Exam 1 Review

- What I regret the most is the loop part
- omega analysis of algorithms
- Bogo sort?

Static vs. instance variables

- What is a static variable

Static Variables

- Variables can either be “attached” to the class or to instances of the class (objects).
- Static variables **are not** associated with any one object’s state. They are usually properties or definitions.
- Non-static variables are called instance variables because they are tied to exactly one instance of an object. They can be accessed with the keyword ‘this’.

Static or No Static?

- When deciding if variable should be static:

Ask yourself: Is it possible that the value of this variable will vary across different objects?

- Consider:

`Rectangle` class :

```
    numSides;
```
static (all rectangles have 4 sides)

```
    height;
```
not static (rectangles can have different dimensions)

Static Methods

- Methods also can either be “attached” to the class or to instances of the class.
- Static methods **do not** depend on the state of the object.
- They can be answered without anything that could reference the keyword “this”. Called using the class name.
- Non-static methods rely on an object’s state, often depending on the values of instance variables. Called on an instance.

Static or No Static?

- To decide if your method should be static:

Ask yourself: Does this method depend on the state of the object, or is it always the same regardless?

- Consider a `Rectangle` class:

```
getArea();
```

not static (depends on a particular rectangle's dims)

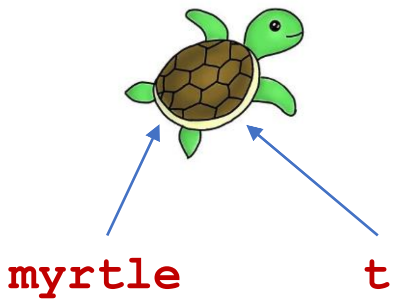
```
calculateArea(int h, int w);
```

static (formula; all info provided as inputs)

Passing references as parameters: 1/2

- The reference (the address of an object) is copied into a new reference variable

```
static void addYear(Turtle t){
    t.age ++;
  }

public static void main (String[] args){
    Turtle myrtle = new Turtle();
    myrtle.age = 5;
    addYear(myrtle);
    System.out.println(myrtle.age);
    System.out.println ("Happy birthday!");
}
```

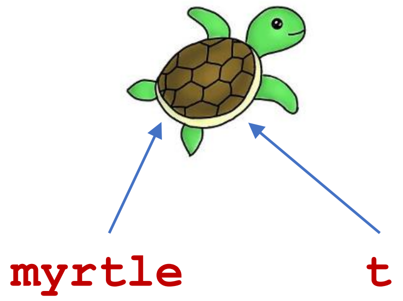


Will myrtle's age change?

Passing references as parameters: 2/2

- The reference (the address of an object) is copied into a new reference variable

```
static void addYear(Turtle t){
    t = new Turtle();
    t.age ++;
  }

public static void main (String[] args){
    Turtle myrtle = new Turtle();
    myrtle.age = 5;
    addYear(myrtle);
    System.out.println(myrtle.age);
    System.out.println ("Happy birthday!");
}
```

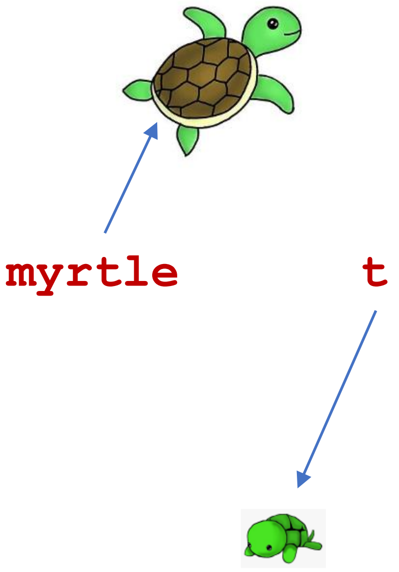


Will myrtle's age change?

Passing references as parameters: 2/2

- The reference (the address of an object) is copied into a new reference variable

```
static void addYear(Turtle t){
    t = new Turtle();
    t.age ++;
  }

public static void main (String[] args){
    Turtle myrtle = new Turtle();
    myrtle.age = 5;
    addYear(myrtle);
    System.out.println(myrtle.age);
    System.out.println ("Happy birthday!");
}
```



Will myrtle's age change?

