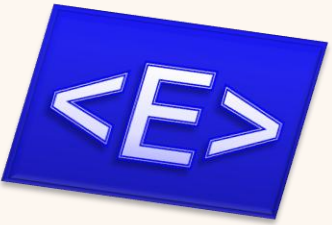
Introduction to Object Oriented Programming

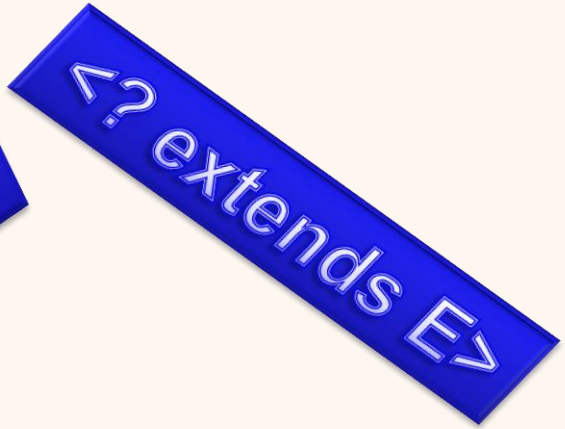
(Hebrew University, CS 67125 / Spring 2014)

Lecture 10

Generics



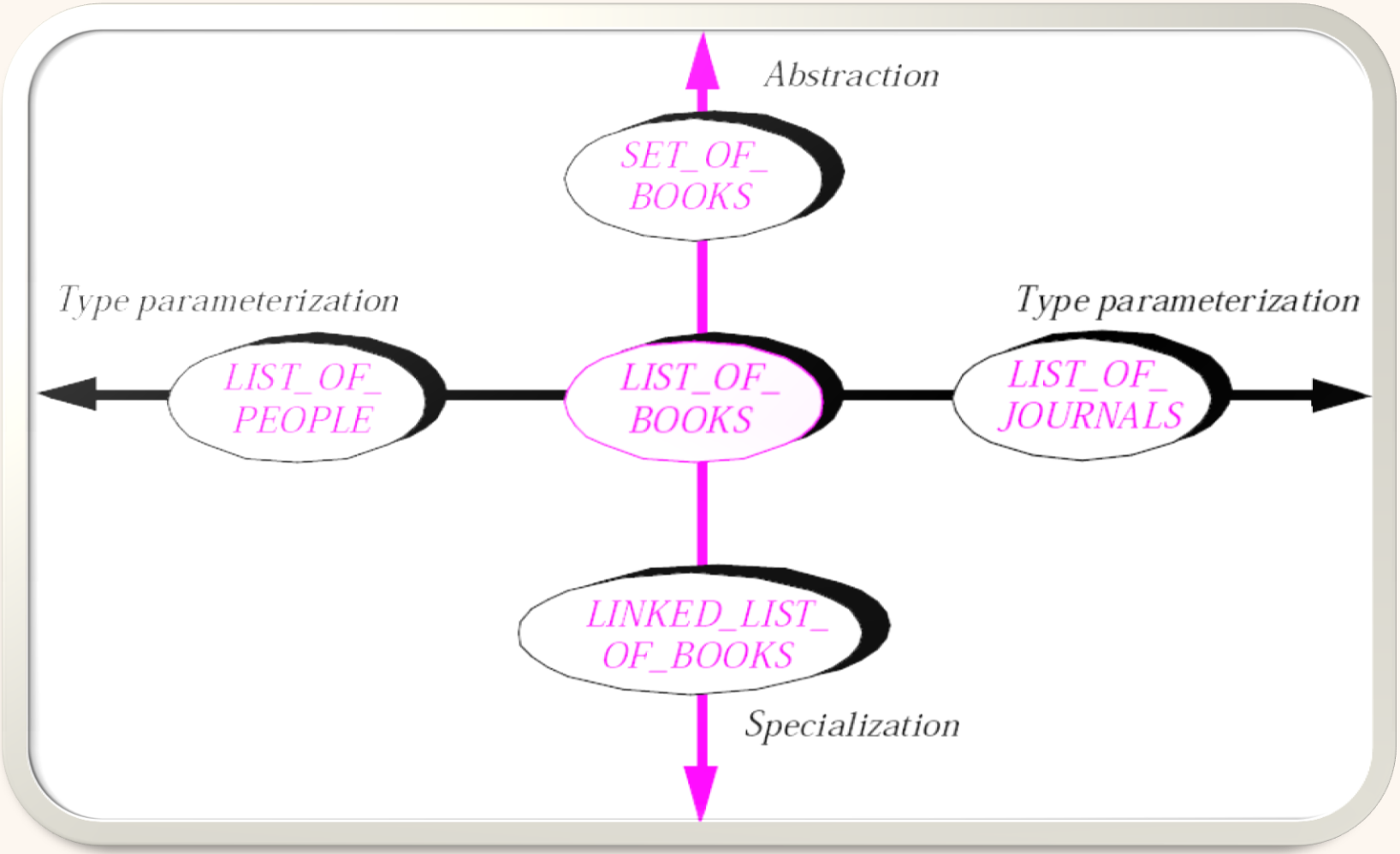
Generic Wildcards



Reminder: What are Generics?

- Generics **abstract** over **non-primitive types**
  - Classes or arrays (including primitive arrays)
- *Classes*, *interfaces* and *methods* can be **parameterized** by types
- Generics provide increased **readability** and **type safety**
  - Very useful when working with **collections**

