Programming Concepts Using Java

Quiz 2 Revision

W5:L1: Polymorphism Revisited

- In object-oriented programming, polymorphism usually refers to the effect of **dynamic dispatch**
- Depending upon the object type stored in a reference variable appropriate version of overridden and non-overridden methods are invoked automatically.
- We are actually **grouping types with one common behaviour** under a parent type (class/interface) which can then polymorphically refer to appropriate subtypes depending upon actual instance type.
- More generally, polymorphism refers to behaviour that depends only a specific capabilities — structural polymorphism
  - Reverse an array/list (should work for any type)
  - Search for an element in an array/list (need equality check)
  - Sort an array/list (need to compare values)

W5:L1: Polymorphism Revisited (Cont.)

Structural Polymorphism: Example

- Use the Java class hierarchy to simulate this

```
public int find (Object[] objarr, Object o) {
    int i;
    for (i = 0; i < objarr.length; i++){
        if (objarr[i] == o) {return i};
    }
    return (-1);
}
```

- Polymorphic `find`
  - `==` translates to `Object.equals()`
- Polymorphic `sort`
  - Use interfaces to capture capabilities

W5:L1: Polymorphism Revisited (Cont.)

- **Using *`Object`* to generalize types in a program**
- **Type Consistency**:
  Source type can be either same as target type or a subtype of target type. In other words a super type reference variable/array can store subtype objects but not vice versa – `tgt[i] = src[i];`
- Arrays, lists, . . . should allow arbitrary elements
- A polymorphic list stores values of `type Object`
- **Problems:**
  - Type information is lost needs explicit casting on every use.
  - Homogeneity cannot be guaranteed.
- **Solution: Generics**
  - Classes and functions can have type parameters
    - `class MyDataStructure<T>` – holds values of unbounded type `T`
    - `public T getMatch(T obj)` – accepts and returns values of same type `T` as enclosing enclosing class
  - Can also use constraints by mixing inheritance rules.

```
public static <S extends T,T> void getMatch(S[] sarr, T obj){...}
```

W5:L2: Generics

- Generics introduce structural polymorphism into Java through type variables
- Example of a polymorphic List

```
public class LinkedList<T>{
    private int size;
    private Node first;
    public T head(){
        T returnval;
        ...
        return(returnval);
    }
    public void insert(T newdata){...}
    private class Node {
        private T data;
        private Node next;
        ...
    }
}
```

W5:L2: Generics (Cont.)

- Be careful not to accidentally hide a type variable

```
class Myclass<S>{
    public <S,T> void myMethod(S obj){
        T obj2;
        ...
    }
}
```

- Quantifier `<S, T>` of `myMethod` masks the type parameter `S` of `class MyClass`.

W5:L3: Generic Subtyping

- **Covariance of types:**
  If `S` is a subtype of `T` and a reference of `T` can store an object of `S`, then `T[]` can also refer `S[]`.
  Arrays are covariant:

```
Integer[] arr1 = {10,20,30};
Number[] arr2 = arr1;
System.out.println(arr2[2]); // 30
```

- Now, try running this:

```
arr2[1] = 9.8; // It's not allowed, generates exception.
```

  The detailed type checking is only done **only at runtime**, *compiler will check for supertype-subtype relation and if that conforms, it will allow the code.*

- **Issue with generics:**
  JVM erases the type information related to a generic type after the compilation is done, i.e. at runtime unlike non-generic type variables/references, generic variables will not have any type characteristics. This process is called **type erasure.**
  Which means all type checking must be done by compiler, but as compiler cannot check object's type during compile time, so JAVA prohibits the covariance property for generic types.

- `List<Subtype>` is not compatible with `List<Supertype>` – not covariance

```
List<String> s = {"A","B"};
List<Object> o = s; //Illegal use
```

W5:L3: Wildcard

- A method `public static void printlist(LinkedList<Object> l){ ... }`
  - cannot be called by
    ```
    LinkedList<String> l;
    printlist(l);
    ```
- One way to solve the problem make the method generic by introducing a type variable: `public static <T> void printlist(LinkedList<T> l) { ... }`
- If `T` is not actually used inside the function We can solve this problem using wildcards `<?>`.
  - Avoid unnecessary type quantification when type variable is not needed elsewhere.
- Beneficial while comparing two different subtypes of a common supertype.
- **Bounded Wildcard**
  - `LinkedList<?>`
- **Bounded Wildcards**
  - `LinkedList<? extends T>`
  - `LinkedList<? super T>`

W5:L5: Type Erasure

- Java does not keep type information of generics at runtime, all type compatibilities are checked during compile time.

  `if(s instanceof ArrayList<String>) //compilation error`

- At run time, all type variables are promoted to `Object`

  `ArrayList<T>` becomes `ArrayList<Object>`

- Or the upper bound,if available

  `ArrayList<T extends Mammals>` becomes `ArrayList<Mammals>`

- Type erasure leads to illegal overloading which were legal on non generics.

```
public void myMethod(ArrayList<Integer> i) {...}
public void myMethod(ArrayList<Mammal> m) {...}
```

- Type erasure means the comparison in following code fragment returns `True`

```
o1 = new LinkedList<Employee>();     o2 = new LinkedList<Date>();
if (o1.getClass() == o2.getClass){
    // True, so this block is executed
}
```

- To avoid runtime errors generic type arrays can be declared but can't be instantiated.

```
T[] arr;
arr = new T[20]; // compiler error
```