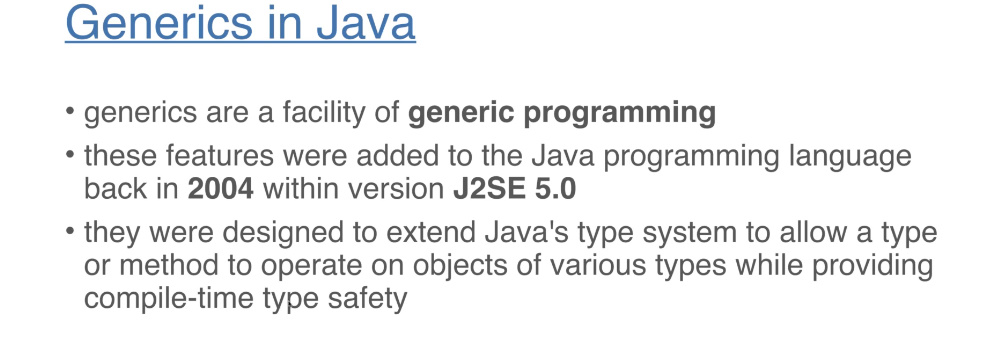
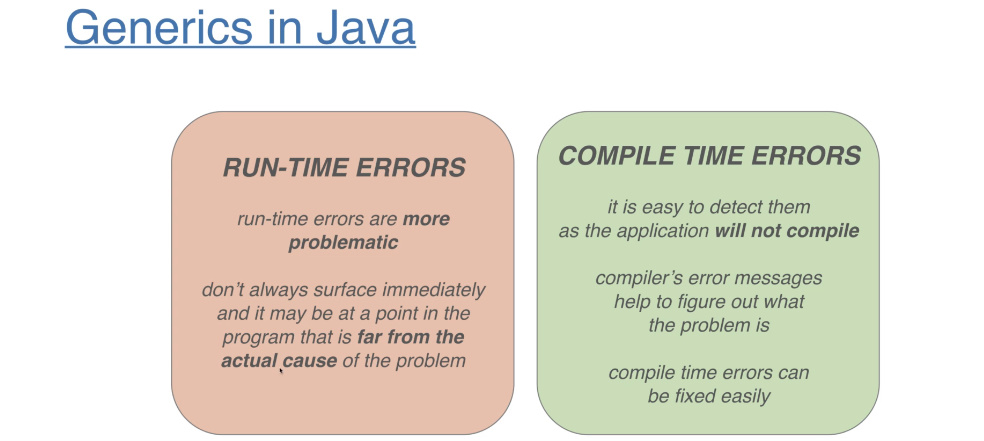
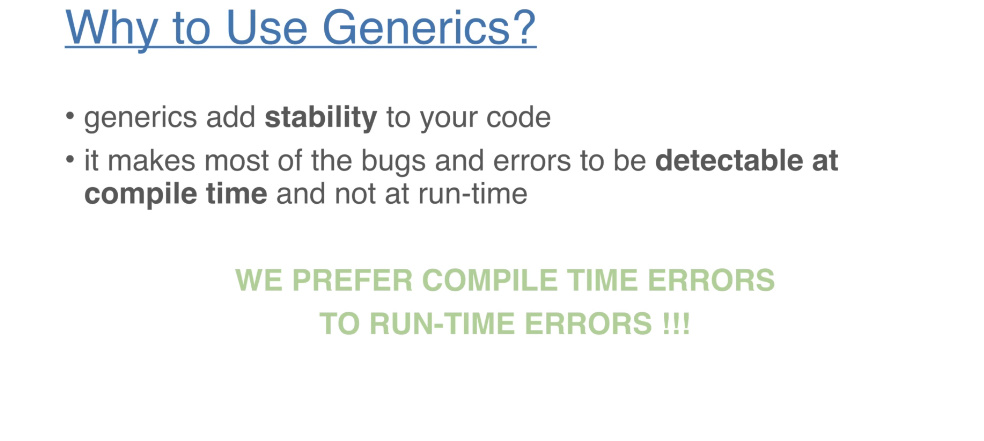
**Generics and Generic Programming**

