دانشکده ریاضی و علوم کامپیوتر

برنامه‌سازی پیشرفته و کارگاه

Generics

استاد درس

دکتر مهدی قطعی

استاد دوم

بهنام یوسفی مهر

نگارش

سام قربانی، پریا اصحابی و مهدی جعفری

بهار ۱۴۰۳

فهرست

مقدمه 3

چرا به Generics احتیاج داریم؟ 4

چجوری می‌تونیم با Genericها کار کنیم؟ 7

متدهای جنریک 9

مفهوم Bounding (Upper-Bound) و محدود سازی در Generics 11

Wildcard چیه و چرا و چطوری استفاده میشه؟ 13

چرا از wildcard ها استفاده می‌کنیم و چرا List<Object> کافی نیست؟ 14

انواع Wildcard 17

چگونه Wildcard ها به مدیریت پیچیدگی کد کمک می‌کنند؟ 18

مقدمه

در برنامه‌نویسی با جاوا، حتماً براتون پیش اومده که بخواین با انواع مختلف داده‌ها کار کنین و نخواین واسه هر نوع، یه کلاس یا متد جدا بنویسین. اینجاست که ویژگی‌ای به نام **Generics** حسابی به کار میاد . Generics از اون قابلیت‌هاییه که هم کدتون رو مرتب‌تر و قابل‌فهم‌تر می‌کنه، هم کمک می‌کنه موقع اجرا با خطاهای عجیب و غریب روبه‌رو نشین. توی این داک قراره ببینیم اصلاً چرا به Generics نیاز داریم، چه مشکلی رو حل می‌کنه، و چطوری می‌تونیم باهاش کدی بنویسیم که هم امن‌تر باشه، هم قابل استفاده مجدد.

چرا به Generics احتیاج داریم؟

فرض کنید می‌خوایم یک کلاس Pair داشته باشیم که دو مقدار رو نگه داره، مثلاً یه String و یه int.

```
public class PairStringInteger {

    private String first;
    private int second;

    public PairStringInteger(String first, int second) {
        this.first = first;
        this.second = second;

    }
    // getters and setters
}
```

حالا اگه بخوایم یه pair از یه ترکیب دوتایی دیگه از نوع داده‌ها مثل int و double داشته باشیم چی؟ میتونیم دوباره یه همچین کلاسی بسازیم ولی این سری PairDoubleInteger باشه:

```
public class PairDoubleInteger {

    private double first;
    private int second;

    public PairDoubleInteger(double first, int second) {
        this.first = first;
        this.second = second;

    }
    // getters and setters
}
```

احتمالا تا همین الان هم متوجه شده باشید که داریم کار بیهوده‌ای میکنیم. اگه بخوایم همه ترکیب‌های ممکن رو پوشش بدیم، تعداد کلاس‌ها خیلی زیاد می‌شه و نگهداریش واقعاً کابوسه و همونطور که میدونید، ما هیچ علاقه‌ای به کدهای تکراری نداریم.

همون‌طور که از جلسه‌ی مربوط به شیءگرایی و ارث‌بری به یاد دارید، هر کلاسی که توی جاوا تعریف می‌شه، به‌نوعی (چه به‌صورت مستقیم، چه غیرمستقیم) از کلاس Object ارث می‌بره. همچنین، با توجه به مفهومی که از inheritance یاد گرفتید، می‌دونید که اشیاء تا حدی قابل جایگزینی هستن؛ یعنی اگه یه آبجکت از یه کلاس دیگه ارث برده باشه، می‌تونه به‌جای آبجکت والدش استفاده بشه.