Example of passing references (immutable Strings) as parameters

```
static void changeName(String name){
    name  = "Mr.  " + name;
}

public static void main (String[] args){
    String myname = "Smith";
    changeName (myname);
    System.out.println (myname);
}
```

Example of passing references (mutable StringBuilder) as parameters

```
static void changeJunior(StringBuilder name){
    name.append(" Jr.");
}

public static void main (String[] args){
    String myname = "Smith";
    changeJunior (myname);
   System.out.println (myname);
}
```

Initialization of generics 1/2

```
public class public LinkedList<E> {
    public LinkedList() {
    }

     boolean add(E e) {
    }
}
```

- Program:

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

When this code compiles: the definition of the class changes: all Es are substituted with Integer

Initialization of generics: 2/2

```
public class public LinkedList {
    public LinkedList() {
    }

     boolean add(Integer e) {
    }
}
```

At this moment the generic list becomes a LinkedList of integers

- Program:

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

It is important because the logic of operations remains the same for any type of data: so we do not need to implement a separate LinkedList for integers, for Strings and for the list of Cars.

Example: method with generics Not everything can be compared

Access

Static or instance method

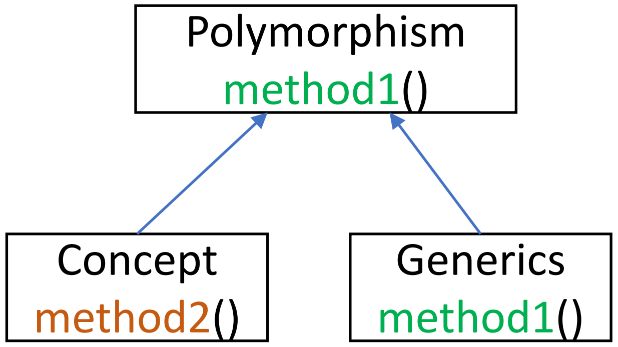
Declaring that we are going to use a generic type here

Return type

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

public static void main(String [] args) {
    String minStr = min(args[0], args[1]);
    System.out.println(minStr);
}
```

Dynamic/late binding: 1/5

```java
private class Polymorphism {
  void method1 () {
    System.out.println("Polymorphism");
  }
}

private class Concept extends Polymorphism {
  void method2 () {
    System.out.println("Concept");
  }
}

class Generics extends Polymorphism {
  void method1() {
    System.out.println("Generics");
  }
}
```

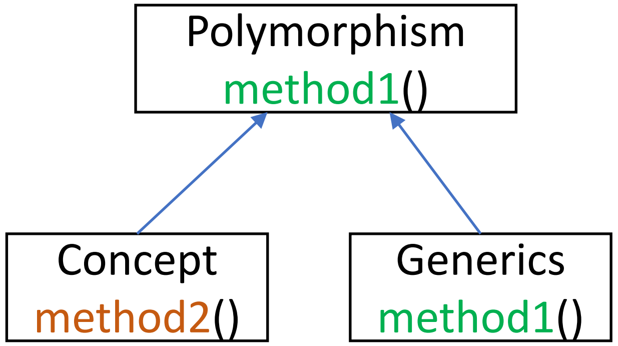


- Overriding method1 of a superclass

Dynamic/late binding 2/5