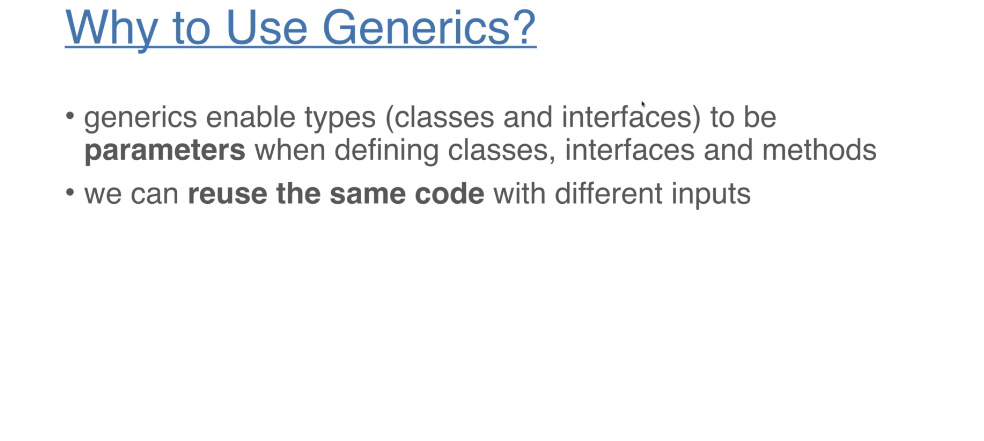




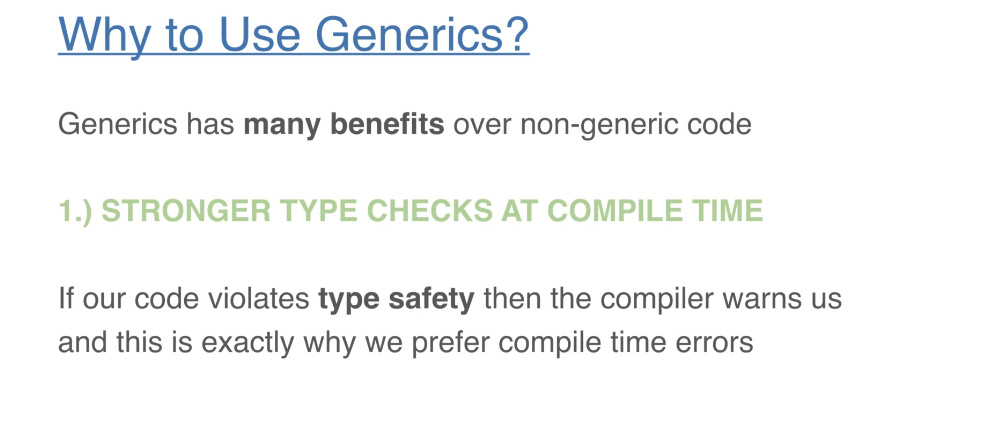


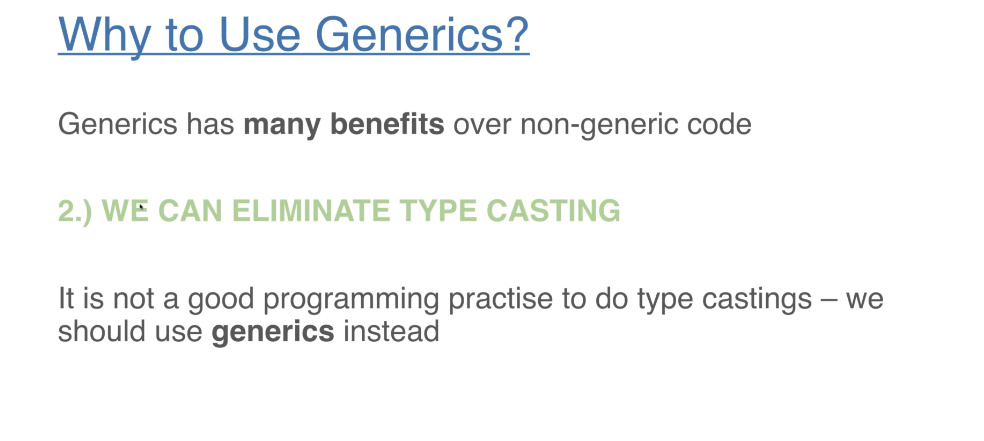
Deci, Generics anume ne ajuta sa transformam posibilele errori din Run-Time errors in Compile time errors, si asa e mult mai usor de a le rezolva.





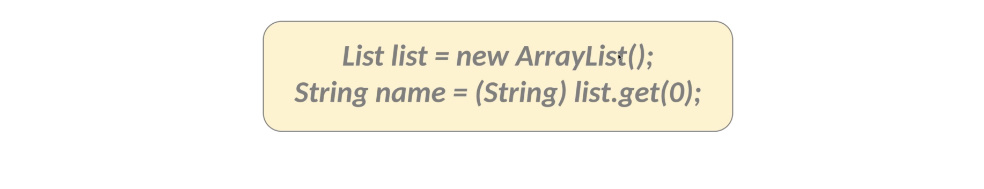
**Generics Avantaje**





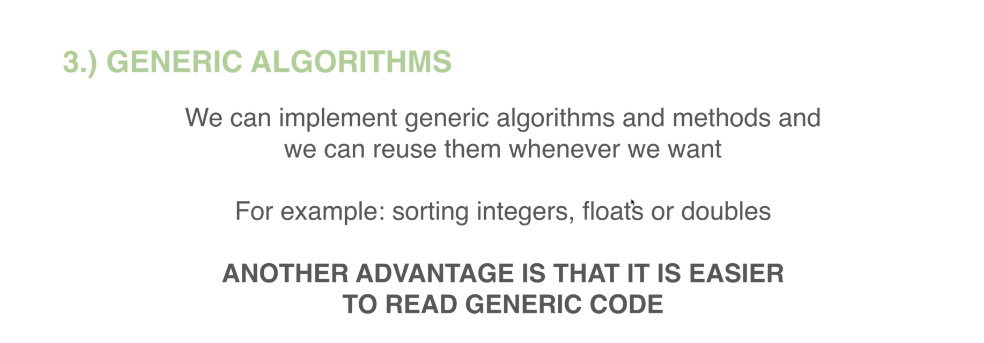
Ex de typecasting:

Daca nu folosim generics, trebuie sa facem cast la obiect in String, iar JVM trebuie sa testeze type casting abea la runtime. In plus, Java va returna mereu un obiect de tip Object daca noi nu specificam tipul la Generics.



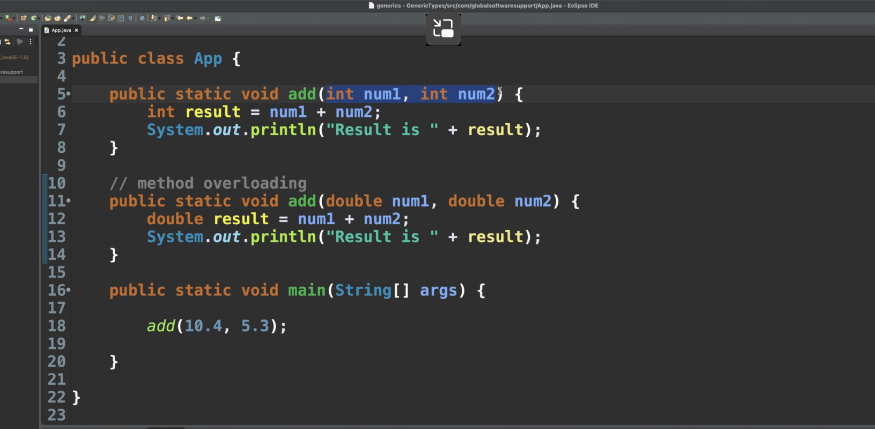
Insa, daca folosim generics, nu trebuie sa mai facem type cast in String, caci compiler stie deja tipul si poate sa o faca el, si daca facem vreo greseala, ne anunta chiar la compile time. Java va face automat castingul.