Introduction to Genericity

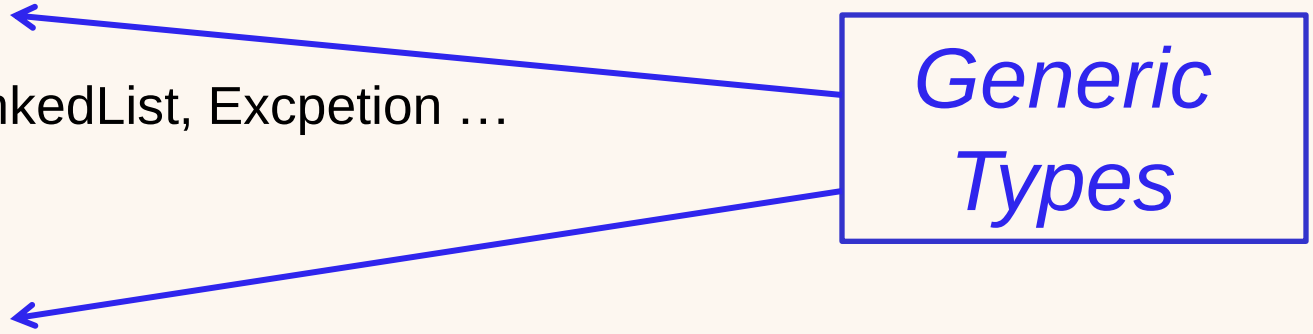


Generics: Prevention More than Cure

- The key to software reliability is **prevention more than cure**
- The cost of correcting an error grows astronomically when the time of detection is delayed
- Generics enables the early detection of **type errors**, hence it is a fundamental tool in the quest for reliability

What is a Type?

- Primitives
  - int, double, boolean …
- Classes
  - String, LinkedList, Excpetion …
- Arrays
  - int[], String[], …



What is a Type Error?

