

Lecture 4

Generic Types

Object is the most “generic” type

```
public class ObjectPair {  
    Object first;  
    Object second;  
    public ObjectPair(Object a, Object b) {  
        first = a;  
        second = b;  
    }  
    public Object getFirst() { return first; }  
    public Object getSecond() { return second; }  
}
```

Reference variable of
type Object – the
superclass of anything


```
ObjectPair bid = new ObjectPair("ORCL", 32.07);  
Object o = bid.getFirst();  
String stock = (String) o;
```

casting: Object to String

Danger! Might be not String → Run-Time error

Object is the most “generic” type

```
public class ObjectPair {  
    Object first;  
    Object second;  
    public ObjectPair(Object a, Object b) {  
        first = a;  
        second = b;  
    }  
    public Object getFirst() { return first; }  
    public Object getSecond() { return second; }  
}  
  
ObjectPair bid = new ObjectPair("ORCL", 32.07);  
Object o = bid.getFirst();  
if (o instanceof (String))  
    String stock = (String) o;
```



We can check, of course

Parametrized types

```
public class Pair<A,B> {  
    A first;  
    B second;  
    public Pair(A a, B b) {  
        first = a;  
        second = b;  
    }  
    public A getFirst() { return first; }  
    public B getSecond() { return second; }  
}  
  
Pair<String,Double> bid = new Pair<>("ORCL", 32.07);  
String stock = bid.getFirst();  
double price = bid.getSecond();
```

No casting is required
Safe use of types
Checked by compiler

Generics

```
public class Pair<A,B> {  
    A first;  
    B second;  
    public Pair(A a, B b) {  
        first = a;  
        second = b;  
    }  
    public A getFirst() { ...}  
    public B getSecond() { ... }  
}
```

- We define **formal** types using variables: letters A, B

Generics

```
public class Pair<A,B> {  
    A first;  
    B second;  
    public Pair(A a, B b) {  
        first = a;  
        second = b;  
    }  
    public A getFirst() { ...}  
    public B getSecond() { ... }  
}
```

```
Pair<String,Double> bid  
    = new Pair<>("ORCL", 32.07);
```

- We define formal types using variables: letters A, B
- We substitute these formal letters with actual data types when we create an instance of a generic class

Generics

```
public class Pair<String,Double>
{
    String first;
    Double second;
    public Pair(String a,
                Double b) {
        first = a;
        second = b;
    }
    public String getFirst() {...}
    public Double getSecond() {...}
}

Pair<String,Double> bid
    = new Pair<>("ORCL", 32.07);
```

- We define formal types using variables: letters A, B
- We substitute these formal letters with actual data types when we create an instance of a generic class
- Each formal type letter is replaced with an actual data type throughout the entire class

The benefits of Generic types

- Type safety which is checked during compilation, and not during run time
- Collections:
 - Without generics we can create a collection of Objects – but this will allow elements of heterogeneous types to be mixed together
 - Using generics we ensure that all the slots in the collection are of the same type *T*. This will allow to use the methods defined in the corresponding class
- Algorithms:
 - We can implement an algorithm which works with different types of data. We do not write a special implementation for integers, doubles, Strings ...

Example 1: generic array

- Array of Objects:

```
Object A [] = new Object [10];
```

- We can place into A objects of any type, for example objects of type String
- However we can also place objects of any other type alongside Strings

- Generic array:

```
T A [] = new T [10];
```

- We want the objects in our collection to be related somewhat – to have the same type T
- Using generics, we restrict the types of entries stored in our array

Building an array of a general type

- Java requires that the type of elements in the array will be specified when the array is created
- So the declaration on the previous slide would not compile:

```
T A [] = new T [10] ;
```

- We will create a class that mimics a simple Java array but allows to store the elements of an arbitrary (and uniform) type

Specs of our generic array

- The **capacity** of the array (the number of slots) is determined when the array is created
- We want the type of elements to be **any Java type**
- However, it should be homogeneous – cannot mix types
- We want to be able to tell the **size** (the actual number of elements) in the array
- As in a normal (not generic) array:
 - We want to be able to assign a value at a given index
 - We want to be able to retrieve a value from a specified location

Array storing homogeneous things

```
public class MyArray<T> {  
    private T [ ] theArray;  
  
    public T get(int i) {  
        return theArray[i];  
    }  
  
    public void set(int i, T data) {  
        theArray[i] = data;  
    }  
}
```

- <T> is a formal type parameter
- We reference type T through the class body
- <T> is just a placeholder for the future data type
- All Ts will be substituted with an actual type during compilation