Nu ar fi o idee buna sa cream cate o metoda pentru fiecare tip de date primitve.

* Generics code e mult mai usor de citit si inteles
* Generics ne pot ajuta sa evitam Overloading cand metoda face acelasi lucru pentru orice tip de date:



**Creating**

* O metoda naiva de a mima generics este de a folosi Object, asa cum orice instanta este oricum si de tip Object
* Dar asta nu e o practica prea buna



Dar, daca vom folosi in loc de 45 alt numar, ca 45.5, vom descoperi eroarea abea la run time, nu la compile time.



* Pentru a folosi gnerics, dupa numele clasei punem <> si definim cum se vor numi generics, apoi i folosim unde vrem.
* Deobicei se foloseste T
* 
* Apoi, cand cream obiectul, specificam tipul la generic:

Store<String> store = new Store<String>();

* In principiu, chiar si cand folosim Generics, Java tot va folosi Object type, doar ca se va ocupa mereu de casting si va oferi informatii la compile time
* <> se numeste diamond operator
* Daca la crearea obiectului nu vom specifica noi tipul, va trebui sa facem iar type casting:

Store store = new Store();  
store.setItem(45.5);  
Double integer = (Double) store.getItem();

* Asta e o practica foarte rea, si se cheama **raw type**
* Java va folosi automat <Object>

**Generics Methods**

* Daca de exemplu clasa nu a defenit nici-un generic, putem defini generics doar pentru o metoda.
* Pentru asta, punem generics pana la tipul de return al metodei:
* public class Test {  
   public static void main(String[] args) throws InterruptedException {  
    
   }  
   public static <T> void testMethod(T item){  
    
   }  
    
  }
* Cand apelam o metoda generics, nu mai folosim <>, caci Java va determina automat tipul la generic pe baza la parametrul trimis de noi:
* public class Test {  
   public static void main(String[] args) throws InterruptedException {  
    
   *testMethod*("Test");  
   }  
   public static <T> void testMethod(T item){  
   System.*out*.println(item);  
   }  
    
  }
* Generics poate fi folosit si pentru a crea arrays, dar numai nu arrays de date primitive:

public class Test {  
 public static void main(String[] args) throws InterruptedException {  
  
 int[] array = {1,2,3,4,5};  
 *testMethod*(array);  
 }  
 public static <T> void testMethod(T[] item){  
 for(T t : item)  
 System.*out*.println(t);  
 }

**dar:**

Integer[] array = {1,2,3,4,5};

**Bounded Type parameters**

* Uneori s-ar putea sa avem nevoie sa restrictionam tipurile care pot fi folosite ca type arguments
* De ex, poate ca o metoda are de gand sa lucreze doar cu numere, si deci va avea nevoie sa definim doar un generic type care mosteneste de la clasa Number
* Anume asta si este bounded type parameters
* Bounded Types ne permit sa limitam tipurile ce pot fi folosite ca generics
* **Bounded Type Parameters e folosit cu extends** keyword:
* este pentru a ne asigura ca tipul de generic dat extinde sau implementeaza clasa/interfata, sau e de tip clasa/interfata
* <T extends Comparable>
* Atentie! E interesant ca folosim extends indiferent ca ceea ce vine dupa el e o interfata sau clasa
* Acum, vom putea folosi orice metoda din Comparable
* Fara acest extends, nu am putea folosi decat metode ale clasei Object

public static <T> void testMethod(T item){  
 item.getItem();  
}

* **dar cu bounded tipe, putem folosi metodele specifice:**
* public static <T extends Store<T>> void testMethod(T item){  
   item.getItem();  
  }

**Cu el, vom putea folosi metodele claselor/interfetelor de dupa extends**

* putem utiliza si & pentru a specifica mai multe tipuri:

**<T extends Number & Comparable>**

* Atentie! Inaintea lui & putem pune doar o clasa cel mult. Dupa &, sunt acceptate doar interfate.
* public static <T extends Store<T> & Thread> void testMethod(T item){  
   item.getItem();  
  }

public static <T extends Store<T> & Runnable & Comparator> void testMethod(T item){  
 item.getItem();  
}

**Asa e bine**

* **Putem pune si doar interfete:**
* public static <T extends Runnable & Comparator> void testMethod(T item){

In asa mod, daca vom oferi un tip de generics care nu implementeaza sau extinde ce am setat noi, vom primi eroare:

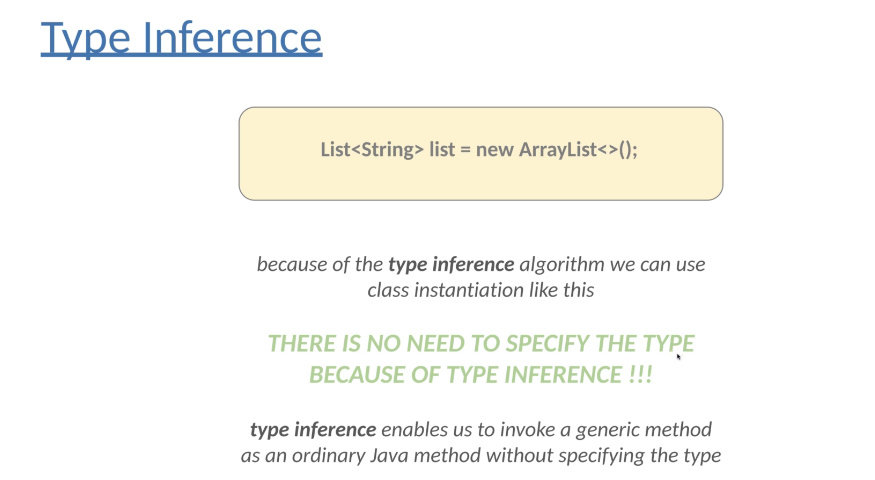
public static void main(String[] args) throws InterruptedException {  
 *compare*(new Store<String>(),new Store<String>());  
  
