



哈爾濱工業大學(深圳)  
HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

規格嚴格 功夫到家  
1920—2025

# 面向对象的软件构造导论

## 第十一章：泛型与反射



## 第九章（回顾）

---

- 进程与线程
- 多线程
- Java中对线程的控制
- 同步、死锁及如何避免
- 多线程应用——生产者与消费者模式
- 任务与线程池



# 生产者-消费者设计模式

## Java代码实现缓冲区

```
public class Buffer {  
    private List<Integer> data = new ArrayList<>();  
    private static final int MAX = 2;  
    private static final int MIN = 0;  
    public void put(int value){  
        while (true){  
            try { //模拟生产数据  
                Thread.sleep(500);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            synchronized (this){  
                //当容器满的时候, producer处于等待状态  
                while (data.size() == MAX){  
                    System.out.println("buffer is full,waiting ....");  
                    try {  
                        wait();  
                    }  
                }  
            }  
        }  
    }  
}
```

```
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}  
//没有满, 则继续produce  
System.out.println("producer--"+  
Thread.currentThread().getName()+"--put:" + value);  
data.add(value);  
//唤醒其他所有处于wait()的线程, 包括消费者和生产者  
notifyAll();  
}  
}
```



# 生产者-消费者设计模式

## Java代码实现缓冲区

```
public Integer take(){
    Integer val = 0;
    while (true){
        try { //模拟消费数据
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        synchronized (this){
            //如果容器中没有数据，consumer处于等待状态
            while (data.size() == MIN){
                System.out.println("buffer is empty,waiting ...");
                try {
                    wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

```
//如果有数据，继续consume
    val = data.remove(0);
    System.out.println("consumer--"+
        Thread.currentThread().getName()+"--take:" + val);
```

//唤醒其他所有处于wait()的线程，包括消费者和生产者

```
        notifyAll();
    }
}
}
```



# 生产者-消费者设计模式

Java代码实现  
生产者

```
public class Producer implements Runnable{  
    private Buffer buffer;  
    public Producer(Buffer buffer) {  
        this.buffer = buffer;  
    }  
    @Override  
    public void run() {  
        buffer.put(new Random().nextInt(100));  
    }  
}
```



# 生产者-消费者设计模式

Java代码实现  
消费者

```
public class Consumer implements Runnable{  
    private Buffer buffer;  
    public Consumer(Buffer buffer) {  
        this.buffer = buffer;  
    }  
    @Override  
    public void run() {  
        Integer val = buffer.take();  
    }  
}
```



# 生产者-消费者设计模式

---

- 问题:

- 为什么缓冲区的判断条件是 `while(condition)` 而不是 `if(condition)`?
- *Java*中要求`wait()`方法为什么要放在同步块中?

- 答案:

- 防止线程被错误的唤醒 (Spurious Wake-Up)
- 防止出现无法唤醒 (Lost Wake-Up)



# 线程同步

- 代码实现:

- Java程序使用synchronized关键字对一个对象进行加锁。

```
synchronized(lock) {  
    n = n + 1;  
}
```

- 改写之前的代码

```
class AddThread extends Thread {  
    public void run() {  
        for (int i=0; i<10000; i++) {  
            synchronized(Counter.lock) { // 获取锁  
                Counter.count += 1;  
            } // 释放锁  
        }  
    }  
}
```

```
class DecThread extends Thread {  
    public void run() {  
        for (int i=0; i<10000; i++) {  
            synchronized(Counter.lock) { // 获取锁  
                Counter.count -= 1;  
            } // 释放锁  
        }  
    }  
}
```





# 线程死锁

- 结果

**PerA: 我有蓝色钥匙请给我红色钥匙**

**PerA 锁住蓝色钥匙**

**PerB: 我有红色钥匙请给我蓝色钥匙**

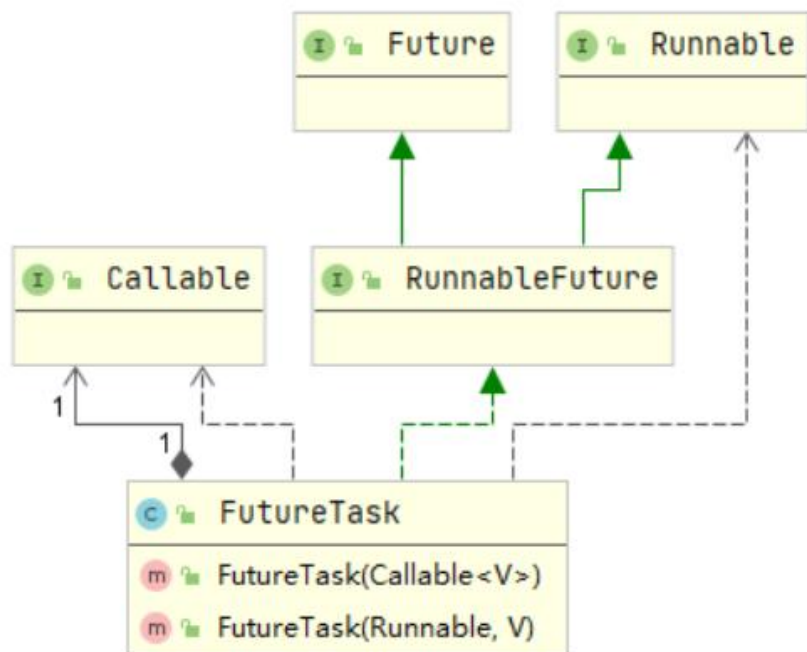
**PerB 锁住 红色钥匙**

- 死锁: 在线程A (PerA) 持有**蓝色钥匙**并想获得**红色钥匙**的同时, 线程B (Per) B持有**红色钥匙**并尝试获得**蓝色钥匙**, 那么这两个线程将永远地等待下去。
- 避免死锁: 线程获取锁的顺序要一致。即严格按照先获取蓝色钥匙, 再获取红色钥匙的顺序, 或者先获取红色钥匙, 再获取蓝色钥匙的顺序。

# FutureTask

- FutureTask: 执行 Callable 的一种方法

- 控制 Callable 任务的执行，获得其运算结果
- 间接实现了 Runnable 接口，进而可以通过 Thread 启动线程