Can we store primitives in *MyArray*?

```
MyArray<String> S = new MyArray<String>(5) ;  
MyArray<MyRectangle> R= new MyArray<MyRectangle>(3) ;
```

- The actual types that can substitute type parameters <T> MUST be of reference type (Think: why?)
- Primitive types, such as int, boolean, and float are not allowed
- Fortunately, Java provides wrapper classes for each of the primitive types:

```
MyArray<Integer> I = new MyArray<Integer>(10) ;
```

Using MyArray

- Now you can substitute T with any Java reference type
- As we wanted: the data is of an arbitrary type, but still of the same type throughout the array

```
public class MyArray<T> {...}
```

```
MyArray<String> S = new MyArray<String>(5) ;
```



MyArray: constructor

```
@SuppressWarnings("unchecked")  
T [] temp = (T []) new Object[size];  
theArray = temp;
```

- Note how the array is created
 - We make an array of type *Object* and cast it to *T*
 - This is necessary due to Java type rules for arrays

Demo

See [MyArray.java](#) and [Example4.java](#)

Example 2: generic sorting

- Let's look at a simple sorting algorithm: **Selection sort**
 - Find smallest, swap into location 0
 - Find next smallest, swap into location 1, etc.
- Say, we implemented this algorithm for an array of integers

```
public static void selectionSort (int[] a, int n)    {  
    for (int i = 0; i < n - 1; i++)                {  
        int iNextSmallest = getIndexOfSmallest(a, i, n-1);  
        swap(a, i, iNextSmallest);  
    }  
}
```

Sorting objects of any type

- What if we want to sort an arrays of doubles? Array of Strings? Array of Dogs?
 - We would need to write a different method for each type!!!
 - We can compare numbers, Strings using < sign.
 - But what about people, grades etc?
- Can we write a single method that can sort **anything**?
 - We can't really sort “anything”, but we can write a method that can sort any *Comparable* objects

Comparable<T> interface

- Consider the *Comparable<T>* interface:
- It contains one method:

```
int compareTo (T another) ;
```
- Returns:
 - a negative number if the current object is less than another
 - 0 if the current object equals another
 - a positive number if the current object is greater than another
- The type parameter **T** allows arbitrary data type but with compile-time type checking to ensure that two objects to be compared are of the same type

Example of a Comparable class

```
public class MyRectangle implements
    Comparable<MyRectangle> {
    public int compareTo(MyRectangle other) {
        if (this.area() > other.area())
            return 1;
        if (this.area() < other.area())
            return -1;
        return 0;
    }
}
```

Could have been simplified because area() returns an int:

```
public int compareTo(MyRectangle other) {
    return this.area() - other.area();
}
```

We can sort only *Comparable* things

- Consider what we need to know to sort data:
 is $A[i]$ less than, equal to or greater than $A[j]$?
- That's it!
- Thus, **we can sort Comparable data without knowing anything else about it**

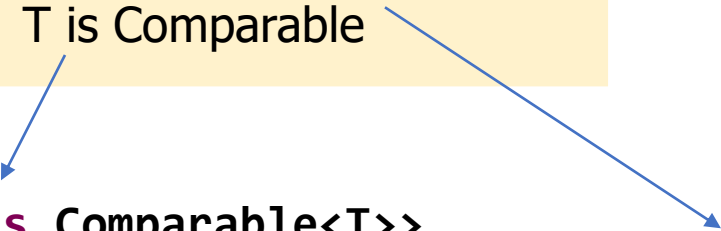
Abstracting Objects into Comparable black boxes

- Think of the objects we want to sort as “black boxes”
 - We know we can compare them because they implement *Comparable*
 - Each type can implement *compareTo()* to be tailored to that type
 - We **don't know (or need to know) anything else about them** – even though they may have many other methods / instance variables – that is all irrelevant to sorting
- Thus, a single *sort* method will work for an array of any class that implements *Comparable*
 - Did I mention that this was awesome!?