}  
public static <T extends Comparable<T>> int compare(T param1, T param2){  
 return param1.compareTo(param2);  
}

public static void main(String[] args) throws InterruptedException {  
System.*out*.println(*difference*(100,50));  
}  
public static <T extends Number> int difference(T param1, T param2){  
 return param1.intValue() - param2.intValue();  
}

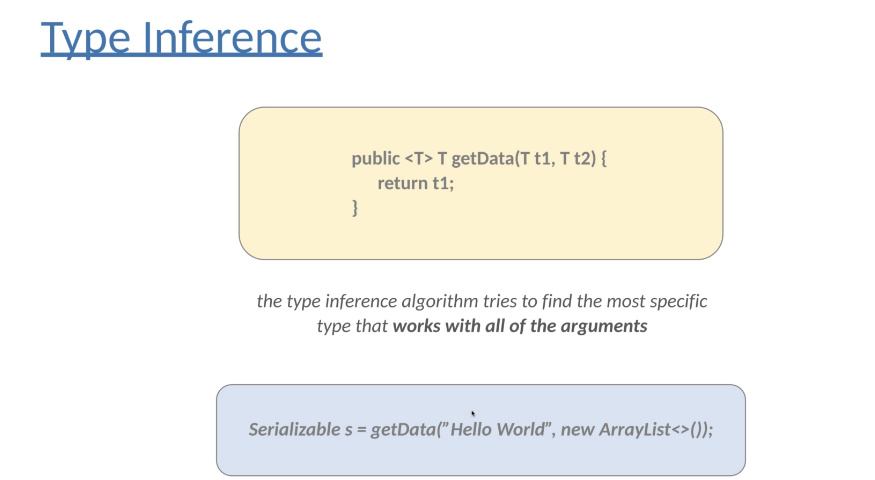
* **Chiar daca folosim extends Number, asta nu inseamna ca vom putea folosi < ,>,+,-,.... caci nu putem folosi acesti operatori asupra la obiecte**
* public static <T extends Number> boolean difference(T param1, T param2){  
   return param1>param2;  
  }
* **Nu putem folosi super cu Generics!!!**
* public static <T super b > void copy(List<T> from,List<T> to){  
   to.add(from.get(1));  
  }

**Type Inference**

* **Type Inference** – abilitatea compilatorului de a se uita la fiecare invocatie de metoda si declaratie corespunzatoare ca sa determine tipul argumentului(a lui T de ex)
* **Type Inference Algorithm determina tipul argumentului ce corespunde lui <T>**
* Exemple:



Deci, nu mai e nevoie sa scriem <String> si la ArrayList(), caci compilatorul, datorita la Type Inference, va prelua tipul la generic de la List <String>

* 

Aici, Java va folosi Type Inference Algorithm pentru a incerca sa depisteze un type la Generic care sa mearga pentru toti parametrii si pentru variabila s. Deoarece String implementeaza Serializabl si ArrayList tot, si variabila s e de tip Serializable, Java va intelege ca <T> va fi de tip Serializable. Oricum noi nu putem sa punem <> in cazul dat

**Method explicit Type Generics**

* Totusi, exista o metoda prin care sa specificam ce tip de generics sa fie pus la o metoda, si anume prin a pune <> in fata metodei de executat.
* public class Test {  
   public static void main(String[] args) throws InterruptedException {  
   System.*out*.println(Test.<Integer>*difference*(100,50));  
   }  
   public static <T extends Number> int difference(T param1, T param2){  
   return param1.intValue()+param2.intValue();  
   }  
  }

Atentie, nu putem pur si simplu scrie :

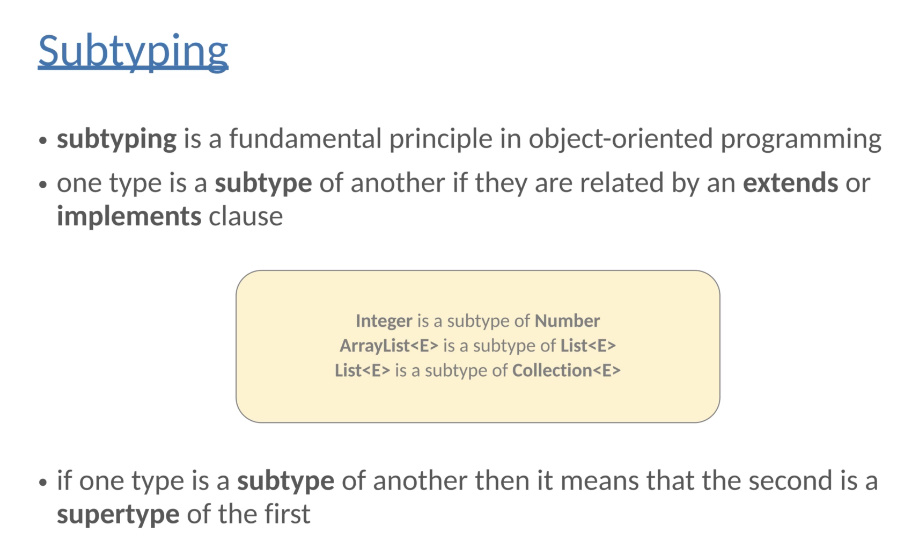
<Type>metoda(arg)