- `String a = new Integer();`

Compile-time type error

- `Object i = new Integer(5);`
- `String b = (String)i;`

run-time type error

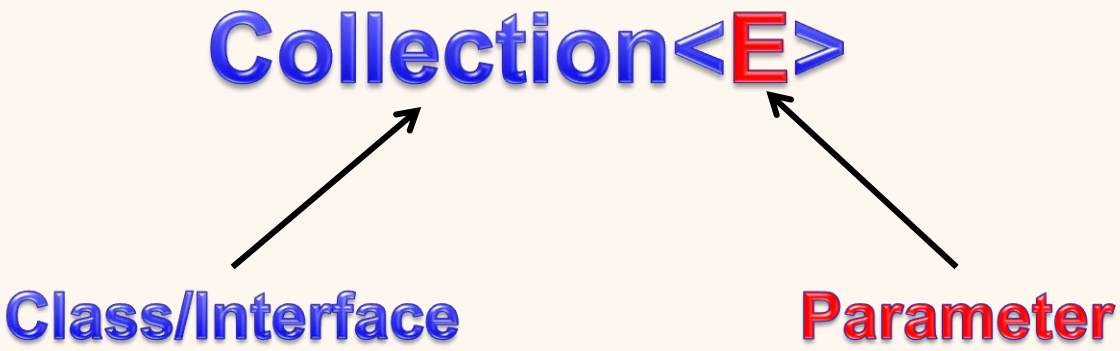
Which one is easier to detect?

Type Safety

- A **type safe** program is a program in which type errors are compile-time errors and not run-time errors

- Generics is a way of ensuring **type-safety**

Generics Example



Why Generics?

Before:

```
/**
 * My stamp collection. Contains only Stamp instances.
 */
private final Collection stamps = ... ;
```

```
// Erroneous insertion of coin into stamp collection
stamps.add(new Coin( ... ));
```


```
// Now a raw iterator type - don't do this!
for (Iterator i = stamps.iterator(); i.hasNext(); ) {
    Stamp s = (Stamp) i.next(); // Throws ClassCastException
    ... // Do something with the stamp
}
```

Why Generics?

After:

```
// Parameterized collection type - typesafe
private final Collection<Stamp> stamps = ... ;
```

```
Test.java:9: add(Stamp) in Collection<Stamp> cannot be applied
to (Coin)
    stamps.add(new Coin());
                   ^
```

```
// for-each loop over a parameterized collection - typesafe
for (Stamp s : stamps) { // No cast
    ... // Do something with the stamp
}
```

Making a Class Generic

Take 1

Node before Generics

```
public class Node {
    private Object elem;

    public Node(Object elem) {
        this.elem = elem;
    }

    public Object data() {
        return elem;
    }

    …
}
```

Node with Generics

```
public class Node<T> {
    private T elem;

    public Node(T elem) {
        this.elem = elem;
    }

    public T data() {
        return elem;
    }

    …
}
```

Using a Generic Class

Node before Generics

```
Node n = new Node("hello");


...

String s = (String) n.data();
```

Node with Generics

```
Node<String> n =
    new Node<String>("hello");


...

String s = n.data();
```

Using a Generic Class

Before Generics

```
LinkedList list = new LinkedList();


list.add("hello");

String s = (String) list.get(0);

list.add(new Integer(5));
String s = (String) list.get(1);
// Run-time error
```

With Generics

```
LinkedList<String> list =
        new LinkedList<String>();

list.add("hello");

String s = list.get(0);

list.add(new Integer(5));
// Compilation error
```

Generic Parameters

- A generic parameter can only be defined in a class definition

```
public class Node<T>
```

  - (See exception in TA session)

- Inside the class body, T is not redefined, but is used as a concrete value

```
private T elem;
```

- This is true even if T is used as a concrete parameter for a generic **object**

```
private Node<T> next;
```

Making a Class Generic

Node before Generics

```
public class Node {
    private Object elem;
    private Node next;


    public Node(Object elem) {
        this.elem = elem;
        this.next = null;
    }

    public Object data() {
        return elem;
    }

    …
}
```

Node with Generics

```
public class Node<T> {
    private T elem;
    private Node<T> next;


    public Node(T elem) {
        this.elem = elem;
        this.next = null;
    }

    public T data() {
        return elem;
    }

    …
}
```

