CS 211
GENERICS

Generics

Generics

What problem does using generics solve?

What do they look like?

What do they mean?

Problem: "lost" types

`java.util.ArrayList` provides a nice suite of methods

- like our old friend List from Python: `add()`, `insert()`, etc.
- If we use `ArrayList` without generics (a "raw type"), we only know that `Object`s were stored in it:

```
ArrayList alist = new ArrayList();
Person bob = new Person("bob",89);
alist.add(bob);

// can't do this: Objects stored, need a Person.
Person p = alist.get(0);
```

→ compile-time: found Object, needed Person.

```
// must cast, every time:
Person p = (Person) alist.get(0);
```

Problem: "lost" types

This issue would happen all over:

```
ArrayList personList = new ArrayList();
// add many Person objects

// NOT ALLOWED:
for (Person p : personList) {
    p.whatever();
}

// allowed, but annoying/error-prone
for (Object p : personList){
    ((Person) p).whatever();
}
```

Generics: Establish & Remember Types

Generics allow us to define **type parameters** – we can parameterize blocks of code with types!

Where can we add type parameters?
→ **at class declarations** (available for entire class definition)
→ **at method signatures** (available through just this method)

Instead of **just** having the values parameter list,
we can **also** give a type parameters list

- type params may be types of value parameters

Declaring Generic Types: Classwide

We can add a generic type to a **class definition**:

```
public class Box <T> {

   // T can be anywhere: like field types.
   public T value;

   public Box(T t) {// T used as parameter type
      this.value = t;
   }

   // T used as return type and param type
   public List<T> copies(T v, int n) { … }
}
```

Generic Types: Some Notes

All values have a type. It's the set of values we can store.

→ we declare our **value-holding variables**
with a name and a type.

All types have a kind. It's the set of types we can use.

→ all Java reference types are the same kind, so it's assumed
→ we declare our **type parameters**
with just a name.

Generics Example: Pairs (2-tuples)

```
public class Pair <A,B> {
  public A t1;
  public B t2;

  public Pair(A t1, B t2){
    this.t1 = t1;
    this.t2 = t2;
  }

  public String toString(){
    return ("("+t1+","+t2+")");
  }
}
```

```
Pair<Integer,String> ns = new Pair<Integer,String>(5,"a");
ns.t1 = 10;
System.out.println(ns);
```

Declaring Generic Types: Method-level

We also declare generic types for just one method, like **<U>**:

```
public class Foo {
    …
    public <U> U choose (U u1, U u2, boolean b) {
        return (b ? u1 : u2);
    }
}
```

- declaration is before return type.
- It may be the return type, param type, and in method body
- All we know about u1 or u2 is that it is a value of the U type.

→ that's not much info! Enough for useful/reusable code

- Let's look back at **`ArrayList`**

Generics Example: Methods

Given a generic method (which happens to be static):

```
public static <U> U choose (U u1, U u2, boolean b) {
  return (b ? u1 : u2);
}
```

We instantiate the parameters and can call it like this:

```
String s = Foo.<String>choose("yes","no",true);
String t = Foo.choose("yes","no",true);
```

If it were non-static, we'd need an object to call it:

```
Foo f = new Foo();
String s = f.<Integer>choose(5, 3, true);
String t = f.choose(5, 3, true);
```

(We only require the type when it's not clear from the params)

Generics Example: ArrayList

```
public class ArrayList<E> {
  private int size;
  private E[] items;
  public ArrayList(){
  items = new E[10];
  size = 0;
}
public E get(int i)          { return items[i]; }
public void set(int i, E e) { items[i] = e;    }

public void add (E e) {
  if (size>=items.length) {
    E[] longer = new E[items.length*2];
    for (int i=0; i<items.length; i++){
      longer[i] = items[i];
    }
    items = longer;
```

almost!