Generic sorting: 1/2

/ Sorts the first n objects in an array into ascending order.**/**

- Input is array of T where T is Comparable



```
public static <T extends Comparable<T>>  
    void selectionSort (T[] a, int n)  
{  
    for (int i = 0; i < n - 1; i++)    {  
        int iNextSmallest  
            = getIndexOfSmallest(a, i, n - 1);  
        swap(a, i, iNextSmallest);  
    }  
}
```

Generic sorting: 2/2

See how we are declaring parametrized type here in the method header

```
private static <T extends Comparable<T>>
    int getIndexOfSmallest(T[] a, int first, int last){

    T min = a[first];
        int indexOfMin = first;
        for (int index = first + 1; index <= last; index++){
            if (a[index].compareTo(min) < 0) {
                min = a[index];
                indexOfMin = index;
            }
        }

        return indexOfMin;
    }
```

- Code depends only on the fact that the data in the array is Comparable
- The only generic method called is *compareTo()*
- Via polymorphism, *compareTo()* will be specific to each data type

Generics ensure that we sort the objects of the same type

- An array of any *Comparable* objects can be sorted
- However using the same type parameter T – we restrict the objects to be of exactly the same type
 - Cannot compare objects from inherently different types: don't want to compare "apples to oranges"
 - Note that the argument to *compareTo* is type T
 - T can be arbitrary but it must be compatible for two objects that are being compared
 - If the types are not compatible, a **compile-time error will be generated**

What is wrong with the following code?


```
public final class Min1 {  
    public static <T> T smallerOf (T x, T y) {  
        if (x < y)  
            return x;  
        else  
            return y;  
    }  
}
```

Is this better?

```
public final class Min2 {  
    public static <T extends Comparable<T>> T  
        smallerOf (T x, T y) {  
        if (x < y)  
            return x;  
        else  
            return y;  
        }  
}
```

A correct version

```
public final class Min2 {  
    public static <T extends Comparable<T>> T  
        smallerOf (T x, T y) {  
        if (x.compareTo(y) < 0)  
            return x;  
        else  
            return y;  
        }  
    }
```



Wild cards

- What if we want to use polymorphic method calls and the elements in our collection are different subclasses of some common superclass?
 - We should be able to define a generic type that can be substituted by any subclass
 - However with a generic type T we can only use objects of exactly the same type
- This incompatibility may be softened by the wildcard: we use ? as an actual type parameter
 - ? stands for an *any* type

Upper-bounded wildcards

- These wildcards can be used when you want to relax the restrictions on a type: any subclass of this type

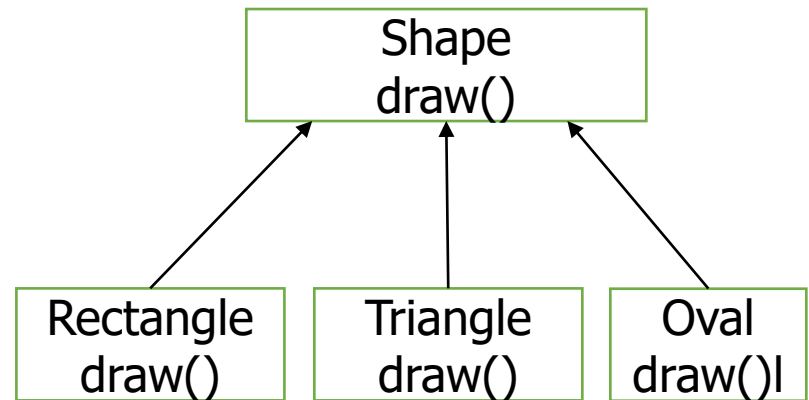
Example 1:

you want to write a method that adds all elements of the List, as long as they are numeric

```
public static void sum (  
    List<? extends Number> list)  
{ ... }
```

Read ? as: anything
that extends
Number, any
subclass of *Number*

Example 2: list of Shapes



```
public void drawAll (  
    List<? extends Shape> shapes)  
{ ... }
```

Lower-bounded wildcards

- It is expressed using the wildcard character ('?'), followed by the *super* keyword, followed by its lower bound: <? super T>
- A lower bounded wildcard restricts the unknown type to be a **specific type or a super of that type**
- Example: you want to write a method that puts Integer objects into a list. To maximize flexibility, you would like the method to work on List<Integer>, List<Number>, and List<Object> — anything that can potentially hold Integer values, or can be cast to Integer

```
public static void sum (List<? super Integer> list)
{ ... }
```

Wildcards in our sorting implementation

```
public static <T extends Comparable<? super T>>  
    void selectionSort (T[] a, int n)
```

- That means we can have an the input array of type T and the Comparable must be implemented in any super of T
- This allows to compare different subclasses if the *compareTo* is defined for the superclass

See [Example3.java](#)

- Here we have an array of People and the *compareTo* is defined in a People class: compares by age
- The array is filled with Students, Workers and other subclasses of People, but they are all valid types because their superclass has an implementation of Comparable interface

Demo

See [SortArray.java](#) and [Example3.java](#)