```
// 实例化 Callable 任务，指定返回类型为 String
Callable<String> callable = new Callable<String>() {
    public String call() throws Exception {
        // balabalabala~
        return "Hello world ~";
    }
};

// 将 callable 任务委派给 FutureTask
FutureTask<String> task = new FutureTask<String>(callable);

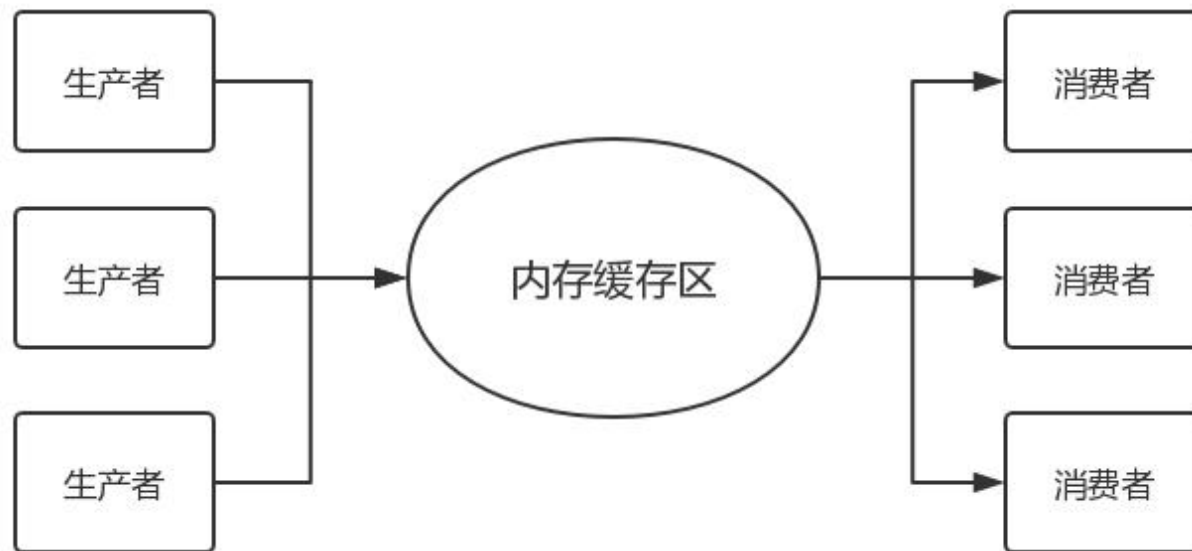
// FutureTask实现了Runnable接口，所以可以直接传给 Thread 启动线程
new Thread(task).start();

// 通过 FutureTask 获取返回值
String call = task.get();