**Trebuie:**

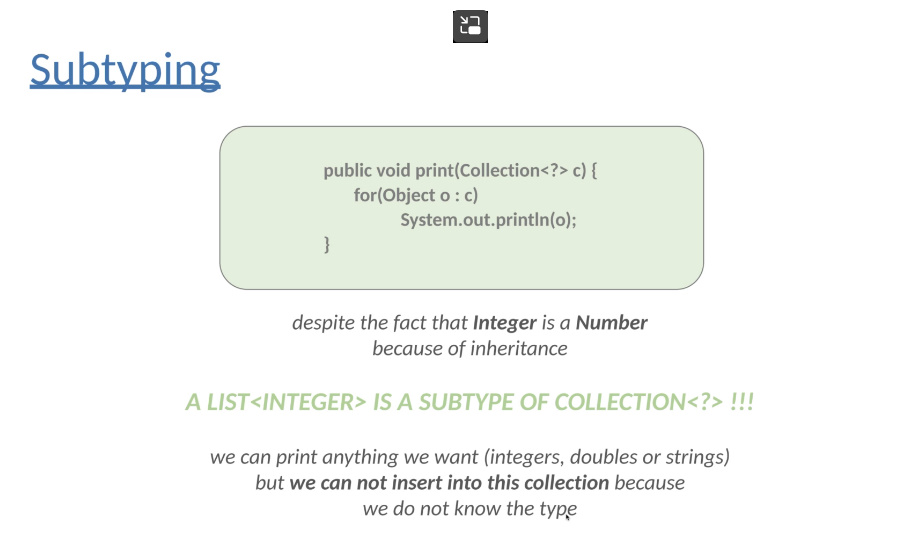
**clasa/obj.<Type>metoda(arg)**

* **Asta se numeste Type witness**
* Asa, Type Inference nu mai trebuie sa determine tipul la Generics
* public class Test {  
   public static void main(String[] args) throws InterruptedException {  
   Store<String> store = new Store<>(); //preia automat String  
   List<Store<String>> list = new ArrayList<>(); // preia automat Store<String>  
   *add*(store,list); //deduce si singur ca tipul lui T va fi Store<String>  
   Test.<Store<String>>*add*(store,list); // ii spunem noi care va fi generics type, dar e optional  
   }  
   public static <T> void add(T item, List<T> list){  
   list.add(item);  
   }  
    
  }  
    
  class Store<T>{  
   private T item;  
    
   public T getItem() {  
   return item;  
   }  
    
   public void setItem(T item) {  
   this.item = item;  
   }  
  }

**WildCards**



* **Wildcards se folosesc doar cand cream obiectele sau variabilele mai bine zis, nu la definirea clasei sau metodei, caci nu e generics!!!**
* Desi Integer e un subtip a lui Number, List<Integer> nu e sub tip a lui List<Number>. Aici deja trebuie sa consideram wildcards
* **wildcards** – supertype la orice colections si nu doar.
* **Se noteaza cu** ? adica asa <?>
* **Cu wildcards, nu se va determina niciodata tipul la obiect**.

****

* **TOTUSI, problema e ca nu putem insera date in aceasta collection din cauza ca nu cunoastem tipul, si nici nu putem citi date cu foreach, doar daca folosim Object sau get().**

**Unbounded wildcards**

* **Unbounded wildcards este pur si simplu <?>**

public static void main(String[] args) {  
 List<Integer> list = Arrays.*asList*(1,2,3,4,5);  
 *print*(list);  
}  
public static void print(List<Object> list){  
 list.forEach(System.*out*::println);  
}

**Deci, desi Integer mosteneste de la Object, vedem ca List<Integer> nu e subtip a lui List<Object>, si avem o eroare la compilare**

* Dar, daca in metoda print() vom folosi ? in loc de Object, totul va lucra bine:
* **WildCards** se folosesc cand nu stim ce tip de date va fi trimis ca Generics, si folosirea lui inseamna ca oricare poate fi folosit.
* **WildCards** pot fi folosite oriunde, nu doar cu Collections
* Totusi, aici apare o problema. ? nu spune ce fel de obiect va fi stocat, deci nu putem folosi altceva decat Object pentru a extrage un element din lista, si nici nu putem accesa alte metode decat cele ale lui Obejct:

public class Test {  
 public static void main(String[] args) {  
 List<Shop> list = new ArrayList<>();  
 *print*(list);  
 }  
 public static void print(List<?> list){  
 for(Shop s : list){  
 s.hello();  
 }  
 }  
}  
class Shop{  
 public void hello(){  
 System.*out*.println("Hello");  
 }  
}

**Trebuie:**

for(Object s : list){

* **Cu <?> nu putem folosi foreach, doar daca Object, nu putem adauga iteme, dar putem folosi get() pentru a primit iteme.**

**Upper Bounded Wildcards(Read Only)**

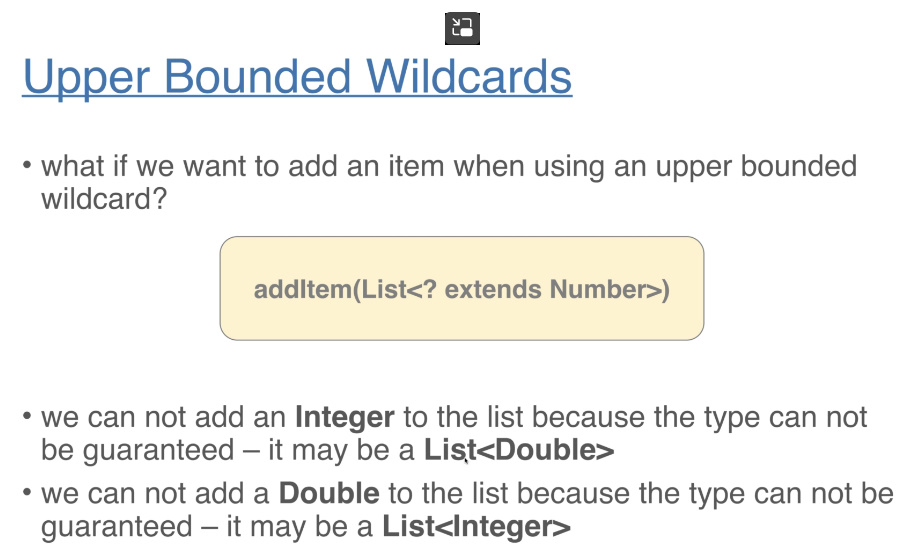
* **Upper Bounded Wildcards Are forma <? extends Class/Interface>**
* Deci <? extends T> inseamna ca se pot lua liste care pastreaza elemente care extind sau iplementeaza T, ca si in situatiile precedente cu extends
* Asa, putem folosi metodele lui T
* **Sunt pentru a citi datele doar folosind un foreach(T t : list), deci folosim ceea ce e dupa extends in foreach**
* Totusi, aici apare o problema. Deoarece nu stim cu ce tip de date avem de a face, nu vom putea adauga iteme intr-o lista ce folosete upper bounded wildcards, dar le vom putea citi doar. Asta e din cauza ca Java nu stie ce tip de date se stocheaza in lista. Doar lista ce are <? extends ...> va avea o referinta catre o lista ce stie concret ce obiecte pastreaza, dar lista cu ? nu va sti asta, si nu va risca sa adauge obiecte cand nici nu stie ce fel de obiecte se pastreaza la sigur acolo.