حالا با در نظر گرفتن اینکه همه‌ی کلاس‌ها در نهایت فرزند java.lang.Object هستن، چرا از Object استفاده نکنیم که بتونیم هر چیزی رو نگه داریم؟

```
public class Pair {

    private Object first;
    private Object second;

    public Pair(Object first, Object second) {
        this.first = first;
        this.second = second;

    }
    public Object getFirst() {
        return first;
    }
    public Object getSecond() {
        return second;
    }
}
```

خب الان فقط یه کلاس داریم که هر چیزی رو می‌تونه نگه داره. ولی اینجا یه مشکل بزرگ داریم: type safety از بین رفته. مثلاً وقتی می‌خوایم از این کلاس استفاده کنیم:

```
Pair p = new Pair("age", 25);
String label = (String) p.getFirst();
Integer value = (Integer) p.getSecond();
```

تا اینجا مشکلی نیست چون درست type cast کردیم. ولی اگه اشتباه کنیم چی؟

```
Pair p = new Pair("age", 25);
String label = (String) p.getSecond(); // Runtime error!
```

کامپایلر نمی‌تونه جلوی این اشتباه رو بگیره چون همه چی از نوع Object هست. پس خطای **runtime** می‌گیریم، که یعنی ممکنه برنامه تو اجرا بترکه (!) بدون اینکه خطای کامپایل جلوش رو بگیره.

ماجرا یه جورایی مثل یه مهمونی بالماسکه‌ست که توش همه‌ی آبجکت‌ها یه ماسک یک شکل زدن و بین جمعیت گم می‌شن. وقتی آبجکت‌ها به صورت نوع Object در میان، کامپایلر دیگه نمی‌تونه تشخیص بده که واقعاً هر کدوم از چه نوعی بودن و ردشون رو گم می‌کنه. حالا کاربر باید خودش بعداً با **type cast** کردن، این نقاب رو برداره و بفهمه پشت اون ماسک چی بوده. مثل وقتی که می‌خوای ریش مصنوعی یه نفر رو تو مهمونی بکَنی. اگه اشتباه کنی، ممکنه با یه سورپرایز ناخوشایند روبه‌رو بشی :)

خیلی از این cast کردن‌ها ممکنه منجر به خطاهایی بشن که موقع کامپایل قابل شناسایی نیستن و این چیزی نیست که ما دنبالش باشیم.

خب قاعدتا تو همچین شرایطی، باید دنبال راه حل باشیم. راه‌حل درست، تمیز و اصولی، استفاده از جنریک‌هاست. با این کار، هم فقط یه کلاس داریم، هم کامپایلر حواسش به نوع‌ها هست تا دیگه به runtime error بر نخوریم. نه تنها type safety حفظ میشه، بلکه دیگه نیازی به تبدیل (casting) هم نیست.

یکی از مزایای generics اینه که می‌تونید کدهای عمومی‌تر و قابل استفاده مجدد بنویسید. به جای نوشتن کدهای تکراری برای انواع مختلف داده‌ها، می‌تونید یک کلاس یا متد عمومی بنویسید که بتونه با انواع مختلف کار کنه. به این ترتیب، دیگه نیازی به نوشتن کد جداگانه برای هر نوع داده ندارید.

حالا لازم داریم ببینیم که چجوری میتونیم با جنریک‌ها کار کنیم.

چجوری می‌تونیم با Genericها کار کنیم؟

گفتیم که جنریک‌ها به ما کمک می‌کنن تا کلاس‌ها رو برای نوع خاصی از داده‌ها تخصصی کنیم. یعنی یه کلاس جنریک می‌تونه با یه یا چند type parameter تعریف بشه و خودش رو بر اساس اون‌ها سفارشی کنه.

مثلاً اگه به کلاس Box تو مثال پایین نگاه کنیم، چیزی شبیه این می‌بینیم:

```
public class Box<T> {
    private T value;

    public void set(T value) {
        this.value = value;
    }

    public T get() {
        return value;
    }
}
```

اینجا Eداخل <> یک type parameter هست. یعنی List یک کلاس جنریکه که برای کامل بودنش باید نوع مشخصی بهش داده بشه. در این مثال، E یعنی نوع عناصری که قراره توی لیست ذخیره بشن.