System.out.println(call); // print: Hello world ~
```



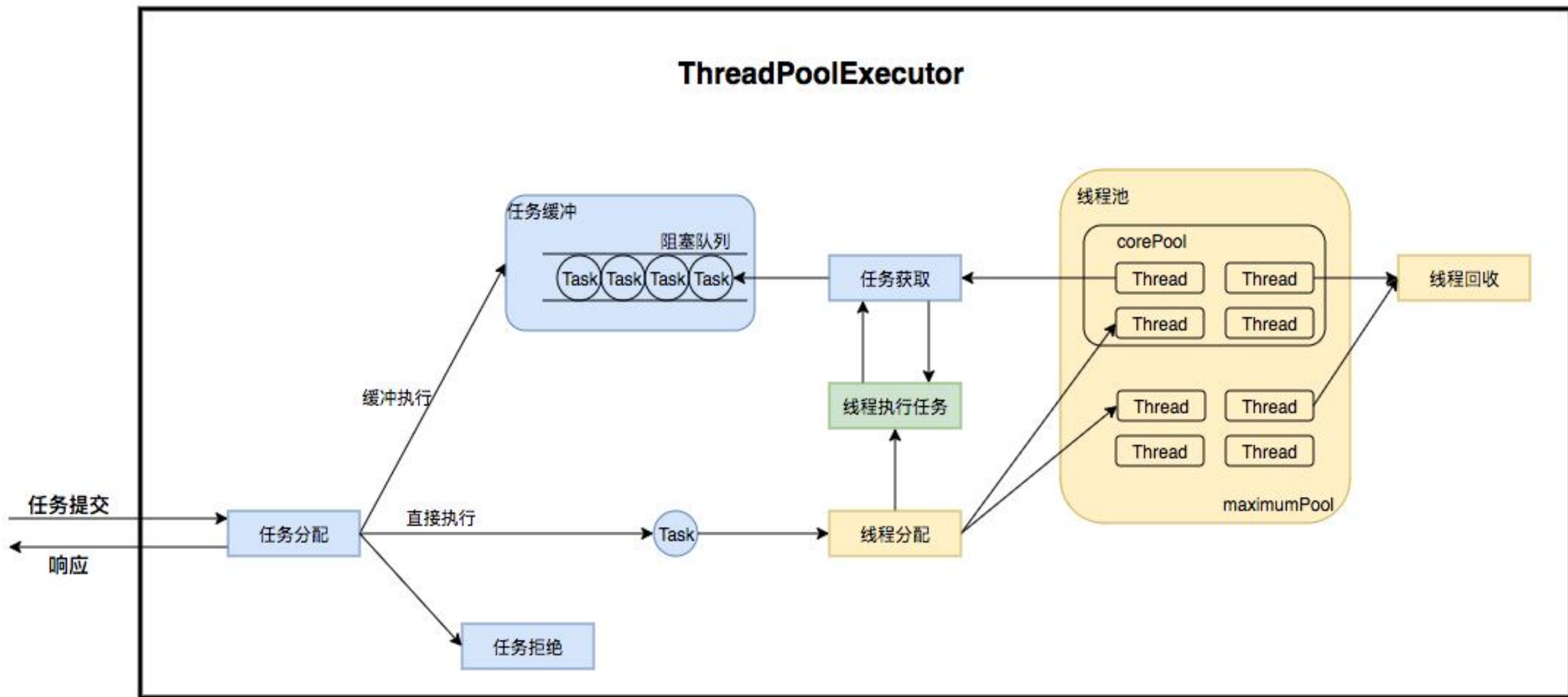
# 生产者-消费者设计模式



- 好处：
  - **并发**（异步）：生产者和消费者各司其职，生产者和消费者都只需要关心缓冲区，不需要互相关注，通过异步的方式支持高并发，将一个耗时的流程拆成生产和消费两个阶段。
  - **解耦**：生产者和消费者进行解耦（通过缓冲区通讯）。



# 线程池的多任务控制





# 泛型与反射（第十一章）

---

- 为什么需要泛型？什么是泛型？
- 泛型类、泛型方法、泛型接口
- 泛型的通配符
- 泛型的设计——模板方法模式
- 反射
- 设计安全的全局单例



## 课程导航

- 为什么需要泛型？什么是泛型？
- 泛型类、泛型方法、泛型接口
- 泛型的通配符
- 泛型的设计——模板方法模式
- 反射
- 设计安全的全局单例



# 为什么需要泛型

## □ 考虑以下代码存在问题

```
ArrayList list = new ArrayList();  
list.add("str");  
list.add(100);  
list.add(true);  
String name = (String) list.get(0);
```

1. 没有错误检查，可以向数组列表内添加任何对象。
2. 获取一个值时必须进行强制类型转换。



# 什么是泛型

- Java泛型是Java5中引入的一个新特性，提供了编译时类型安全监测机制，该机制允许我们在编译时检测到非法的数据结构。
  - 引入类型参数（type parameters），将所有操作的数据类型被指定为一个参数。

如指定泛型String

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

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

当调用get方法时，不需要进行强制类型转换，编译器就知道返回结果值类型为String，而不是Object。





# 什么是泛型

□ 考虑以下操作是否可行

```
ArrayList<String> list = new ArrayList<>();  
list.add(100);
```

- 不可行原因，编译器提示

```
Required type: String  
Provided: int
```

编译器知道List<String> 中add方法有一个类型为String的参数，相对使用Object类型更加安全。在编译阶段检测到非法的数据类型。



## 课程导航

- 为什么需要泛型？什么是泛型？
- 泛型类、泛型方法、泛型接口
- 泛型的通配符
- 泛型的设计——模板方法模式
- 反射
- 设计安全的全局单例



# 泛型类（1）

□ 泛型类（generic class）就是具有一个或多个类型参数的类。

- 语法格式：在类名之后声明泛型，泛型类先声明类型变量再使用。类型变量通常使用较短的大写，如 T，E，K，V等。

## 声明单个泛型

```
class Generic<T> {           //声明类的泛型 <T>
    public T t;              //变量t的类型为泛型T
    public Generic(){}
    public Generic(T t){     //传入参数的类型为T
        this.t = t;
    }
    public void fun1(T t) {}
    public T fun2(T t) {}    //函数返回值为类型T
}
```



## 泛型类（2）

▣ 类型参数（又称类型变量）作占位符，指示分配的类型

- E: (Element) 元素
- K: (Key) 键
- N: (Number) 数字
- T: (Type) 类型
- V: (Value) 值
- S、U、V 等：多参数情况中的第 2、3、4 个类型


```
public class Generic<T>{  
    .....  
}
```



## 泛型类 (3)

### 单个泛型类使用

```
class Generic<T> {  
    public T t;  
    public void fun(T t) {}  
}
```

```
Generic<String> g1 = new Generic<>();  
g1.t = "str";           //此处为合法数据类型  
g1.t = 100;              //此处为非法数据类型
```

```
Generic<Integer> g2 = new Generic<>();  
g2.t = 100;             //此处为合法数据类型
```



## 泛型类（4）

### 多个泛型类使用

```
class Generic<T, E> {  
    public T t;  
    public E e;  
    public void fun(T t, E e) {}  
}
```