* Extends doar spune ca obiectele inserate vor fi suptipuri ale unui anumit tip, si atat, nu spune anume si ce tip vor fi obiectele inserate.

public static void main(String[] args) {  
 *read*(Arrays.*asList*(1,2,3,4,5));  
}  
public static void read(List<? extends Number> list){  
 list.add(new Integer(10));  
}

**Deci, list are referinta la o lista ce contine Integer object, dar el nu stie de asta. El stie pur si simplu ca elementele din lista la care are referinta extinds clasa Number si deci poate accesa blocul corespunzator clasei Number din iteem. Nu poate adauga vreun element, caci risca asa sa adauge ceva ce lista originala nu accepta, ca doar are referinta la ea.**

****

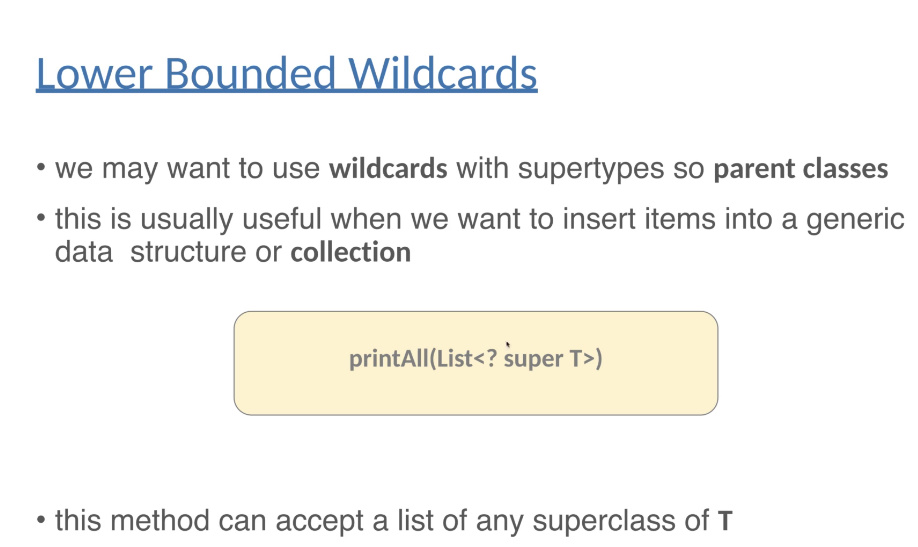
**Deci, chiar si cand parametrul dat va primi o referinta la o lista, Java nu va sti ce iteme stocheaza acea lista, adica nu le va sti tipul, si de aceea nici nu va putea adauga vreun element in lista ce foloseste upper bounded wildcards.**

**Faptul ca stim ca elementele sunt subtype ale lui Number nu ne spune nimic despre ce sunt ele anume**

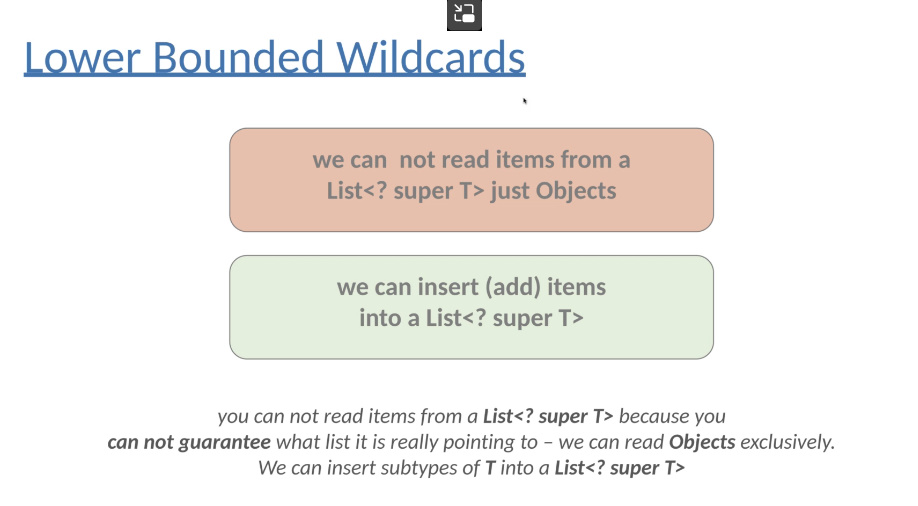
* **Dar, putem citi itemele din liste folosind ce e dupa extends,adica Number:**

public static void main(String[] args) {  
 *read*(Arrays.*asList*(1,2,3,4,5));  
}  
public static void read(List<? extends Number> list){  
 for(Number n : list)  
 System.*out*.println(n);  
}

**Lower Bounded Wildcards(Add only)**

* **Are forma <? super T>**
* Este pentru a adauga date doar
* 
* **Deci, folosind <? super T> inseamna ca putem avea orice obiect care e supertip a lui T, adica daca T implementeaza interfete sau extinde clase, vom putea pastra anume iteme care sunt de tipul la interfetele sau clasele implementate/extinse de T sau chiar T**

public static void main(String[] args) {  
 List<? super Integer> list1 = new ArrayList<Integer>();  
 List<? super Integer> list2 = new ArrayList<Comparable<?>>();  
}