داخل کلاس، از E برای تعریف متغیرها، پارامترهای متد، و نوع بازگشتی متدها، مثل یه نوع واقعی استفاده می‌شه. مثلاً متد ()add یه مقدار از نوع E می‌گیره و ()get هم یه Eبرمی‌گردونه.

برای استفاده از این کلاس، باید نوع واقعی رو جای Eمشخص کنیم، مثلا:

```
Box<String> myBox = new Box<>();
```

اینجا یه باکس از Stringها تعریف کردیم. ولی می‌تونستیم هر نوع شی دیگه‌ای هم بدیم.

حالا اگه کد زیر را در Main ران کنیم:

```
public class Main {
    public static void main(String[] args) {
        Box<String> stringBox = new Box<>();
        stringBox.set("salam");
        System.out.println(stringBox.get());
        Box<Integer> intBox = new Box<>();
        intBox.set(123);
        System.out.println(intBox.get());
    }
}
```

در نهایت، هر جا که از یه نوع استفاده می‌کنیم، چه توی تعریف متغیر، چه پارامتر متد، نوع بازگشتی، یا موقع new کردن یک شی، باید نوع جنریک رو کامل کنیم.

نکته مهم:

پارامترهای نوع توی جنریک‌ها باید کلاس باشن، نه primitiveها مثل int و یا Boolean برای همین به‌جای int از Integer استفاده کردیم.

فرض کنید ما آبجکت زیر رو تعریف کنیم:

```
Box<String> stringBox;
```

مثل این بود که از اول، کلاس Box رو به حالت زیر تعریف کرده باشیم:

```
public class Box {
    private String value;
    public void set(String value) {
        this.value = value;
    }
    public String get() {
        return value;
    }
}
```

انگار ما کلاس Box رو تخصصی کردیم تا فقط با String کار کنه و دیگه نمی‌تونه هر نوع Objectای رو بپذیره.

الان با این Box مخصوص Stringها می‌تونیم با خیال راحت کار کنیم. کامپایلر اجازه نمی‌ده چیزی غیر از String(یا زیرکلاس‌هاش) به Box اضافه کنیم. همچنین وقتی از ()get استفاده می‌کنیم، دیگه نیازی به cast کردن نداریم چون نوع خروجی از قبل مشخصه.

حالا فرض کنید میخوایم یه جعبه بسازیم که بتونیم هر نوعی چیزی رو توش بذاریم و برش داریم اما یه شرط داره، این جعبه باید حتماً از یه قاعده کلی پیروی کنه. اینجا همون جاییه که یک کلاس جنریک باید بتونه یک interface جنریک رو implement کنه. بریم ببینیم چجوری:

مثلا برای همون مثال Box که قبل‌تر داشتیم، یک اینترفیس براش مینویسیم که شامل متود set و get است.

```
interface Container<T> {
    void set(T value);
    T get();
}
```

میخوایم کلاس Boxمون بر اساس همین الگو پیش بره.

```
public class Box<T> implements Container<T> {
    private T value;
    @Override
    public void set(T value) {
        this.value = value;
    }
    @Override
    public T get() {
        return value;
    }
}
```

حالا که با کلاس‌های جنریک آشنا شدیم و فهمیدیم چطور می‌تونیم ساختارهایی بسازیم که با انواع مختلف داده‌ها کار کنن، وقتشه بریم سراغ یه بخش مهم دیگه از جنریک‌ها

متدهای جنریک

گاهی وقتا فقط کلاس‌ها نیستن که باید انعطاف‌پذیر باشن، بلکه خود متدها هم ممکنه نیاز داشته باشن با انواع مختلف داده کار کنن، بدون اینکه بخوایم برای هر نوع یه نسخه جدا بنویسیم.
یعنی در واقع شما می‌خواید یک کلاس غیرجنریک بنویسید، اما توی اون فقط یک متد کوچولوی جنریک داشته باشین.