```
Generic<String, Integer> g3 = new Generic<>();  
g3.t = "str";  
g3.e = 100;
```

- 注意，声明泛型不能使用基本数据类型，如int不行，需为Integer  
Generic<int> g = new Generic<>(); ❌  
Generic<Integer> g = new Generic<>();




## 泛型方法（1）

- 泛型方法可以定义在普通类中，也可以定义在泛型类中。泛型类中使用了泛型成员的方法不是泛型方法，只有声明泛型的方法才是泛型方法。

注意以下不是泛型方法

```
class Generic<T> {  
    public T t;  
    public void func(T t) {}  
}
```



注意以下是泛型方法

```
class Generic<T> {  
    public T t;  
    public <T> void func(T t) {}  
}
```



## 泛型方法（2）

非泛型类中定义泛型方法

```
class GenericFun{  
    public <T,E> void fun1(E e){}  
    public <T> T fun2(T t){  
        return t;  
    }  
}
```

- 调用泛型方法时，在方法名前的尖括号中填入具体类型。

```
GenericFun g = new GenericFun();
```

```
g.<String>fun2("str");
```

- 多数情况下，方法调用可以省略类型参数。

```
g.fun2("str"); //返回值为str
```





## 泛型方法 (3)

泛型类中定义泛型方法

```
class GenericFun<K>{  
    public <T> T fun2(T t, K k){  
        return null;  
    }  
}
```

- 使用泛型方法，此处传入参数必须与泛型类声明的类型一致

```
GenericFun<Integer> g = new GenericFun<>();  
g.fun2("str1", 123) ;
```

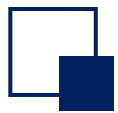


## 泛型方法（4）

- 如果泛型方法的泛型与泛型类声明的泛型名称一致，则泛型方法中的泛型会覆盖类的泛型。

```
class GenericFun<K>{  
    public <T,K> T fun2(T t, K k){  
        return null;  
    }  
}
```


- 使用泛型方法，此时类的泛型为Integer，泛型方法传入的参数为两个String  
`GenericFun<Integer> g = new GenericFun<>();`  
`g.fun2("str1","str2") ;`



## 泛型方法（5）

❑ 类的静态泛型方法，不得使用泛型类中声明的泛型，可以独立声明。

```
class GenericStaticMethod<K>{  
    private K k;  
    public GenericStaticMethod(K k){  
        this.k=k;  
    }  
    public static GenericStaticMethod <K> fun1(K k){  
        return new GenericStaticMethod <K>(k)}  
} //报错：无法从静态上下文中引用非静态类型变量K
```



```
public static <K> GenericStaticMethod<K> fun1(K k){  
    return new GenericStaticMethod<K>(k)}  
} //可以编译成功  
public static <T> GenericStaticMethod<T> fun1(T t){  
    return new GenericStaticMethod<T>(t)}  
} //写成另一种类型
```



# 泛型方法

// 定义类 Durian 时使用了泛型声明

```
class Durian<T> {
```

```
// 使用T类型形参定义实例属性
```

```
private T info;
```

```
// 下面方法中使用T类型形参来定义构造函数
```

```
public Durian(T info) { this.info = info; }
```

```
public void setInfo(T info) { this.info = info; }
```

```
public T getInfo() { return this.info; }
```

```
// 静态泛型方法应该使用其他类型区分：
```

```
public static <K> Durian<K> readyear(K info) {
```

```
    return new Durian<K>(info);
```

```
}
```

```
}
```



# 泛型方法

```
public class GenericDurian {  
    public static void main(String[] args) {  
        // 由于传给T形参的是String, 所以构造器参数只能是String/  
        Durian<String> a1 = new Durian<>("猫山王");//名字  
        System.out.println(a1.getInfo());  
        // 由于传给T形参的是Double, 所以构造器参数只能是Double  
        Durian<Double> a2 = new Durian<>(1.23);//重量  
        System.out.println(a2.getInfo());  
  
        Durian<Integer> a3 = Durian.readyear(2022);  
        System.out.println(a3.getInfo());  
    }  
}  
  
    public static <K> Durian<K> readyear (K info) {  
        return new Durian<K>(info);  
    }
```



## 泛型接口 (1)

定义以下泛型接口

```
interface GenericInterface <T>{  
    T fun1();  
}
```

实现类非泛型类，需具体指定接口的泛型

```
class GenericInterfaceImpl implements GenericInterface<String> {  
    @Override  
    public String fun1() {  
        return null;  
    }  
}
```




## 泛型接口 (2)

定义以下泛型接口

```
interface GenericInterface <T>{  
    T fun1();  
}
```

实现类为泛型类，实现类的泛型要与接口一致

```
class GenericInterfaceImpl<T> implements GenericInterface<T> {  
    @Override  
    public T fun1() {  
        return null;  
    }  
}
```





# 泛型接口

```
interface ShowInterface<T> { public void show(T t); }
```

//实现类是非泛型类，已确定类型

```
class ShowClass1 implements ShowInterface<String>{  
    public void show(String t){  
        System.out.println("show:"+t);  
    }  
}
```

//实现类是泛型类，未确定类型

```
Class ShowClass2<T> implements ShowInterface<T>{  
    public void show(T t){  
        System.out.println("show:"+t);  
    }  
}
```





# 泛型接口

```
public static void main(String[] args) {  
    //实现类已确定类型  
    ShowClass1 obj = new ShowClass1();  
    obj.show("java");  
  