* De ex, daca folosim <? super Integer>, vom putea stoca iteme de tip Number, Comparable, Serializable caci ele toate sunt implementate de Integer class si vom putea adauga iteme de tip Integer. Dar, putem chiar avea si iteme de tip Integer. <? super Integer> se refera si la Integer, nu doar la superclasele lui
* ****
* **nu putem citi date, doar de tip Object sau doar cu metoda get()**
* Cu un foreach citirea datelor nu e posibila, dar e posibila cu get()
* public static void add(List<? super Integer> list){  
   System.*out*.println(list.get(0));  
   for(Comparable<?> c : list) { }  
  }

Nu are de unde Java sti ca in lista vor fi elemente de tip Comparable sau Number etc.

* **cu foreach putem lua doar Object**
* public static void add(List<? super Integer> list){  
   System.*out*.println(list.get(0));  
   for(Object c : list) { }  
  }

public static void main(String[] args) {  
 List<Number> list = Arrays.*asList*(1,2,3,4);  
 *add*(list);  
}  
public static void add(List<? super Integer> list){  
 list.add(new Integer(10));  
}

**Putem adauga obiecte de tip care sunt dupa super,adica Integer aici, deoarece indiferent de ce tip de obiecte vor fi stocate in lista originala, ele vor fi parinti ale lui Integer class, si cand oferim de ex unei variabile de tip copil o variabila de tip parinte, ea va accesa blocul corespunzator.**

**Adica, mai sus va fi ceva de genul:**

**Comparable/Number variabila = new Integer();**

**Dar putem folosi doar Integer, nu clase copii ale lui Integer pentru a adauga elemente, ca nu are de unde Java sti care clase parinti ale lui Integer vor fi in viitoarele liste.**

**Metoda pentru a copia elemente dintr-o lista si ale pune in alta(orice tip de lista)**

public <T>void copy(List<? extends T> source, List<? super T> destination) {

for(T item : source){

destination.add(item);

}

}

Deci, putem citi iteme din source list folosit ce e dupa extends, adica T, si putem adauga iteme in destination cu ce e dupa super

Nu e nicio problema ca dupa super e Integer, si noi i vom trimite ca referinta o List<Integer>, si acelasi lucru ar fi valabil si daca ar fi extends, nu super

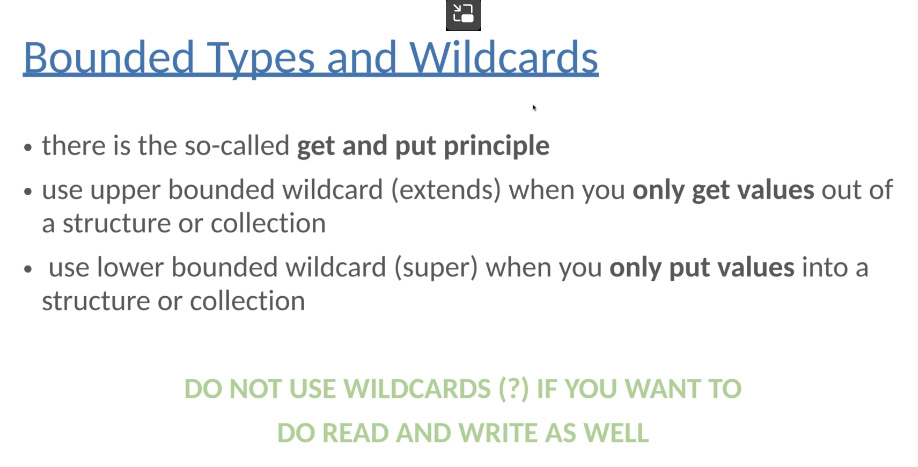
public static void main(String[] args) {  
 List<Integer> list = Arrays.*asList*(1,2,3,4);  
 *add*(list);  
}  
public static void add(List<? super Integer> list){  
 list.add(new Integer(100));  
}

**Dar metoda de copy() poate fi si asa:**

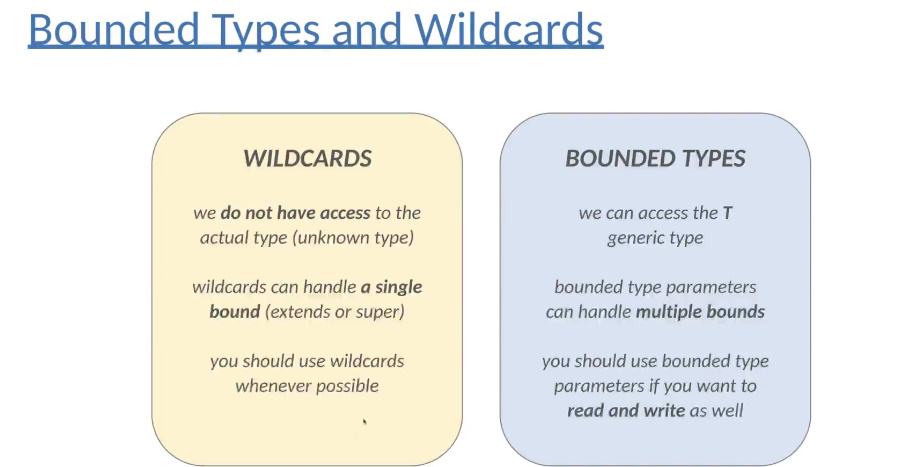
public static <T> void copy(List<T> from,List<T> to){  
 to.add(from.get(1));  
}

**WildCards vs bounded type parameters**

* Bounded Type parameters le folosim doar cand cream clasa/metoda, adica class Nume <T> sau <T> void func(), dar wildcard e folosit doar cand cream obiecte/ variabile, gen List<?> lista = new ArrayList<Integer>();



* Deci, daca vrem si sa cititm si sa inseram iteme intr-o colectie de ex, folosim bounded type parameters

