<String>> s = New.Map();
    }
}
```

1．由 Map<String, List<String>> s = New.Map();
推断出 Map() 方法中的 K=String，V=List<String>

2．可以通过语法明确泛型方法调用时的类型参数
Map<String, List<String>> s = New.<String, List<String>>Map();

```
// Diamond

ArrayList<String> s = new ArrayList<>();
HashMap<String, Integer> h = new HashMap<>();
```

小结

- 泛型类

```
public class TwoTuple<A, B> { //... }
```

- 泛型接口

```
public interface Generator<T> { T next(); }
```

- 泛型方法

```
public <T> void f(T x)  {  //… }
```

```
public class Generators {
    public static <T> Collection<T> fill(Collection<T> coll, Generator<T> gen, int n) {
        for(int i = 0; i < n; i++)
            coll.add(gen.next());
        return coll;
    }
    public static void main(String[] args) {
        Collection<Coffee> coffee = fill(new ArrayList<Coffee>(), new CoffeeGenerator(), 4);
        for(Coffee c : coffee)
            System.out.println(c);

        Collection<Integer> fnumbers = fill(new ArrayList<Integer>(), new Fibonacci(), 12);
        for(int i : fnumbers)
            System.out.print(i + ", ");
        }
}
```

Type Erasure

- 当给定不同的类型参数时，泛型的类型是否相同？

```
class A { }
class B extends A{ }

public class Test {
    public static void main(String []args){
        ArrayList<String> strlist = new ArrayList<>();
        ArrayList<Integer> intl = strlist;

        ArrayList<B> blist = new ArrayList<>();
        ArrayList<A> alist = blist;
    }
}
```

编译错误！
ArrayList<String>, ArrayList<Integer>, ArrayList<B>, ArrayList<A>
是不同类型！

Type Erasure

- 当给定不同的类型参数时，泛型的类型是否相同？
  - Class Object

```
import java.util.*;

public class ErasedTypeEquivalence {

    public static void main(String[] args) {
        Class c1 = new ArrayList<String>().getClass();
        Class c2 = new ArrayList<Integer>().getClass();
        System.out.println(c1 == c2);
    }
}
```

输出：**true**！

Type Erasure

```
import java.util.*;
class Frob { }
class Fnorkle { }
class Quark<Q> { }
class Particle<POSITION,MOMENTUM> { }

public class LostInformation {
    public static void main(String[] args) {
        List<Frob> list = new ArrayList<Frob>();
        Map<Frob,Fnorkle> map = new HashMap<Frob,Fnorkle>();
        Quark<Fnorkle> quark = new Quark<Fnorkle>();
        Particle<Long,Double> p = new Particle<Long,Double>();

        System.out.println(Arrays.toString(list.getClass().getTypeParameters()));
        System.out.println(Arrays.toString(map.getClass().getTypeParameters()));
        System.out.println(Arrays.toString(quark.getClass().getTypeParameters()));
        System.out.println(Arrays.toString(p.getClass().getTypeParameters()));
    }
}
```

输出：

[E]
[K, V]
[Q]
[POSITION, MOMENTUM]

Type Erasure

- 泛型是如何实现的？
- ArrayList<String> 与 ArrayList <Integer> 的区别在哪里？

Type Erasure

- 类型擦除 (Type Erasure),
  - T 仅作为占位符，不包含任何具体类型的信息
  - 所有 T 类型的对象引用最终实现为 Object 对象引用

Type Erasure

```
public class HasF {
    public void f() { System.out.println("HasF.f()"); }
}

class Manipulator<T> {
    private T obj;
    public Manipulator(T x) { obj = x; }
    // Error: cannot find symbol: method f():
    public void manipulate() { obj.f(); }
}

public class Manipulation {
    public static void main(String[] args) {
        HasF hf = new HasF();
        Manipulator<HasF> manipulator = new Manipulator<HasF>(hf);
        manipulator.manipulate();
    }
}
```

Type Erasure

```
public class Erased<T> {
    private final int SIZE = 100;
    public static void f(Object arg) {
        if(arg instanceof T) { }      // compile error
        T var = new T();              // compile error
        T[] array = new T[SIZE];  // compile error
    }
}
```

Type Erasure

- Type Erasure 的实现原理 1:
  - 所有引用最后都变为 Object 引用
  - 编译器自动添加 Downcasting 代码

```
class Apple { }

public class Holder<T> {
    private T a;
    public Holder(T a) { this.a = a; }
    public void set(T a) { this.a = a; }
    public T get() { return a; }

    public static void main(String[] args) {
        Holder<Apple> h = new Holder<Apple>(
            new Apple());
        Apple a = h.get();
    }
}
```

```
class Apple { }

public class Holder {
    private Object a;
    public Holder(Object a) { this.a = a; }
    public void set(Object a) { this.a = a; }
    public Object get() { return a; }

    public static void main(String[] args) {
        Holder h = new Holder(new Apple());
        Apple a = (Apple)h.get();
    }
```

生成相同的机器代码
编译器在 get() 返回处自动添加了类型转换代码

Type Erasure

- Type Erasure 的实现原理 2:
  - 编译器只做静态检查

```
class A { }
class B extends A{ }

public class Test {
    public static void main(String []args){
        ArrayList<String> strlist = new ArrayList<>();
        ArrayList<Integer> intl = strlist;