    //实现类未确定类型，使用时确定  
    ShowClass2<Integer> obj = new ShowClass2<>();  
    obj.show(6);  
}
```



## 课程导航

- 为什么需要泛型？什么是泛型？
- 泛型类、泛型方法、泛型接口
- 泛型的通配符
- 泛型的设计——模板方法模式
- 反射
- 设计安全的全局单例



# 泛型的通配符

---

- ❑ 严格的泛型类型系统一旦指定便无法改变
- ❑ 需要允许类型参数发生变化
- ❑ 同时需要控制可以指定的类型，而非不加限制



# 泛型的通配符

## ❑ 为什么需要通配符

- 考虑为所有List抽象一个方法，不论给的参数是List<Integer>，List<String>，都可以接收并且打印List中的元素

```
public static void printAllObject(List<Object> list) {  
    for (Object object : list) {  
        System.out.println(object);  
    }  
}  
  
public static void main(String[] args) {  
    ArrayList<String> list1 = new ArrayList<>();  
    list1.add("java");  
    printAllObject(list1);  
}
```

Object是所有类型的父类，  
但是List<Object>并不是  
List<String>的父类

报错: printAllObject(List<Object>) is not applicable for the arguments(List<String>)



# 泛型的通配符

```
public static void printAllObject(List<?> list) {  
    for (Object object : list) {  
        System.out.println(object);  
    }  
}  
  
public static void main(String[] args) {  
    List<String> list1 = new ArrayList<>();  
    list1.add("java");  
    printAllObject(list1);  
    List<Integer> list2 = new ArrayList<>();  
    list2.add(7);  
    printAllObject(list2);  
}
```



# 泛型的通配符

## □ 通配符

- 允许类型参数发生变化
- `<? extends ClassName>`: 类型参数是`ClassName`的 子类
- `<? super ClassName>`: 类型参数是`ClassName`的 超类
- `<?>`: 无限定通配符



wildcard



# 上界通配符

## □<? extends T>

- 上界通配符实例化的类必须是T类，或是T类的子类

```
public class WildCardExtendsDemo {  
    public static void printAllObject(ArrayList<? extends Number> list) {  
        for (Object object : list) {  
            System.out.println(object);  
        }  
    }  
    public static void main(String[] args) {  
        ArrayList<Double> list1 = new ArrayList<>();  
        list1.add(1.23);  
        printAllObject(list1);  
    }  
}
```



# 下界通配符

## □<? super T>

- 上界通配符实例化的类必须是T类，或是T类的超类

```
public class WildCardSuperDemo {  
    public static void printAllObject(ArrayList<? super Double> list) {  
        for (Object object : list) {  
            System.out.println(object);  
        }  
    }  
    public static void main(String[] args) {  
        ArrayList<Number> list1 = new ArrayList<>();  
        list1.add(7);  
        printAllObject(list1);  
    }  
}
```





# 类型参数T与通配符？

□ T表示一个确定的类型

- 常用于泛型类和泛型方法的定义

□ ?表示不确定的类型，不是类型变量

- 通常用于泛型方法的调用代码和形参，不能用于定义类和泛型方法。

```
T t = operate();
```

```
? a = operate(); ❌ //非法
```



## 课程导航

- 为什么需要泛型？什么是泛型？
- 泛型类、泛型方法、泛型接口
- 泛型的通配符
- 泛型的设计——模板方法模式
- 反射
- 设计安全的全局单例



# 模板方法模式

---

## □ 模板方法模式（Template Method）

- 定义一个操作中的 **算法骨架**
- 将一些步骤 **延迟到子类** 中
- 模板方法模式使得子类可以 **不改变** 一个算法的结构即可 **重新定义** 该算法的某些 **特定步骤**



# 模板方法模式

## □ AbstractClass 抽象类

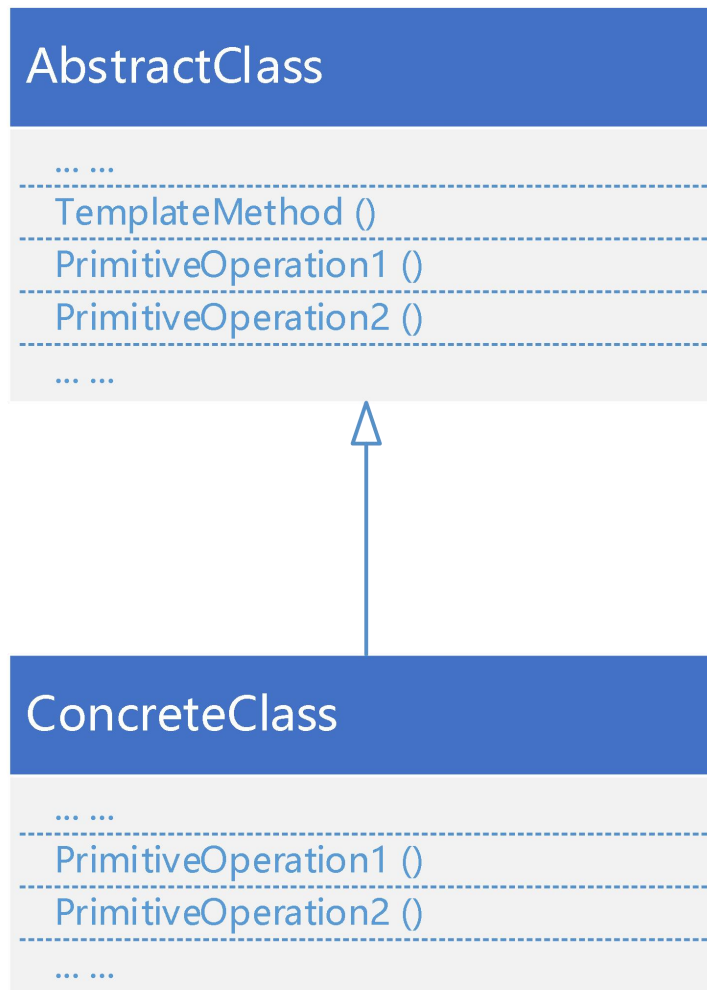
- 抽象模板，定义并实现了一个模板方法
- 给出顶层逻辑的骨架


## □ ConcreteClass 具体类

- 实现父类定义的一个或多个抽象方法

## □ 一个抽象类可以有任意多个具体子类

## □ 每一个抽象类都可以给出抽象方法的不同实现