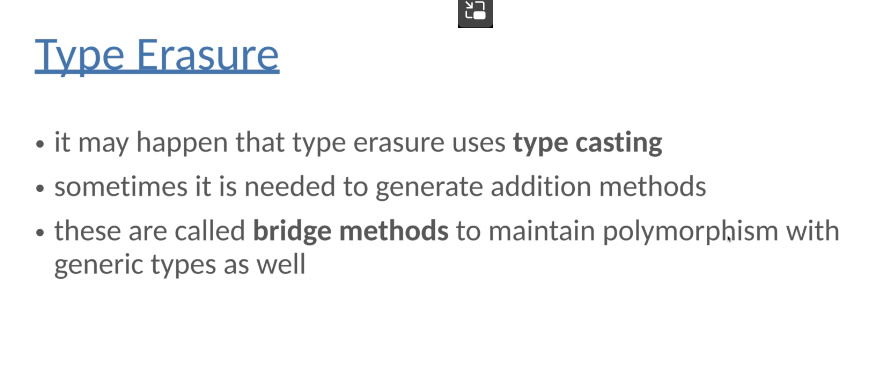


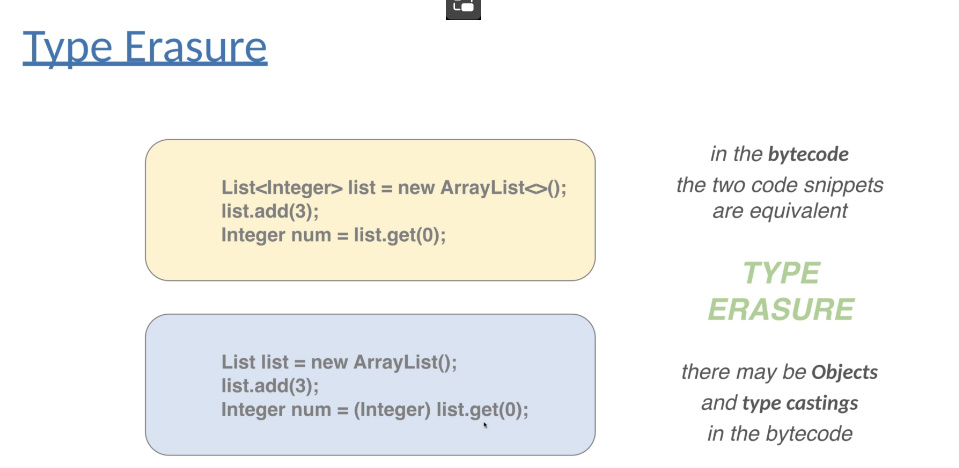
**<? extends Integer> sau <? super Integer> cu List<Integer>**

* Deci, ori ca avem extends Integer ori super Integer, putem sa avem o List<Integer>, nu e nicio problema.
* Ideea e ca daca avem <? extends Integer>, chiar de dam o List<Integer>, oricum nu vom putea adauga iteme, caci Java nu are de unde sti ca noi am dat o lista cu Integer
* daca avem <? super Integer>, si o List<Integer> vom putea doar adauga iteme, caci Java nu are de unde sti ca va fi Integer. Putem adauga liber iteme Integer

**Type Erasure**

* Java foloseste Type Erasure pentru a implementa generics, dar si wildcards
* El are grija ca la run time sa se inlocuiasca toate Generics cu tipurile stabilite adica bound(daca avem extends), sau daca niciun generics nu e specificat, atunci pune Object



* 
* **Interview: Este Generics mai rapid ca casting?**

**Raspuns: ambele au aceeasi viteza. Generics tot se bazeaza pe casting, deci generics tot fac casting, doar ca il fac in locul nostru.**

* Acum, cand avem new ArrayList<Integer>(), Java va pune Object in loc de T dar se va asigura ca mereu sa faca conversia (Integer) Object
* Daca avem new ArrayList(); Java tot va pune Object peste tot in loc de T, dar nu va face niciun cast automat.
* Type Erasure va face ca tot ce este in locul lui T sa fie inlocuit cu Object, si daca noi specificam ceva intre <>, sa faca type castingul mereu.
* De ex:
* class Shop<T>{  
   private T item1;  
   private T item2;  
   public T getItem1() {  
   return item1;  
   }  
    
   public void setItem1(T item1) {  
   this.item1 = item1;  
   }  
   public T getItem2() {  
   return item2;  
   }  
   public void setItem2(T item2) {  
   this.item2 = item2;  
   }  
  }

Va fi transformat de Java in:

class Shop{  
 private Object item1;  
 private Object item2;

public void setItem1(Object item1) {  
 this.item1 = item1;  
}  
public Object getItem2() {  
 return item2;  
}  
public void setItem2(Object item2) {  
 this.item2 = item2;  
}

}

Dar, cand se va returna vreun obiect sau se va adauga unul ca parametru, mereu se va face conversia in, de ex, (Integer), daca vom avea <Integer>

* Dar, daca avem vreun bound, atunci Object va fi inlocuit cu Bound:
* class Shop<T extends Number>{  
   private T item1;  
   private T item2;  
   public T getItem1() {  
   return item1;  
   }  
    
   public void setItem1(T item1) {  
   this.item1 = item1;  
   }  
   public T getItem2() {  
   return item2;  
   }  
   public void setItem2(T item2) {  
   this.item2 = item2;  
   }  
  }

class Shop{  
 private Number item1;  
 private Number item2;  
 public Number getItem1() {  
 return item1;  
 }  
  
 public void setItem1(Number item1) {  
 this.item1 = item1;  
 }  
 public Number getItem2() {  
 return item2;  
 }  
 public void setItem2(Number item2) {  
 this.item2 = item2;  
 }  
}

Deci, daca avem <T extends Number>, peste tot in loc de T nu se va mai pune Object, ci Number care e Bound, dar iarasi, in dependenta de ce vom pune in < > la crearea unui obiect se face castingul