فرض کنید میخوایم یک متود تعریف کنیم که یک آرایه بگیره و عنصر اولش رو برگردونه. می‌دونیم آرایه ممکنه از هر نوعی باشه، ولی بدون استفاده از Generic باید حالت بندی کنیم برای هر نوع آرایه مختلف مثل int و String و... متود مربوط به خودش را بنویسیم. پس اینجا بهتره که از مفهوم Generic استفاده کنیم.

```
Public class Main {
    public static <T> T getFirstElement(T[] array) {
        if (array == null || array.length == 0) {
            return null;
        }
        return array[0];
    }
}
```

توی کد بالا، وسط کلاسی که خودش generic نیست، ما یه متد کوچیک جنریک تعریف کردیم. یه مقدار با این متد ور برین و ببینید که چطوری می‌تونید صداش کنید. آیا لازمه برای اون هم توی <> نوع type parmeterتون رو مشخص کنید؟

یا مثلا متود زیر، دوتا ورودی از یک نوع میگیره و مقایسه‌شون میکنه و اون که بزرگتره رو خروجی میده.

```
public static <T extends Comparable<T>> T getMax(T a, T b) {
    return (a.compareTo(b) > 0) ? a : b;
}
```

مفهوم Bounding (Upper-Bound) و محدود سازی در Generics

خیلی وقتا نمی‌خواید هر نوعی توی جنریک استفاده بشه.حالا شاید بپرسین چرا مگه هدف جنریک همین نبود که کلی سازی کنه؟ کی این کارو باید کرد؟

وقتی یه کلاس یا متد جنریک تعریف می‌کنین، به طور پیش‌فرض می‌تونه با هر نوعی از داده‌ها کار کنه.

اما بعضی وقتا فقط استفاده و به زبان بهتر اجازه کار با برخی نوع‌های خاصی منطقی هست.

مثلا برای یه برنامه مثل ماشین حساب که فقط با عدد سر و کار داریم، فرض کنید یه کلاس نوشتید برای یه عملیات ریاضی، مثلاً مربع گرفتن:

```
class Calculator<T> {
    public double square(T value) {
        return value * value;
    }
}
```

به نظرتون مشکلش چیه؟

این نوع استفاده رو ببینید:

```
Calculator<String> calc = new Calculator<>();
calc.square("hello");
```

خب این منطقی نیست که مربع یه رشته رو داشته باشیم. اصلا چنین چیزی وجود نداره. و این کار ارور میده به همین دلیلی که دیدید.

حالا راه حلش چیه؟ گذاشتن محدودیت روی نوع داده قابل استفاده یا به عبارتی کران دار کردنش که همون Bounding هست.

پس توی این مثال باید محدودش کنیم به اعداد. برای این کار لازمه از کلاسی به اسم Number استفاده کنیم. Number یک کلاس از پیش تعریف‌شده (built-in) در Java هست که توی java.lang قرار داره، و همه کلاس‌های عددی ازش ارث‌بری می‌کنن.

مثلا اینطوری (بیشتر بدانید):

```
abstract class Number {
    abstract double doubleValue();
}

class Integer extends Number {
    int value;
    Integer(int value) { this.value = value; }

    @Override
    double doubleValue() {
        return (double) value;
    }
}
```

پس وقتی یه متغیر از نوع Number داشته باشید، می‌تونید به راحتی ازش doubleValue() بخواید.

پس این کارو میکنید:

```
class Calculator<T extends Number> {
    public double square(T value) {
        return value.doubleValue() * value.doubleValue();
    }
}
```

حالا برگردیم به جنریک.

در واقع با این کار داریم میگیم فقط نوع‌هایی مثل Integer، Double، یا هر چیزی که از Number ارث‌بری کرده باشه، مجازن. به طور دقیق تر دارین میگین که

«هر چی Tهست، **باید از Number ارث برده باشه**، بنابراین مثلا کامپایلر مطمئنه که T این متد رو داره(doubleValue() : »

یعنی حتی اگه کسی بیاد از Calculator<Integer> یا Calculator<Double> یا حتی Calculator<BigDecimal> استفاده کنه، مشکلی نیست، چون همه‌شون از Number اومدن.