```
public abstract class AbstractClass
{
    public abstract void PrimitiveOperation1();
    public abstract void PrimitiveOperation2();

    public void TemplateMethod()
    {
        PrimitiveOperation1();
        PrimitiveOperation2();
        System.out.println("");
    }
}
```

一些抽象行为放到子类实现

模板方法，给出了逻辑的骨架，而逻辑是由一些相应的抽象操作组成，它们都推迟到子类实现



```
public class ConcreteClassA extends AbstractClass
{
    public void PrimitiveOperation1()
    {
        System.out.println("具体类A方法1实现");
    }
    public void PrimitiveOperation2()
    {
        System.out.println("具体类A方法2实现");
    }
}

public class ConcreteClassB extends AbstractClass
{
    public void PrimitiveOperation1()
    {
        System.out.println("具体类B方法1实现");
    }
    public void PrimitiveOperation2()
    {
        System.out.println("具体类B方法2实现");
    }
}
```



# 模板方法模式

- 把 **不变行为** 搬到 **超类**，去除子类中的重复代码
- 提高了代码复用
- 应用情况
  - 一次性实现一个算法的 **不变部分**，并将 **可变的** 部分留给 **子类** 来实现
  - 各子类中 **公共的行为** 应被 **提取** 出来并 **集中** 到一个 **公共父类** 中以避免代码重复
  - 控制子类 **拓展**：模板方法只在特定点调用“hook”操作  
(hook operation: 提供 **缺省** 的行为，子类可以在 **必要时** 进行拓展)



# Hook operation (由父类定义)

```
abstract class PaymentProcessor {  
    // 模板方法 (固定流程)  
    public final void process() {  
        validate();        // 钩子: 验证支付  
        deductMoney();     // 固定步骤  
        notifyUser();      // 钩子: 通知用户  
    }  
  
    // 钩子1: 默认验证逻辑 (子类可覆盖)  
    protected void validate() {  
        checkBasicInfo(); // 基础校验  
    }  
  
    // 钩子2: 空方法 (子类必须重写才有意义)  
    protected abstract void notifyUser();  
  
    private void deductMoney() { ... } // 固定步骤  
}
```

```
class WeChatPayment extends PaymentProcessor {  
    // ✅ 这是钩子重写: 实现父类预留的扩展点  
    @Override  
    protected void notifyUser() {  
        sendWeChatMessage(); // 微信专属通知  
    }  
  
    // ✅ 这也是钩子重写: 增强默认验证  
    @Override  
    protected void validate() {  
        super.validate(); // 保留父类基础校验  
        checkWeChatLimit(); // 增加微信限额校验  
    }  
  
    // ❌ 普通重写 (破坏性!)  
    @Override  
    public void deductMoney() {  
        // 父类未开放此方法作为钩子 (应声明为final)  
        // 重写可能导致资金扣除逻辑出错!  
    }  
}
```

protected vs private





# 模板方法模式

## □ 参与者

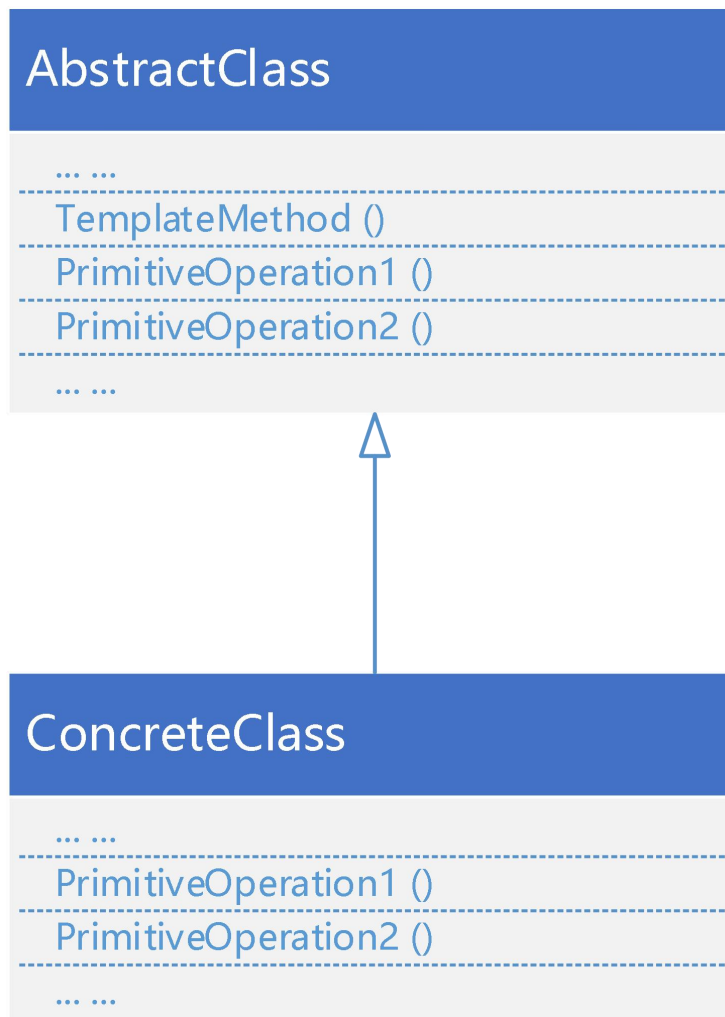
- 抽象类和具体类

## □ 协作

- 具体类靠抽象类实现算法中的不变步骤
- 抽象类靠具体类实现算法的具体细节

## □ 效果

- 模板方式方法在类库中尤为重要，他们提取了类库中的公共行为





# 模板方法模式

## □ 模板方法导致一种反向控制结构

- 指的是父类调用一个子类的操作，而不是相反
- 有时被称为“好莱坞法则”，即“别找我们。我们找你”

## □ 注意

- 模板方法必须指明那些操作是钩子操作（可被重新定义）以及哪些是抽象操作（必须被重新定义）
- 尽量减少一个子类具体实现该算法时必须重定义的那些抽象操作的数目



## 课程导航

- 为什么需要泛型？什么是泛型？
- 泛型类、泛型方法、泛型接口
- 泛型的通配符
- 泛型的设计——模板方法模式
- 反射
- 设计安全的全局单例



# 为什么需要反射

如果有成百上千个不同的 X 的实现类需要创建  
需要写上千个 if 语句来返回不同的 X 对象?