Example: Using ArrayList Generically

Let's look at how we actually get to use generics with ArrayList:
→ we need to **instantiate** the class's type parameter:

```
//instantiate the type parameter with <>'s:
ArrayList<String> slist = new ArrayList<String>();

//now use all methods without having to specify again.
slist.add("hello");
slist.add("goodbye");
String elt = slist.get(0);

System.out.println("first:  " + elt);
System.out.println("entire: " + slist);
```

We instantiate it both in the variable's type declaration as well as in the constructor call.

Advanced Generics (going further)

- We can **place a Bound** on a generic parameter:

  `<T extends SomeType>`

- here, `extends` can actually mean:
  - `T` is a sub-class of `SomeType`
  - `T` implements the `SomeType` interface. (confusing, yes.)
    → useful to restrict the types at which we can use `T`
  - instead of any types, it must provide some view `T.`
- add multiple views: `<T extends A & B & C>`

we can make multiple unrelated view-claims at once this way

Generics and Java Collections

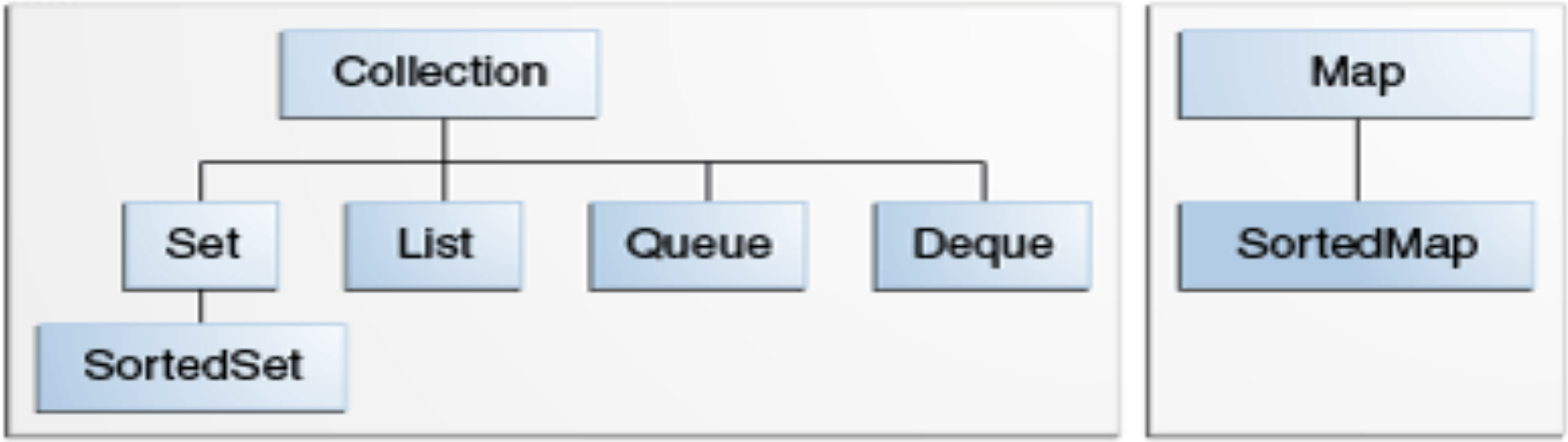
- Many collections use Java Generics.
  → e.g., a list of what? A set of what? Maps from what to what?
- Instantiate the given type parameter(s) to what you want contained by that collection.

  ex: `ArrayList<Integer> xs = new ArrayList<Integer>();`

- use those methods, and Java knows this ArrayList holds only Integers.
  → no casting down from Object.

Java Collections: Many Interfaces, Many More Implementations

Some interfaces of the Java Collections[†]:



List Interface: Some Implementations

Visit the API for these classes that implement the **List<E> interface:**

ArrayList

→ uses arrays to provide the list operations

→ some operations are faster/slower as a result.

LinkedList

→ individual nodes each pointing to neighbors

→ different operations are faster/slower as a result.

Comparing ArrayList and LinkedList

Run building/navigating operations and time each implementation to see where each style performs better/worse.