CMSC 132

Intro to Object Oriented Programming II

Ekesh Kumar
Prof. Nelson Padua-Perez • Fall 2019 • University of Maryland
https://www.cs.umd.edu/class/fall2019/cmsc132/

Last Revision: January 14, 2020

Contents

1 Monday, August 27, 2019

This class is CMSC132: Object Oriented Programming II. We will be covering modern software development techniques, various algorithms and data structures, and more. This course is taught in Java 11, but the techniques taught in this course carry over to several other programming languages.

Logistics

- All lectures are recorded and posted on Panopto.
- No collaboration on projects.
- No pop quizzes.
- We will be using the Eclipse IDE.
- Projects, office hours, lecture videos, and most other resources can be accessed from the class webpage: https://www.cs.umd.edu/class/fall2019/cmsc132/.
- Projects are due at 11:30 p.m. on the specified day. They can, however, be submitted up to 24 hours late with a 12% penalty.
- The grade breakdown is 26% for Projects, 16% for Quizzes, Exercises, and Lab Work, 8% for Midterm 1, 11% for Midterm 2, 11% for Midterm 3, and 28% for the Final Exam.

Abstraction and Encapsulation

There are important two techniques used in Object-Oriented Programming that we need to become familiar with: abstraction and encapsulation.

Abstraction is a technique in which we provide a very high-level model of activity or data. Small details about the model's functionality are not specified to the user. A good example . Abstraction can further be divided into two sub-categories, which are described below:

1. **Procedural abstraction** is a type of abstraction in which the user is aware of what actions are being performed, but they are not told how the action is performed. For example, suppose we would like to sort a list of numbers. There are many algorithms that can do this for us. Under procedural abstraction, we would know that our end result is a sorted list of numbers, but we wouldn't know which algorithm is being used.
2. **Data abstraction** is a type of abstraction in which various data objects are known to the user, but how they are represented or implemented is not known to the user. An example of data abstraction is shown by representing a list of people. While the user would know that they have a list of people, they wouldn't know how the list is being represented (for example, we could use an array, an ArrayList, or any other data structure).

An **abstract data type** (ADT) is an entity that has values and operations. More formally, an abstract data type is an implementation of interfaces (a set of methods). Note that it is "abstract" because it does not provide any details surrounding how these various operations are implemented.

An example of an abstract data type is a queue, which supports the operation of inserting items to the end of the queue as well as the operation of retrieving items from the front of the queue. Note, again, that we are not concerned with how these operations should be performed internally.

Finally, **encapsulation** is a design technique that calls for hiding implementation details while providing an interface (a set of methods) for data access. A familiar example of encapsulation is shown through the ArrayList in Java. The ArrayList provides various methods that are accessible to us, such as the `.add()` and `.at()` methods. We aren't concerned with how they are implemented internally.

The Java Programming Language

Different programming languages provide varying levels of support for object-oriented programming. In particular, Java and C++ allow us to easily perform object-oriented programming. How does Java does this?

- Java provides us with **interfaces**, which allows us to specify a set of methods that another class must implement. This allows us to easily express an abstract data type since we can specify the operations and data that an entity must have. Interfaces follow an "is-a" relationship, meaning that a class that implements an interface is what the interface specifies. As an example, consider an animal interface. If an elephant implements the animal interface, we can perform tasks that are meant for animals with elephants (such as passing an elephant into a function that accepts animals).
- Java provides us with **classes**, which can be used as blueprints for other classes. Classes can extend other classes, which makes them **subclasses**. These subclasses inherit functions from the original class, and this also defines an "is-a" relationship.

Here are some key points that we should remember about interfaces:

- An interface cannot be instantiated. So, if we have an interface called `animal`, typing `Animal a = new Animal()` **would not compile**.
- An interface can contain many public members, including static final constants, abstract methods (which have no body), default methods (with a body), static methods, and static nested types.

Let's look at an example of an interface:

```
import java.util.*;

public interface Animal {

    public void feed(String food);
    public int getAge();
    public boolean manBestFriend();

    default void grow() {
        System.out.println("I grow");
    }
}
```

Here, we've created an interface named `Animal`, which other classes can implement. Any class that implements this class will need to also implement the functions `feed`, `getAge()` and `manBestFriend()` with the same return values and parameters specified. The `grow()` function will be associated with any class that implements this interface.