Not the same meaning of <T>

Generic are Invariant

- For any two distinct types Type1 and Type2: List<Type1> is **neither** a **subtype** nor a **supertype** of List<Type2>
  - **Invariance**

- This means, for example, that LinkedList<String> **doesn't** extend LinkedList<Object>
  - **// Compilation error**
  - LinkedList<Object> = **new** LinkedList<String>();

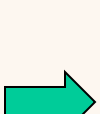
So Far…

- Generics are abstractions over types
- Generics help to detect type errors in compile-time
- Syntax: “Sequence<E>”
- Generics are invariant

Wildcards Overview

List<?> list

Object Creature Animal Dog MyDog



Retrieves only Object

Cannot add anything

List<?> list

- If you want to use a generic type but you don't **know** or **care** what the actual type parameter is, you can use a question mark instead
- You can't assume **anything** about the type of the objects that you get out of a List<?>
- You can't put any element (other than **null**) into a List<?>

<?> – Why?

1) Object is enough

```
void printList(List<?> c) {
        for (Object e : c) {
                System.out.println(e);
        }
}
```

```
public void removeAll(List<?> list) {
        Iterator<?> e = list.iterator();
        while (e.hasNext()) {
                e.remove();
        }
}
```

<?> – Why? (cont'd)

2) Even object isn't needed

```
public boolean isEqualSizes(List<?> c1, List<?> c2) {
        return c1.size() == c2.size();
}
```

3) Force constraints on a list

```
/**
  * …
  * This method will not add anything to c1 (except null), and
  * will only retrieve Objects from it.
  */
public void immutableMethod(List<?> c1) {
        …
}
```

<?> – Why? (cont'd)

4) Method returns list of unknown type

```
private List<?> generateListFromUserInput(String str) {
        if (str.equals("String")) {
                return new LinkedList<String>();
        } else if (str.equals("Integer") ){
                return new LinkedList<Integer>();
        } else {
                return new LinkedList<Object>();
        }
}
```

List<?> Alternatives?

- Why not use List<*Object*>?

```
void printList(List<Object> c) {
   for (Object e : c)
         System.out.println(e);
}
```

- Say we wanted to print a list of *String*s

```
LinkedList<String> list = new LinkedList<String>();
…
printList(list);    // ERROR: Generics are invariant:
                    // LinkedList<String> does not extends LinkedList<Object>
```

- Why not use non-generic List?
  - List<?> is **type-safe**!

List<?> Properties

- Why can’t we add anything?
  - The real type of *list* can be any class, so there is no way of knowing which objects we are allowed to add
- Why can we only retrieve *Object*s?
  - As each class extends *Object*, we can always retrieve an *Object*
  - However, for the reasons above, we cannot assume anything more about the real type of the content of *list*

List<?> Properties (cont'd)

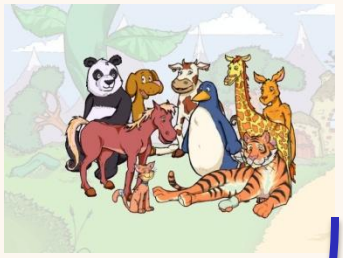
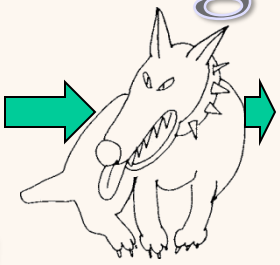
- Only **references** (not actual data structures) can use wildcards
- A *concrete* object must have an *actual* parameter
  - `LinkedList<?> c1 = new LinkedList<String>();` // Legal
  - `LinkedList<?> c2 = new LinkedList<?>();` // Illegal
- Wildcards work with polymorphism
  - `List<?> c3 = new LinkedList<String>();`

Wildcards Overview

List<? extends Animal> animals;

Object Creature Animal Dog MyDog




Retrieves up to Animal

Can only add null

List <? extends Animal> animals

- As generic classes are **invariant**, polymorphism cannot be accomplished by generic classes

```
// Compilation error!
LinkedList<Animal> l = new LinkedList<Dog>();
```

- In order to achieve polymorphism, we must use **wildcards**

```
// Now it works!
LinkedList<? extends Animal> l = new LinkedList<Dog>();
```

<? extends Animal> – Why?