ولی اگه کسی بگه:

```
Calculator<String> calc = new Calculator<>();
```

به ارور می‌خوره! چون String از Numberارث نبرده و بنابراین متد doubleValue() هم نداره.

وقتی می‌گین T extends Number، داریم به کامپایلر می‌گیم:

من تضمین می‌کنم که T حداقل یه Number هست

پس کامپایلر با خیال راحت می‌تونه doubleValue()، intValue()و بقیه‌ی متدهای Number رو روش صدا بزنه.

یه نکته یکم حرفه‌ای تر:

اگه به جای کلاس Number از یه interface استفاده کنین (مثلاً Comparable<T>)، باز هم همین ایده کار می‌کنه.

```
class Sorter<T extends Comparable<T>> {
    public T max(T a, T b) {
        return a.compareTo(b) > 0 ? a : b;
    }
}
```

گاهی وقتا نمی‌خواید بگین دقیقاً چه نوعی داریم، فقط می‌خوایم بگیم «یه چیزی» هست.

Wildcard چیه و چرا و چطوری استفاده میشه؟

اگه خیلی خلاصه مرور کنید، تا اینجا یاد گرفتید که چطوری با Generics کلاس یا متدهایی بنویسید که بشه با انواع مختلف داده‌ها به‌صورت type-safe کار کرد. اما یه جایی ممکنه به مشکل بخورین. مثلاً فرض کنین یه متد نوشتین که قراره یه لیست رو چاپ کنه. خب خیلی منطقیه که بنویسین

```
public void printList(List<Object> list) {
      for (Object obj : list) {
            System.out.println(obj);
      }
}
```

ولی اگه این کد رو با یه لیست از نوع List<Integer> صدا بزنین، کامپایلر بهتون ایراد می‌گیره!
چرا؟ مگه Integer هم خودش یه Object نیست؟ پس چرا نمی‌تونیم یه List<Integer> رو به متدی بدیم که لیست از نوع Object می‌گیره؟

ماجرا اینه که در Java ، List<Object> و List<Integer> هیچ ربطی به هم ندارن — حتی اگه Integer یه Object باشه، ولی List<Integer> هیچ‌وقت فرزند List<Object> نیست. این با چیزیه که توی خیلی از زبون‌های دیگه مثل Kotlin یا Python می‌بینین که راحت می‌تونن لیست انواع مختلف رو قبول کنن. ولی جاوا سخت‌گیره(که بعدا به بررسی دقیقش میرسیم).

دقیقاً اینجاست که Wildcard وارد می‌شه تا نجاتتون بده.

Wildcard یعنی می‌تونین بگین: یه لیستی می‌خوایم، حالا مهم نیست از چه نوعیه، فقط یه چیزی توشه!

پس به جای List<Object> ، بنویسین List<?>

مثلاً متد بالا رو این‌طوری اصلاح کنید:

```
public void printList(List<?> list) {
    for (Object obj : list) {
        System.out.println(obj);
    }
}
```

الان دیگه این متد می‌تونه با List<String> ، List<Integer>، یا حتی List<MyCustomClass> هم کار کنه. این همون انعطاف‌پذیریه که با wildcard به دست میاد.

چرا از wildcard ها استفاده می‌کنیم و چرا List<Object> کافی نیست؟

بیاید دقیق تر بررسی کنیم که چرا به جای List<?> از List<Object> استفاده نکنید؟ در ظاهر، به نظر می‌رسد این دو باید رفتاری شبیه به هم داشته باشند، اما تفاوت مهمی بین آن‌ها وجود دارد از کد بالا به یاد دارید که فقط می‌تواند لیست‌هایی از نوع Object را بپذیرد. به عبارت دیگر، اگر لیستی از نوع List<String>، List<Integer> یا هر نوع دیگری را به این متد بدید، با خطای کامپایل مواجه خواهید شد.