```
private class Polymorphism {
  void method1 () {
    System.out.println("Polymorphism");
  }
}

private class Concept extends Polymorphism {
  void method2 () {
    System.out.println("Concept");
  }
}

class Generics extends Polymorphism {
  void method1() {
    System.out.println("Generics");
  }
}
```

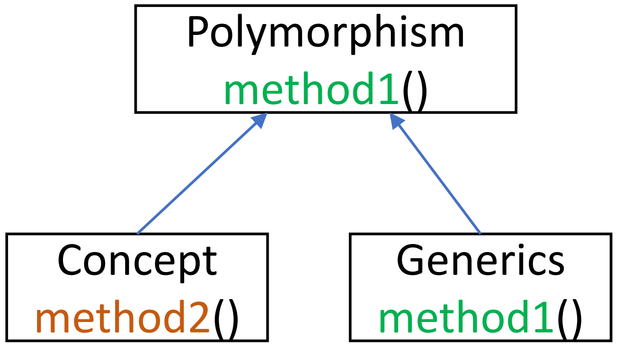
- During **compilation** the compiler only checks that the method exists in the object of type Polymorphism – because all the objects in the array are stored in a variable of type Polymorphism (superclass)



```
private class Test  {
public static void main ( …) {
  Polymorphism [ ] p =
          new Polymorphism [3];
  p[0] =  new Polymorphism();
  p[1] = new Concept();
  p[2] = new Generics();

  for (int i =0; i< 3; i++)
    p[i].method2();
  }
}
```

Dynamic/late binding 3/5

```
private class Polymorphism {
  void method1 () {
    System.out.println("Polymorphism");
  }
}

private class Concept extends Polymorphism {
  void method2 () {
    System.out.println("Concept");
  }
}

class Generics extends Polymorphism {
  void method1() {
    System.out.println("Generics");
  }
}
```



```
private class Test  {
public static void main ( …) {
  Polymorphism [ ] p =
        new Polymorphism [3];
  p[0] =  new Polymorphism();
  p[1] = new Concept();
  p[2] = new Generics();

  for (int i =0; i< 3; i++)
    p[i].method2();
  }
}
```

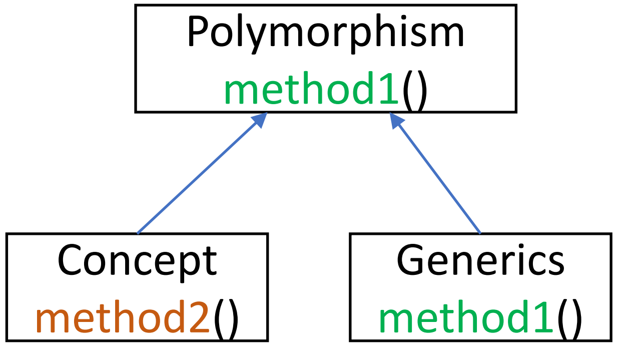
THIS WILL NOT COMPILE

- During **compilation** the compiler only checks that the method exists in the object of type Polymorphism – because all the objects in the array are stored in a variable of type Polymorphism (superclass)

Dynamic/late binding 4/5

```
private class Polymorphism {
   void method1 () {
      System.out.println("Polymorphism");
   }
}

private class Concept extends Polymorphism {
   void method2 () {
      System.out.println("Concept");
   }
}

class Generics extends Polymorphism {
   void method1() {
      System.out.println("Generics");
   }
}
```

- During **compilation** the compiler only checks that the method exists in the object of type Polymorphism – because all the objects in the array are stored in a variable of type Polymorphism (superclass)



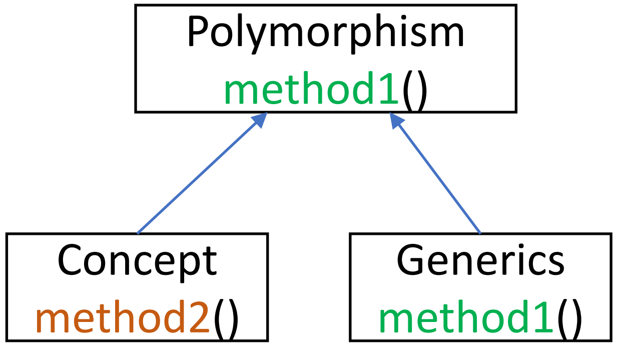
```
private class Test  {
public static void main ( …) {
  Polymorphism [ ] p =
        new Polymorphism [3];
  p[0] =  new Polymorphism();
  p[1] = new Concept();
  p[2] = new Generics();

  for (int i =0; i< 3; i++)
    p[i].method1();
  }
}
```

This compiles but the code for method 1 is not bound to this place in the program

Dynamic/late binding 5/5