```
public class Test {  
    interface X {  
        public void test();  
    }  
    class A implements X {  
        @Override  
        public void test() {  
            System.out.println("I am A");  
        }  
    }  
    class B implements X {  
        @Override  
        public void test() {  
            System.out.println("I am B");  
        }  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        X a = create1("A");  
        a.test();  
        X b = create1("B");  
        b.test();  
    }  
  
    public static X create1(String name){  
        if (name.equals("A")) {  
            return new A();  
        }  
        else if(name.equals("B")) {  
            return new B();  
        }  
        return null;  
    }  
}
```



# 反射

```
// 上例中create1可以改写为
public static X create2(String name) {
    Class<?> class = Class.forName(name);
    X x = (X) class.newInstance();
    return x;
}
```

## □ 什么时候需要反射

- 在运行时 **获知** 任意一个对象 **所属的类**。
- 在运行时 **构造** 任意一个 **类的对象**。
- 在运行时 **获知** 任意一个类所具有的 **成员变量和方**
- 在运行时 **调用** 任意一个对象的 **方法和属性**。

getClass()

forname()

getDeclaredFields()

getMethod(), invoke()



# Class类

---

- ❑ Java运行时系统始终为所有对象维护一个运行时类型标识符
- ❑ 会跟踪每个对象所属的类的完整结构信息，包括包名、类名、实现的接口、拥有的方法和字段等，JVM利用这些信息选择要执行的正确方法
- ❑ 可以使用特殊的Java类访问这信息
- ❑ 对Class类的理解
  - 可以把Class类理解为类的类型
  - 一个Class对象称为类的类型对象

# Class类

```
import java.util.Date; // 先有类
public class Test {
    public static void main(String[] args) {
        Date date = new Date(); // 后有对象
        System.out.println(date);
    }
}
```

2. JVM从本地磁盘查找  
Date.class文件，并  
加载到JVM到内存中

Date date = new Date

1. 当我们调用new Date  
的时候，JVM会加载  
Date.class

Date.class

JVM

Date对象空间

3. 将.class文件读入内存

Class对象  
(存储Date类的信息)

3. 创建一个Class对象

一个类只产生一个Class对象



# Class类

## □ 正常方式

```
Date date = new Date();
```

引入的包类名称

new实例化

实例化对象

## □ 反射方式

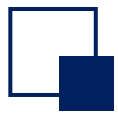
实例化对象

getClass()方法

完整的包类名称

```
System.out.println(date.getClass()); // "class java.util.Date"
```





# 获取Class类对象（1）

## □getClass() 方法

- Object类中的getClass()方法会返回一个Class类型的实例

```
public class getClassTest {  
    public static void main(String[] args) {  
        Student Harry = new Student("Harry Potter",11); // Student类描述学生  
        System.out.println(Harry.getClass()); // Class对象描述一个特定类的属性  
        // 输出 Class Student  
        System.out.println(Harry.getClass().getName()); //Class.getName() 返回类的名字  
        // 输出 Student  
    }  
}
```



## 获取Class类对象（2）

### □ Class.forName() 方法

- 将类名保存在字符串中

### □ T.class

- T是任意的Java类型（可能是类也可能不是类）

```
public class forNameTest {  
    public static void main(String[] args) throws ClassNotFoundException {  
        String className = "Reflect.Student";  
        Class cl1 = Class.forName(className); //Class.forName  
        Class cl2 = Student.class; //T.class  
    }  
}
```



# 通过反射构造类的实例

## □ 方法一：使用 `Class.newInstance`

- `newInstance` 方法调用默认的构造函数(无参)初始化新创建的对象
- 如果这个类没有默认的构造函数，就会抛出一个异常

类没有无参构造函数、  
类构造器是 **private** 时怎么办？