        ArrayList<B> blist = new ArrayList<>();
        ArrayList<A> alist = blist;
    }
}
```

1. 编译器通过静态检查报错 （ArrayList<String> 与 ArrayList<Integer> 类型不同）
2. 但实际存储中，无论 <String> 还是 <Integer> 都存储的是 Object 类型的引用

Type Erasure

- Type Erasure
  - 类型变量仅仅对编译器的静态检查有用
  - 当通过静态检查，所有类型参数被擦除

Type Erasure

- 如何解决 Type Erasure 带来的问题？
  - Class 对象

```java
public class Erased<T> {
    private final int SIZE = 100;
    public static void f() {
        T var = new T();     // compile error
    }
}
```

```java
public class TypeCapture<T> {
    Class<T> kind;
    public TypeCapture(Class<T> kind){ this.kind = kind; }
    public T f() {
        T r = null;
        try{ r = kind.newInstance(); }
        catch (Exception e){
            System.out.println("fail");
        }
        return r;
    }
    public static void main(String []args) {
        TypeCapture<String> c = new TypeCapture<>(String.class);
        System.out.println(c.f());
    }
}
```

Type Erasure

- 如何解决 Type Erasure 带来的问题？

```
public class HasF {
    public void f() { System.out.println("HasF.f()"); }
}
```

```
class Manipulator<T> {
    private T obj;
    public Manipulator(T x) { obj = x; }
    // Error: cannot find symbol: method f():
    public void manipulate() { obj.f(); }
}
```

```
class Manipulator<T extends HasF> {
    private T obj;
    public Manipulator(T x) { obj = x; }

    public void manipulate() { obj.f(); }
}
```

```
public class Manipulation {
    public static void main(String[] args) {
        HasF hf = new HasF();
        Manipulator<HasF> manipulator = new Manipulator<HasF>(hf);
        manipulator.manipulate();
    }
}
```

被限定的类型参数

- 限定类型参数的范围
  - 语法：

```
class A <T extends B>  { //... }
```

  - 表示类型参数 T 只能是 B 类型或者 B 的子类型
  - 作用：静态检查时，可以合法引用 T 的方法 （但最终仍然是 Object）
  - 多个限定类型

```
class A <T extends B & C>  { //... }
```

    - 如果有类和接口，类应该出现在第一个位置

被限定的类型参数

```
interface HasColor { java.awt.Color getColor(); }
class Colored<T extends HasColor> {
    T item;
    Colored(T item) { this.item = item; }
    T getItem() { return item; }
    // The bound allows you to call a method:
    java.awt.Color color() { return item.getColor(); }
}

class Dimension { public int x, y, z; }
// Multiple bounds:
class ColoredDimension<T extends Dimension & HasColor> {
    T item;
    ColoredDimension(T item) { this.item = item; }
    T getItem() { return item; }
    java.awt.Color color() { return item.getColor(); }
    int getX() { return item.x; }
    int getY() { return item.y; }
    int getZ() { return item.z; }
}
```

通配符

```
class Fruit { }
class Apple extends Fruit{ }
class Orange extends Fruit{ }

public class Test {
    public static void main(String []args){
        List<Apple> alist = new ArrayList<Apple>();
        // compile error
        // List<Fruit> flist = new ArrayList<Apple>();
    }
}
```

如何表达 ArrayList 中的元素是 Fruit 的某个子类？

通配符

```
class Fruit { }
class Apple extends Fruit{ }
class Orange extends Fruit{ }

public class Test {
    public static void main(String []args){
        List<Apple> alist = new ArrayList<Apple>();
        // compile error
        // List<Fruit> flist = new ArrayList<Apple>();
        List<? extends Fruit> flist = new ArrayList<Apple>();
        // Compile Error: can't add any type of object:
        // flist.add(new Apple());
        // flist.add(new Fruit());
        // flist.add(new Object());
        flist.add(null); // Legal but uninteresting
        // We know that it returns at least Fruit:
        Fruit f = flist.get(0);
    }
}
```

通配符

- 类型参数为某个 B 的子类型
- 具体是哪一个无法确定

```
<? extends B>
```

- List<? extends Apple>
  - 无法 add 任何对象
  - 可以 get 对象

泛型

- 类型参数
  - Holder, Tuple
- 泛型接口
- 泛型方法
- Erasure
- 被限定的类型参数
- 通配符

通配符

<? extends B>

- 类型参数为某个 B 的子类型
- 具体是哪一个无法确定
- List<? extends Fruit>
  - 无法 add 任何对象
    - List 中存储的对象类型无法确定
  - 可以 get 对象
    - Fruit 类型

通配符

<? super B>

- 类型参数为某个 B 的父类型
- 具体是哪一个无法确定
- List<? super Apple>
  - 无法 get 任何对象
    - 无法确定是哪个父类
  - 可以 add Apple 的子类对象

```java
import java.util.*;
public class SuperTypeWildcards {
    static void writeTo(List<? super Apple> apples) {
      apples.add(new Apple());
      apples.add(new Jonathan());
      // apples.add(new Fruit()); // Error
    }
}
```

总结

- Type Erasure
  - 类型变量仅仅对编译器的静态检查有用
  - 当通过静态检查，所有类型参数被擦除
- 被限制的类型参数

```
class A <T extends B & C>   { //... }
```

- 通配符

```
<? extends B>
```

```
<? super B>
```