دلیل این محدودیت آن است که List<String> یک زیرمجموعه (subtype) از List<Object> محسوب نمی‌شود، حتی اگه String زیرنوع Object باشد. داستان این اتفاق یه خاصیت توی جاواست که بهش Generic Type Invariance میگن. یعنی چی؟ بیاید تا روی یه مثال عملی بررسیش کنین.

فرض کنین ،دو تا کلاس دارید Animal و Dog :

و Dog از Animal ارث‌بری کرده:

```
class Animal {}
class Dog extends Animal {}
```

حالا سؤال:

اگه یه لیست از Dog داشته باشید(List<Dog>) ، آیا می‌تونید اون رو به یه متد که انتظار List<Animal>داره، پاس بدید؟

مثلا بگید:

```
public void addAnimal(List<Animal> animals) {
    animals.add(new Animal());
}
```

و اینجوری صداش کنید:

```
List<Dog> dogs = new ArrayList<>();
addAnimal(dogs);
```

این کد بهتون ارور میده، چون در حقیقت درسته که Dog یه ساب تایپ از Animal هستش و خب منطقیه که یه لیست از سگ ها هم یه ساب تایپ از یه لیست از حیوونا باشه اما جاوا همچین چیزی رو قبول نداره. به بیان دیگه جنریک ها توی جاوا invariant هستن. خب این خاصیت به چه دردی میخوره؟ برگردیم به مثالمون، چرا List<Dog> نمی‌تونه جای List<Animal> بشینه؟

جواب ساده‌ست: چون اگه جاوا اجازه می‌داد این کار رو بکنین، می‌تونستین به یه لیست از سگ، یه حیوان غیرسگ مثل یه گربه یا حتی خود کلاس (Animal) اضافه کنید، که اون موقع دیگه لیست خالصی از سگ‌ها نیست!

```
List<Dog> dogs = new ArrayList<>();
addAnimal(dogs);
```

که متد استفاده شده به این صورت هست:

```
public void addAnimal(List<Animal> animals) {
    animals.add(new Animal());
}
```

خب الان توی dogs که قراره فقط Dog باشه، یه Animal ریخته شده که ممکنه Dog نباشه. این یعنی هرجا بعداً بخواید با فرض اینکه همه‌ی اعضای لیست از نوع Dog هستن کار کنید، ممکنه به مشکل بخورید.

راه‌حل چیه؟

Wildcards-

خب، گفتیم که جنریک‌ها invariant هستن، پس نمی‌تونین یه List<Dog> رو جایی پاس بدین که List<Animal> انتظار می‌ره.
اما اگه فقط می‌خواید **از لیست بخونین** (چیزی بهش اضافه نکنید)، اون‌وقت می‌تونیمد از wildcardاستفاده کنید.

```
public void readAnimals(List<? extends Animal> animals) {
    for (Animal a : animals) {
        System.out.println(a);
    }
}
```

این یه لیستیه از چیزایی که یه‌جوری Animal هستن (یعنی یا خود Animal یا یکی از زیرکلاساش، مثل Dog یا Cat اینطوری می‌تونید بدون مشکل List<Dog>، List<Cat> یا هر نوع دیگه‌ای از Animal رو به این متد پاس بدید. ولی حواستون باشه: نمی‌تونید چیزی به این لیست اضافه کنید (به‌جز null) چون کامپایلر دقیق نمی‌دونه نوع واقعی لیست چیه، فقط می‌دونه «یه چیزی که Animal هست». پس اگه بخواید یه Animal ساده بهش اضافه کنید، ممکنه اون لیست در واقع List<Cat>باشه، که تو اون صورت نمی‌تونید یه Animal عمومی بریزید توش. واسه همین، جاوا اجازه‌ی add کردن نمی‌ده.