```
Date date1 = new Date();  
Class dateClass2 = date1.getClass();  
Date date2 = dateClass2.newInstance();
```



# 通过反射构造类的实例

---

## ❑ 方法二：使用 `Constructor` 的 `newInstance`

- 通过反射先获取构造方法再调用
- 先获取构造函数，再执行构造函数

## ❑ 区别

- `Constructor.newInstance` 是可以携带参数的
- `Class.newInstance` 是无参的



# 通过反射构造类的实例

## □ 获取构造函数

- 获取 **所有** "公有的" 构造方法
  - `public Constructor[] getConstructors() { }`
- 获取 **所有** 的构造方法（包括私有、受保护、默认、公有）
  - `public Constructor[] getDeclaredConstructors() { }`
- 获取 **一个指定** 参数类型的 "公有的" 构造方法
  - `public Constructor getConstructor(Class... parameterTypes) { }`
- 获取 **一个指定** 参数类型的 "构造方法"，可以是私有的，或受保护、默认、公有
  - `public Constructor getDeclaredConstructor(Class... parameterTypes) { }`



# 通过反射构造类的实例

## □使用Constructor的newInstance构造类的实例

```
public class ConTest {  
    public static void main(String[] args) {  
        Student Harry = new Student("Harry Potter", 11);  
        Class StudentClass = Harry.getClass();  
        Constructor con = StudentClass.getConstructor(String.class, int.class);  
        Student Ron = (Student)con.newInstance("Ron Weasley", 11);  
        System.out.println(Ron);  
    }  
}
```



# 通过反射获取和修改成员变量

## □ 获取和修改成员变量

- 获取所有公有的字段
  - `public Field[] getFields() { }`
- 获取所有的字段（包括私有、受保护、默认的）
  - `public Field[] getDeclaredFields() { }`
- 获取一个指定名称的公有的字段
  - `public Field getField(String name) { }`
- 获取一个指定名称的字段，可以是私有、受保护、默认的
  - `public Field getDeclaredField(String name) { }`
- 使用Field类中的get方法查看字段值
- 使用Field类中的set方法修改字段值



# 通过反射获取和修改成员变量

```
public class FieldTest {  
    public static void main(String[] args) {  
        Student Harry = new Student("Harry Potter", 11);  
        Class StudentClass = Harry.getClass();  
        Field f = StudentClass.getDeclaredField("name");  
        f.setAccessible(true);  
        Object v1 = f.get(Harry);  
        System.out.println(v1);  
        f.set(Harry, "The boy who lived");  
        System.out.println(Harry.getName());  
    }  
}
```

// 调用f.set(Harry, newValue)可以修改harry对象的name属性为newValue中的值





# 通过反射获取成员方法

## □ 通过反射获取成员方法

- 获取 **所有** "公有方法" (包含父类的方法, 当然也包含 **Object** 类)
  - `public Method[] getMethods() {}`
- 获取 **所有** 的成员方法, 包括私有的 (不包括继承的)
  - `public Method[] getDeclaredMethods() {}`
- 获取 **一个指定** 方法名和参数类型的成员方法
  - `public Method getMethod(String name, Class<?>... parameterTypes)`



# 调用反射获取的成员方法

## Object invoke(Object obj, Object... args)

- 第一个参数是哪个对象要来调用这个方法
- 第一个参数是调用方法时所传递的实参
- 对于静态方法，第一个参数可以忽略，即可设为null

```
public class MethodTest {  
    public static void main(String[] args) {  
        Student Harry = new Student("Harry Potter", 11);  
        Method m = Student.class.getMethod("getName");  
        String n = (String) m.invoke(Harry);  
    }  
}
```



# 反射

---

## □ 优点

- 比较灵活，能够在运行时动态获取类的实例

## □ 缺点

- 性能瓶颈：反射相当于一系列解释操作，通知JVM要做的事情，性能比直接的Java代码要慢很多
- 安全问题：反射机制破坏了封装性，因为通过反射可以获取并调用类的私有方法和字段。



## 课程导航

- 为什么需要泛型？什么是泛型？
- 泛型类、泛型方法、泛型接口
- 泛型的通配符
- 泛型的设计——模板方法模式
- 反射
- 设计安全的全局单例



# 单例

□ 单例的目的是保证某个类**仅有一个实例**

- 在类被加载时就实例化一个对象

```
public class Singleton {  
    private static Singleton singleton = new Singleton();  
    private Singleton(){}  
  
    public static Singleton getInstance(){  
        return singleton;  
    }  
}
```



# 利用反射破坏单例

❑通过反射中的`setAccessible(true)`覆盖Java中的访问控制，进而调用私有的构造函数

```
public class SingletonTest {  
    public static void main(String[] args) {  
        Class objectClass = Singleton.class;  
        Constructor constructor = objectClass.getDeclaredConstructor();  
        constructor.setAccessible(true);  
        Singleton instance = Singleton.getInstance();  
        Singleton newInstance = (Singleton)constructor.newInstance();  
        System.out.println(instance == newInstance);  
    }  
}  
} // instance和newInstance是不同的实例 False
```



# 抵御反射破坏

```
public class Singleton {  
    private static Singleton singleton = new Singleton();  
    private Singleton(){  
        if(singleton != null) {  
            throw new RuntimeException("单例构造器禁止通过反射调用");  
        }  
    }  
    public static Singleton getInstance(){  
        return singleton;  
    }  
}
```