CS61B Lecture #24	,	The Old Da	ays
Today: Java support for generic programming		• Java library types such as List didn't Lists were lists of Objects.	used to be parameterized. All
Readings for today: A Java Reference, Chapter 10.		 So you'd write things like this: 	
Readings for Monday: Data Structures, §6.4.		<pre>for (int i = 0; i < L.size (); i += 1) { String s = (String) L.get (i); }</pre>	
		• That is, must explicitly cast result of L.get (i) to let the compiler know what it is.	
		 Also, when calling L.add(x), was no ch into it. 	neck that you put only ${ m String} s$
		 So, newest release attempts to alleviate these perceived problems by introducing parameterized types, like List<string>.</string> 	
		• Unfortunately, it is not as simple as or	-
Last modified: Fri Oct 19 19:33:03 2012	CS61B: Lecture #24 1	Last modified: Fri Oct 19 19:33:03 2012	CS61B: Lecture #24 2
Basic Parameterization		Parameters on Methods	
Basic Parameterizatio	on	Parameters on A	Nethods
• From the definitions of ArrayList and Map ir	1 java.util:	 Functions (methods) may also be parar 	
	1 java.util:		neterized by type. Example of ITEM. */
 From the definitions of ArrayList and Map in public class ArrayList<item> implements List public Item get (int i) { } public boolean add (Item x) { } }</item> 	1 java.util:	 Functions (methods) may also be pararuse from java.util.Collections: /** A read-only list containing just static <t> List<t> singleton (T item)</t></t> In this case, compiler figures out T 	neterized by type. Example of ITEM. */) { } 7 without help when you call
• From the definitions of ArrayList and Map in public class ArrayList <item> implements List public Item get (int i) { } public boolean add (Item x) { } </item>	1 java.util:	• Functions (methods) may also be parar use from java.util.Collections: /** A read-only list containing just static <t> List<t> singleton (T item)</t></t>	neterized by type. Example of ITEM. */) { } (without help when you call of x.
<pre>• From the definitions of ArrayList and Map ir public class ArrayList<item> implements List public Item get (int i) { } public boolean add (Item x) { } } public interface Map<key, value=""> { Value get (Key x); }</key,></item></pre>	1 java.util: t <item> {</item>	 Functions (methods) may also be pararuse from java.util.Collections: <pre>/** A read-only list containing just static <t> List<t> singleton (T item)</t></t></pre> In this case, compiler figures out T singleton(x) by looking at the type of 	neterized by type. Example of ITEM. */) { } T without help when you call of x. pllections):
• From the definitions of ArrayList and Map in public class ArrayList <item> implements List public Item get (int i) { } public boolean add (Item x) { } } public interface Map<key, value=""> { Value get (Key x); </key,></item>	n java.util: t <item> { roduce formal <i>type pa</i>- in effect gets substi- om, Key, or Value when</item>	 Functions (methods) may also be pararuse from java.util.Collections: /** A read-only list containing just static <t> List<t> singleton (T item)</t></t> In this case, compiler figures out T singleton(x) by looking at the type of Another example (from java.util.Com) /** An unmodifiable empty list. */ 	neterized by type. Example of ITEM. */ } { } T without help when you call of x. pllections): . } t contain enough information,
 From the definitions of ArrayList and Map ir public class ArrayList<item> implements List public Item get (int i) { } public boolean add (Item x) { } } }</item> public interface Map<key, value=""> { Value get (Key x); }</key,> First occurrence of Item, Key, and Value intrameters, whose "value" (a reference type) tuted for all the other occurrences of Ite ArrayList or Map is "called" (as in ArrayList 	n java.util: t <item> { roduce formal type pa- in effect gets substi- em, Key, or Value when <string>, or ArrayList<int[]>,</int[]></string></item>	 Functions (methods) may also be pararuse from java.util.Collections: /** A read-only list containing just static <t> List<t> singleton (T item)</t></t> In this case, compiler figures out The singleton(x) by looking at the type of singleton(x) by looking at the type of the static <t> List<t> emptyList.*/static <t> List<t> emptyList() {</t></t></t></t> Here, a call to emptyList() would no so instead we write, e.g., Collections 	neterized by type. Example of ITEM. */ } { } T without help when you call of x. pllections): . } t contain enough information,
 From the definitions of ArrayList and Map in public class ArrayList<item> implements List public Item get (int i) { } public boolean add (Item x) { } } }</item> public interface Map<key, value=""> { Value get (Key x); }</key,> First occurrence of Item, Key, and Value intrameters, whose "value" (a reference type) tuted for all the other occurrences of Ite ArrayList or Map is "called" (as in ArrayList< or Map<string, list<particle="">>).</string,> Can also say that you don't care what a ty 	n java.util: t <item> { roduce formal type pa- in effect gets substi- em, Key, or Value when <string>, or ArrayList<int[]>, epe parameter is (wild- X. */</int[]></string></item>	 Functions (methods) may also be pararuse from java.util.Collections: /** A read-only list containing just static <t> List<t> singleton (T item)</t></t> In this case, compiler figures out The singleton(x) by looking at the type of singleton(x) by looking at the type of the static <t> List<t> emptyList.*/static <t> List<t> emptyList() {</t></t></t></t> Here, a call to emptyList() would no so instead we write, e.g., Collections 	neterized by type. Example of ITEM. */ } { } T without help when you call of x. pllections): . } t contain enough information,

