Java Generics
Writing flexible class

From previous lecture:

class ListNode {
    ListNode( Object theElement ) { ... }

    Object element;
    ListNode next;
}

Problem with Object

class Dancer { }

class HipHopDancer extends Dancer { }

class Bboy extends HipHopDancer {
    public void toprock() { }
}

class Popper extends HipHopDancer {
    public void pop() { }
}
Problem with Object

// inside some method:
ListNode node = new ListNode(new Popper());
Bboy hong10 = (Bboy) node.element;
hong10.toprock(); // java.lang.ClassCastException at runtime!
Generics to the Rescue

class ListNode <T> {
    ListNode( T theElement ) { ... }
    T element;
    ListNode <T> next;
}

Here, T is a type parameter that must be specified during instantiation and can be substituted with any non-primitive type.
Generics to the Rescue

// inside some method. This no longer compiles!
ListNode<Popper> node =
    new ListNode<Popper>(new Popper());
Bboy hong10 = (Bboy) node.element; // compile error here
hong10.toprock();

Here, you get the benefit from the compiler to lookout for these unsafe casts for you.
class SimpleCopier<T> {
    private T copy;

    public void copy(ListNode<T> orig) {
        copy = orig.getElem();
    }

    public void copyTo(ListNode<T> dup) {
        dup.elem = copy;
    }
}
Type Bounds

SimpleCopier<HipHopDancer> copy = new SimpleCopier<HipHopDancer>();

ListNode<Bboy> node = new ListNode<Bboy>(new Bboy());
ListNode<Dancer> danceNode = new ListNode<Dancer>(new Dancer());

/* Compile error on these 2 lines because Bboy does not exactly match HipHopDancer and Dancer does not exactly match HipHopDancer. */
copy.copy(node);
copy.copyTo(danceNode);
Type Bounds

Solution to the previous problem: use wildcards (?) and bounds extends and super.

- ? extends T - matches any type that extends T, including T. Note that T can be an interface.

- ? super T - matches any type that is a T's ancestor, including T.
class Copier<T> {
    private T copy;

    public void copy(ListNode<? extends T> orig) {
        copy = orig.getElem();
    }

    public void copyTo(ListNode<? super T> dup) {
        dup.elem = copy;
    }
}
Type Bounds

Copier<HipHopDancer> copy = new Copier<HipHopDancer>();

ListNode<Bboy> node = new ListNode<Bboy>(new Bboy());
ListNode<Dancer> danceNode = new ListNode<Dancer>(new Dancer());

// No more errors!
copy.copy(node);
copy.copyTo(danceNode);
Dissecting Type Bounds

```java
public void copy(ListNode<? extends T> orig) {
    copy = orig.getElem();
}
```

In the example, Node<\texttt{Bboy}> was passed so, \texttt{getElem()} has a return type \texttt{Bboy}. \texttt{copy} is type \texttt{T}, which is \texttt{HipHopDancer} in the example. Assignment of \texttt{Bboy} to \texttt{HipHopDancer} type is legal so this works. If \texttt{extends} is replaced with \texttt{super}, compile error will occur because you can't assign a \texttt{Dancer} instance (which the super wildcard would match) to a \texttt{HipHopDancer}.
public void copyTo(ListNode<? super T> dup) {
    dup.elem = copy;
}

Similarly, dup.elem is a wildcard type, which in our example, can match Dancer or HipHopDancer. And once again, copy is of type HipHopDancer. This is legal because you can assign a HipHopDancer to a Dancer. And if super is replaced with extends, compile error will occur because you can't assign a HipHopDancer to a Bboy instance (which the extends wildcard would match).
Dissecting Type Bounds

Simple mnemonic (from Effective Java Book by Joshua Bloch) to remember when to use super or extends:

**PECS for Producer-extends, consumer-super**

- Producer means an entity that gives the data. Usually at the right hand side of the assignment or being passed as a parameter.

- Consumer means an entity that receives the data. Usually at the left hand side of the assignment or accepts a parameter.
Type Erasure

- Mechanism that erases all generic type parameters during compilation and type casts will be automatically inserted by the compiler if necessary.

- This means that the type parameters **should not** be treated as simple substitution unlike the templates in C++.

- Existed only for historical purposes - for back compatibility for old Java codes that uses raw classes/interface (like the Java Collections and Comparable) before the Generics was introduced.
Important Readings from Weiss Text

- 1.4.2 Wrappers for Primitive Types
- 1.5.2 Autoboxing/Unboxing
- 1.5.4 Generic Static Methods
- 1.5.7 Restrictions on Generics