Now let's look at another class that implements the `Animal` interface:

```
package animalExample;

public class Dog implements Animal {
  private String name;
```

```
   private int age;

   public Dog(String name, int age) {
      this.name = name;
      this.age = age;

   }

   public void feed(String food) {
      System.out.println("Feeding " + name + " with " + food);
   }

   public int getAge() {
      return age;
   }

   public boolean manBestFriend() {
      return true;
   }

   public void bark() {
      System.out.println(name + " is barking");
   }
}
```

Here are the key things that we should note:

- The `Dog` class implements the `Animal` class. Thus, it is **required** to implement all of the methods specified by `Animal` (if we were to comment out even one function, the code wouldn't compile).
- The `Dog` class is allowed to add its own methods, like `bark()`. Likewise, the `Dog` class is allowed to create its own variables, like `name` and `age`.

Here's another class that implements the `Animal` interface:

```
package animalExample;

public class Piranha implements Animal {
   private int age;

   public Piranha(int age) {
      this.age = age;
   }

   public void feed(String food) {
      System.out.println("Piranha is eating " + food);
   }

   public int getAge() {
      return age;
   }

   public boolean manBestFriend() {
      return false;
   }

   public void grow() {
```

```
    System.out.println("I grow like a fish");
  }
}
```

Here are some more things that we should take careful note of:

- More than one class can implement an interface. In this example, both `Dog` and `Piranha` implement `Animal`.
- We can rewrite methods that are provided to us. More specifically, `Piranha` implements `Animal`, but it rewrites the `grow()` function that was originally provided to it.

Recall that interfaces define an "is-a" relationship. Since `Piranha` and `Dog` implement `Animal`, we can now treat them as an `Animal`. A good way to verify the "is-a" relationship is the `instanceof` keyword. In Java, `instanceof` allows us to check whether one object is an instance of a specified type (class, subclass, or interface).

To demonstrate, suppose we declare a piranha variable `p`. We could write a conditional such as `if (p instanceof Animal) { /* Code to be executed */}`, in order to make sure that `p` is an instance of `Animal`. Why would we ever need to do this? This is particularly useful when we're casting. If we attempt to cast an object to a subclass of which it is not an instance of, Java throws a **ClassCastException**. We can prevent this exception by putting the relevant code in a conditional checking the instance of what we're casting.

2 Wednesday, August 30, 2019

Collections

The **Java Collections Framework** is a set of methods and classes that provide support for **collections**, which are objects that group multiple elements into one unit. There is a collection interface that defines a set of methods that certain classes must have. It turns out that ArrayList is in the collections framework, and it implements the set of required methods (such as `.add()`, `.clear()`, `.contains()`, and more).

Consider the following code segment:

```
package miscellaneous;

import java.util.Collection;
import java.util.ArrayList;
import java.util.LinkedList;

public class CollectionExample {

   public static Collection<Integer> defineElements(boolean arrayFlag, int numberOfValues, int
       maxValue) {
      Collection<Integer> elements;

      if (arrayFlag) {
         elements = new ArrayList<Integer>();
      } else {
         elements = new LinkedList<Integer>();
      }

      for (int j = 0; j < numberOfValues; j++) {
         elements.add((int) (Math.random() * maxValue + 1));
      }

      return elements;
   }

   public static void displayValues(Collection<Integer> data) {
      if (data.isEmpty()) {
         System.out.println("Empty Collection");
      } else {
         for (Integer value : data)
            System.out.print(value + " ");
      }
   }

   public static void main(String[] args) {
      Collection<Integer> firstCollection = defineElements(true, 5, 10);

      System.out.println("First Collection:");
      displayValues(firstCollection);

      System.out.println("\nSecond Collection:");
      Collection<Integer> secondCollection = defineElements(false, 5, 10);
      displayValues(secondCollection);

      /* Example of methods defined by the Collection interface */
      Collection<Integer> union = new ArrayList<Integer>();
```

```
        /* Combining elements */
        union.addAll(firstCollection);
        union.addAll(secondCollection);
        System.out.println("\nUnion");
        displayValues(union);

        /* Checking if elements from one collection are present in antoher */
        if (union.containsAll(firstCollection))
           System.out.println("\nIncludes all elements of first collection");

        /* Removing elements in union present in second collection */
        union.removeAll(secondCollection);
        System.out.println("After removing");
        displayValues(union);

        /* Clearing the collection */
        union.clear();
        System.out.println();
        displayValues(union);
    }
}
```

This code segment is another demonstration of how we can utilize the fact that some objects are instances of other types. The `defineElements` takes in various parameters and returns either a LinkedList (another type list provided in the Collections Framework) or an ArrayList depending on the parameters provided. The reason why this is allowed is because of the "is-a" relationship between ArrayLists, LinkedLists, and Collections. More specifically, both ArrayLists and LinkedLists are collections. Consequently, since the return type of this function is a Collection, we are allowed to return any object that is an instance of Collection.