Type Bounds

Type	Bounds	(II)
------	--------	------

 Sometimes, your program needs to ensure that a particular type parameter is replaced only by a subtype (or supertype) of a particular type (sort of like specifying the "type of a type."). For example, <pre>class NumericSet<t extends="" number=""> extends HashSet<t> { /** My minimal element */ T min () { } } </t></t></pre> Requires that all type parameters to NumbericSet must be subtypes		And one more: /** Search sorted list L for KEY, returning either its position (if * present), or k-1, where k is where KEY should be inserted. */ static <t> int binarySearch(List<? extends Comparable<? super T>> L, T key) Here, the items of L have to have a type that is comparable to T's or some supertype of T. Does L have to be able to contain the value key? Why does this make sense?</t>	
of Number (the "type bound"). T can eit bound, as appropriate.			
• Another example: /** Set all elements of L to X. */ static <t> void fill (List<? super T></t>			
means that L can be a List <q> as long or implements) Q.</q>	as T is a subtype of (extends		
Last modified: Fri Oct 19 19:33:03 2012	C561B: Lecture #24 5	Last modified: Fri Oct 19 19:33:03 2012	CS61B: Lecture #24 6
Dirty Secrets Behind the Scenes		Limitations	
Dirty Secrets Behind	the Scenes	Limitation	ns
 Dirty Secrets Behind Java's design for parameterized types for backward compatibility. 		Limitatior Because of Java's design choices, are so gramming:	
• Java's design for parameterized types		Because of Java's design choices, are so	ome limitations to generic pro-
 Java's design for parameterized types for backward compatibility. Actually, when you write class Foo<t> {</t> 	s was constrained by a desire	Because of Java's design choices, are so gramming:	ome limitations to generic pro- ally the same,
 Java's design for parameterized types for backward compatibility. Actually, when you write 		Because of Java's design choices, are so gramming: • Since all kinds of Foo or List are rea	ome limitations to generic pro- ally the same, De true when L is a List <integer>.</integer>
 Java's design for parameterized types for backward compatibility. Actually, when you write <pre>class Foo<t> { T x; T mogrify (T y) { }</t></pre> 	s was constrained by a desire Foo <integer> q = new Foo<integer>();</integer></integer>	Because of Java's design choices, are so gramming: • Since all kinds of Foo or List are rea - L instanceof List <string> will b - Inside, e.g., class Foo, you cannot</string>	ome limitations to generic pro- ally the same, be true when L is a List <integer>. write new T (), new T[], or x</integer>
 Java's design for parameterized types for backward compatibility. Actually, when you write class Foo<t> { T x; T mogrify (T y) { } }</t> Java gives really gives you class Foo { Object x; Object mogrify (Object y) { } } 	<pre>Foo<integer> q = new Foo<integer>(); Integer r = q.mogrify (s); Foo q = new Foo(); Integer r = (Integer) q.mogrify ((Integer) s);</integer></integer></pre>	 Because of Java's design choices, are so gramming: Since all kinds of Foo or List are rea L instanceof List<string> will b</string> Inside, e.g., class Foo, you cannot instanceof T. 	ome limitations to generic pro- ally the same, be true when L is a List <integer>. write new T (), new T[], or x be parameters. rrayList<integer>.</integer></integer>
 Java's design for parameterized types for backward compatibility. Actually, when you write <pre>class Foo<t> { T x; T mogrify (T y) { } } Java gives really gives you class Foo { Object x; Object mogrify (Object y) { } } </t></pre> 	<pre>Foo<integer> q = new Foo<integer>(); Integer r = q.mogrify (s); Foo q = new Foo(); Integer r = (Integer) q.mogrify ((Integer) s); cally, and also throws in some that all those casts will work,</integer></integer></pre>	 Because of Java's design choices, are so gramming: Since all kinds of Foo or List are readed to the constance of List<string> will be a string of the constance of the constanc</string>	ome limitations to generic pro- ally the same, be true when L is a List <integer>. write new T (), new T[], or x be parameters. rrayList<integer>. I unboxing makes this substitu-</integer></integer>