1) Do some subclass-specific operations

```
void printAnimals(List<? extends Animal> c) {
        for (Animal e : c) {
            e.printEars();
            e.printNose();
            e.printRestOfBody()
        }
}
```

2) Use shared interface

```
void jumpAnimals(List<? extends Jumpable> c) {
        for (Jumpable e : c) {
            e.jump();
        }
}
```

<? extends Animal> Properties

List<? extends Animal> myList;

- Can be initialized with any parameter that extends Animal (including Animal)
  - myList = **new** LinkedList<Animal>(); // Ok
  - myList = **new** LinkedList<Dog>(); // Ok
  - myList = **new** LinkedList<Creature>(); **// Compilation error**
- Can only add **null**
  - myList.add(**new** Animal()); **// Compilation error**
  - myList.add(**new** Dog()); **// Compilation error**
  - myList.add(**null**); // Ok

<? extends Animal> Properties

- Can retrieve up to animal
  - Animal a = myList.get(0); // Ok
  - Creature c = myList.get(0); // Ok
  - Dog d = list.get(0); **// Compilation error**

Erasure

- Erasure is the process that allows generic types to interoperate freely with legacy code that does not use generics

- Erasure erases all generics type argument information at compilation phase
  - List<String> is converted to List
  - String t = stringList.iterator().next() is converted to
    String t = **(String)**stringList.iterator().next()

- As part of its translation, a compiler maps every parameterized type to its type erasure

Erasure

- What is the output of the following code?
  - ArrayList <String> l1 = **new** ArrayList<String>();
  - ArrayList<Integer> l2 = **new** ArrayList<Integer>();
  - System.*out*.println(l1.getClass() == l2.getClass());

- **true**
  - getClass() returns “ArrayList“ for both lists

Erasure Process Implications

- An implication of the erasure process is that it makes no sense to ask an object if it is an instance of a particular invocation of a generic type

  c1 ***instanceof*** *LinkedList<String>*

- Also, casting to generic types does not make sense
  - The actual runtime type of a variable of type *List*<String> is *List*
  - Trying to cast to *List*<String> will generate a compile warning

Generics and Arrays

- Unlike Generics, arrays are **covariant**: if **Sub** is a subtype of **Sup**, then the array type **Sub**[ ] is a subtype of **Sup**[ ]

- However, we cannot define an array of generic objects
  - List**<String>[ ]** list = …; **// Compilation error**

Generics and Arrays

- If you were allowed to declare arrays of generic types:

```
ArrayList<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
ArrayList<String>[ ] lsa = new ArrayList<String>[10];  // illegal
// Assuming the previous line was legal:
Object[ ] oa = lsa;        // ArrayList<String> is a subtype of Object
oa[0] = li;                // ok, because oa is an array of Objects
// Now, oa[0] (and consequently, lsa[0]), is of type ArrayList<Integer>
String s = lsa[0].get(0); // ClassCastException (putting Integer into String)
```

Do Generics Affect My Code?

- They don’t – except for the way you’ll code!
- Non-generic code can use generic libraries
- For example, existing code will run unchanged with generic Collection library

Links

- Generics Tutorial by Gilad Bracha
  http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

- Angelika Langer Generics FAQ:
  http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html

- **Java Generics and Collections** by Maurice Naftalin, Philip Wadler (in the library)

- **Effective Java** by Josh Bloch
  http://www.infoq.com/resource/articles/bloch-effective-java-2e/en/resources/Bloch_Ch05.pdf

So Far…

- Generics
  - Type safety
  - Readability
- Wildcards
  - List<?>
  - Allow type-safety even for unknown types
  - Allows generics pseudo-polymorphism
- Erasure