حالا برعکسش چیه؟

اگه فقط می‌خواید به لیست چیزی اضافه کنید و کاری به خوندنش ندارید، می‌تونید از wildcard پایین‌رو (lower-bounded wildcard) استفاده کنید:

```
public void addDog(List<? super Dog> list) {
    list.add(new Dog());
```

```
}
```

این یه لیست از چیزیه که یا خود Dog هست یا یکی از والدینش مثلاً Animal یا حتی Object.
شما مطمئنید که هرچی که باشه، Dog می‌تونه توش جا بشه. پس list.add(new Dog()) کاملاً اوکیه.

ولی این بار برعکسه: نمی‌تونید با اطمینان چیزی از لیست بخونید و نوعش رو بدونید. چون ممکنه لیست از نوع Object باشه و توش هرچی ریخته باشن! تنها چیزی که می‌دونید اینه که لیست مناسب اضافه کردن Dog بوده نه بیشتر.

می‌تونیم چی کار کنیم؟	چی استفاده کنیم؟	هدف ما چیه؟
بخونید ✅، ولی نمی‌تونید چیزی اضافه کنیم❌	? extends T	فقط می‌خواید **بخونین**
بنویسید ✅، ولی نمی‌تونیدبا نوعش بخونیم❌	? super T	فقط می‌خواید **بنویسین**

یه نکته‌ی مهم:
Wildcard فقط توی استفاده از کلاس‌ها و متدهای Generic کاربرد داره، نه توی تعریفشون. یعنی نمی‌تونین یه کلاس بنویسین مثل — class Box<?> { ... } این اصلاً مجاز نیست. اونجا باید نوع Generic مشخص بشه (مثلاً T یا E یا هرچی که بخواین.)

انواع Wildcard

Unbounded Wildcard (?) : این نوع Wildcard هیچ محدودیتی روی نوع داده ندارد. به عبارت دیگر، شما می‌توانید لیستی از هر نوع داده‌ای حتی Object داشته باشید.

```
public void printList(List<?> list) {
    for (Object obj : list) {
        System.out.println(obj);
    }
}
```

در اینجا ? به این معنی است که این پارامتر متد می‌تواند یک لیست از هر نوعی باشد.

Upper Bounded Wildcard (? extends T) : این نوع Wildcard به شما این امکان رو می‌ده که پارامترهای نوع فقط از یه نوع خاص (یا زیرکلاس‌های اون) استفاده کنن. یعنی شما می‌خواید فقط با انواع خاصی کار کنید که از یه کلاس یا اینترفیس خاص ارث بریده باشن. در واقع، این طوری می‌تونید محدودیت‌هایی بذارید که فقط داده‌هایی که با اون کلاس یا اینترفیس هماهنگ هستن وارد بشن، نه هر نوع داده‌ای.

```
public void printNumbers(List<? extends Number> list) {
    for (Number number : list) {
        System.out.println(number);
    }
}
```

در این مثال، فقط لیست‌هایی که از نوع Number یا زیرکلاس‌های آن (مثل Integer، Double، Float و ...) هستند، پذیرفته می‌شوند.

Lower Bounded Wildcard (? super T): به شما این امکان رو می‌ده که پارامترهای نوع از یه نوع خاص یا هر نوعی که از اون ارث برده باشه، استفاده کنن. به عبارت دیگه، این نوع Wildcard زمانی کاربرد داره که بخواید نوع داده‌ها رو محدودتر کنید، ولی در عین حال اجازه بدید که کلاس‌های دیگه هم بتونن از اون استفاده کنن. اینطوری می‌تونید محدوده‌ی استفاده رو کنترل کنید، ولی در عین حال انعطاف‌پذیری بیشتری برای انواع مختلف داده‌ها داشته باشید.

```
public void addNumbers(List<? super Integer> list) {
    list.add(1);
    list.add(2);
}
```

در اینجا، لیست فقط می‌تواند انواع Integer یا والدین آن (مثل Number یا Object) را بپذیرد.