```
private class Polymorphism {
  void method1 () {
    System.out.println("Polymorphism");
  }
}

private class Concept extends Polymorphism {
  void method2 () {
    System.out.println("Concept");
  }
}

class Generics extends Polymorphism {
  void method1() {
    System.out.println("Generics");
  }
}
```

- When the program is running, it will look at what actual object is stored at the other end of a variable: and will **bind (load into memory) the corresponding method during run time**. This is called ***Dynamic binding***.



```
private class Test  {
public static void main ( …) {
  Polymorphism [ ] p =
        new Polymorphism [3];
  p[0] =  new Polymorphism();
  p[1] = new Concept();
  p[2] = new Generics();

  for (int i =0; i< 3; i++)
    p[i].method1();
  }
}
```

Cenerics has its own method1, but it will only be loaded during run-time

Interfaces and multiple inheritance

- One class can only extend one superclass in Java
- Multiple inheritance (inheriting methods from superclasses belonging to different hierarchies) is not supported in Java
- However one class can implement multiple Interfaces
- In this case it still cannot make use of the code from different classes (because interfaces do not contain any actual code)
- However now the object can be part of different polymorphic collections: it can play different roles

Example: multiple inheritance

```
class Report extends Paper, implements
    Printable, Comparable, Disposable {}


class Memo  implements Printable,
    Disposable {}
```

```
interface Printable {
    void print();
}

interface Disposable {
    void dispose();
}
```

```
Printable [] todayPile = new Printable[3];
todayPile[0] = new Report();
todayPile[1] = new Report();
todayPile[2] = new Memo();

for (int i =0; i< 3; i++)
 todayPile[i].print();
}
```

```
Disposable [] discardPile = new Disposable[3];
discardPile[0] = new Report();
discardPile[1] = new Report();
discardPile[2] = new Memo();

for (int i =0; i< 3; i++)
 discardPile[i].dispose();
}
```

Same Report object plays different roles: once as a Printable object and other time as a Disposable object

All objects that implement the same Interface must have the implementation of all the method(s) declared in this Interface, and now can be uniformly treated in the same collection

Abstract data types vs abstract data structures?

- Abstract Data Type represents an idea of a type that based on the needs of the particular application. It includes the description of the data to be stored and the desired operations we want it to support.

- Data structures are not abstract: they are very concrete. They represent the actual layout of data in computer memory. We use Array, Linked List, later the Linked Tree data structure to implement the functionality declared in the specification of a given ADT.

- Each ADT can be implemented using different concrete data structures

Linked List: Node

- The amount of times Node is used to describe different objects. Head is a Node but it has no data so it's not really a Node but it's still a Node... Mindbending
- Could you please explain how linked list work and how do we add in front of the list and in the middle?

Explained on the board

Also see recitation 6

Big O

- Knowing when to use the Big O notation in an algorithm
  - You calculate big O to characterize the quality of an algorithm or to compare several algorithms

Big O practice

- I feel like I need more practice looking at code and determining the time complexity of it.
- How to get the time complexity given code.
- To understand how to individually evaluate a loop and determine its time complexity.
- Quickly understanding the math of the loops
- I think I just need more practice with remembering what applies to what

Here are big O exercises for all different kinds of loops: LINK

Big confusion: O (log n)

- I don't understand why binary search is log n
- how to identify if an algorithm is log n.
- When would O(logn) be used for Big Oh and how is it derived?
- when the answer is O(logn)
- The log values in Big O
- loops that are log(n) or nlog(n)
- How do you calculate logarithmic runtime from *while* loops?
- I'm still a bit lost on the inner loops and whenever log is involved.

Miscellaneous

- RAM model of computation: expressing algorithm runtime in terms of n, not O(n)
- i am confused about the different sorting techniques, for example bubble sort. I just don't understand the implementation of them in this class.
- SQueue
- Balanced Parentheses