چگونه Wildcard ها به مدیریت پیچیدگی کد کمک می‌کنند؟

با استفاده ازWildcard ، می‌تونید کدهای جنریک رو طوری بنویسید که برای انواع مختلف داده‌ها قابل استفاده باشه، بدون اینکه نیاز باشه دقیقاً نوع داده رو مشخص کنید. این ویژگی مخصوصاً وقتی که کد شما باید با داده‌های مختلف از انواع مختلف کار کنه، خیلی مفید می‌شه. یعنی می‌تونید کد رو بازتر و انعطاف‌پذیرتر بنویسید و نیاز به تغییرات زیاد برای انواع مختلف داده‌ها نداشته باشید.

بیایم با یه مثال ساده و سریع مرورشون کنیم:

```java
public class WildcardExample {
    public static void main(String[] args) {
        List<Integer> intList = List.of(1, 2, 3);
        List<Double> doubleList = List.of(1.1, 2.2, 3.3);

        // Upper Bounded Wildcard
        printNumbers(intList);
        printNumbers(doubleList);

        // Lower Bounded Wildcard
        List<Object> objectList = new ArrayList<>();
        addNumbers(objectList);
    }

    // Upper Bounded Wildcard
    public static void printNumbers(List<? extends Number> list) {
        for (Number number : list) {
            System.out.println(number);
        }
    }

    // Lower Bounded Wildcard
    public static void addNumbers(List<? super Integer> list) {
        list.add(1);
    }
}
```

در این کد می‌بینید که چطور Wildcard ها به شما این امکان رو می‌دهند که متدها رو عمومی‌تر و انعطاف‌پذیرتر بنویسید، بدون اینکه نیازی به مشخص کردن نوع دقیق داده‌ها داشته باشید .

لیست‌های مختلف داریم:

intList که یه لیست از Integerهاست.

doubleList که یه لیست از Doubleهاست.

Upper Bounded Wildcard (? extends Number):

توی این متد می‌خوایم هر لیستی که توش از انواع مختلف Number استفاده شده رو چاپ کنیم.
یعنی هم Integer هم Double و حتی انواع دیگه‌ای که از Number ارث می‌برن.

متد printNumbers می‌تونه هر نوع داده‌ای که از Number مشتق شده رو دریافت کنه و چاپ کنه. به همین دلیل، وقتی این متد رو به لیست‌های intList و doubleList می‌دین، هیچ مشکلی پیش نمیاد.

Lower Bounded Wildcard (? super Integer):

اینجا دارین لیستی رو می‌سازین که می‌تونه هر چیزی که از Integer کوچیک‌تر باشه رو قبول کنه. یعنی می‌تونیn Integer رو به این لیست اضافه کنیk و حتی کلاس‌هایی مثل Objectرو هم بهش بریزین.

در متد addNumbers می‌گید که می‌خواین حداقل Integer رو به لیست اضافه کنین، نه کم‌تر از اون. این باعث می‌شه که بتونین چیزهایی مثل Integer و انواعی که پدرشون Integer باشه رو به لیست اضافه کنیم.

خلاصه:

Upper Bounded Wildcard می‌ذاره با هر لیستی که از یه نوع خاص ارث برده، کار کنید (مثلاً همه چیزهایی که از Numberارث می‌برن)

Lower Bounded Wildcard می‌ذاره هر چیزی که از یه نوع خاص پایین‌تر باشه، به لیست اضافه کنید (مثلاً می‌تونی Integerو هر چیزی که بالاتر از اون باشه رو اضافه کنی.)

این‌ها باعث می‌شن که برنامه‌هاتون انعطاف‌پذیرتر بشن و دیگه نیازی نباشه دقیقاً نوع داده‌ها رو مشخص کنین.

در آخر میبینین که Wildcard ها در جنریک‌ها ابزارهای خیلی قدرتمندی هستند که می‌تونند به شما کمک کنند کدهای خودتون رو انعطاف‌پذیرتر و قابل استفاده مجددتر بنویسید. این ابزارها به شما اجازه می‌دهند با انواع مختلف داده‌ها کار کنید، در حالی که همچنان از مزایای ایمنی نوع در زبان جاوا استفاده می‌کنید.