On the other hand, the `displayValues` method takes in a Collection type, and we're allowed to pass in any object that is an instance of Collection, as seen in the driver.

Collections are also really useful because they have built-in functions like `.shuffle()` (which scrambles the collection) and `.sort()`, which sorts the function.

Raw Types

We typically write, `ArrayList<Integer> = new ArrayList<Integer>()`, when we're instantiating an ArrayList of integers. Likewise, we could write, `ArrayList<Bananas> = new ArrayList<Bananas>()` if we had created a class called `Bananas`, or `ArrayList<String> = new ArrayList<String>()` if we wanted an ArrayList of strings. This is example of **generics**: we are allowed to pass in whatever type we want our ArrayList to store.

Why are generics useful? Firstly, generics help us reduce the amount of code we need to write. Instead of writing lots of code that gets executed depending on whether the user is using an integer or a string, we can use generics in order to reduce the amount of code needed. The `.add()`, `.remove()` and other methods used by ArrayList are valid no matter what type we are storing in our ArrayList. If we didn't have generics, we'd need to have.

Secondly, generics help us move some casting errors from runtime to compile time. Consider the following code segment:

```
class A { ... }
class B { ... }
List myL = new ArrayList(); // raw type
myL.add(new A()); // Add an object of type A
```

```
...
B b = (B) myL.get(0);
```

Here, we've declared two classes, and we've created a List called `myL`. In this example, we have not used any generics (this is called a **raw type**) since we haven't specified the type that this list will store. Thus, Java will permit us to add whatever we want to this list. This can lead to issues if we don't remember what type is stored at each index (in the above example, for instance, retrieved the first element by casting it as `B` when it is actually of type `A`.). This issue won't be seen until runtime. Using generics, however, would move this error to compile time.

Here is the same code segment but with generics:

```
class A { ... }
class B { ... }
List<A> myL = new ArrayList<A>();
myL.add(new A());
A a = myL.get(0);
...
B b = (B) myL.get(0); // Compile-time error
```

Now that we've used generics, we'll find the casting error during compile time.

So far, we've only seen how collections use generics. How do we use our own generics? We can parametrize classes, interfaces, and methods using `<X>` notation. For example, we could write, `public class foo<X, Y, Z> { ... }`, to define a class called `foo` that takes in three types.

The Iterable Interface

The **Iterable Interface** defines a method called `Iterator`, which returns an iterator that allows us to traverse a container (particularly lists). ArrayLists in Java implement the iterable interface.

The following example demonstrates how we can use iterators:

```
public static void main(String[] args) {
    ArrayList<String> myList = new ArrayList<String>();
    myList.add("a");
    myList.add("b");

    Iterator<String> it = myList.iterator();

    while (it.hasNext()) {
        System.out.println(it.next());
    }
}
```

This code segment will print out everything in our list `myList`.

However, it turns out that we don't actually need to handle iterators ourselves. We can use the **enhanced for loop** (also known as a **for each loo**), which does this for us. These types of for loops handle the iterator automatically.

Here's how we can use an enhanced for loop with our previous example:

```
public static void main(String[] args) {
    ArrayList<String> myList = new ArrayList<String>();
    myList.add("a");
    myList.add("b");
```

```
    for (String s : myList) {
        System.out.println(s);
    }
}
```

The for loop in this code segment can be read as, "for each string `s` in `myList`, print `s`."

3 Wednesday, September 4, 2019

The Comparable Interface

The **comparable interface** is used in order to provide an ordering to objects of a user-defined class. When a class implements this interface, it must implement the `.compareTo()` method, which returns a negative value if the current object is less than the object being compared to, zero if they're equal, and positive otherwise.

As an example, consider the following code segment:

```
package equalsComparable;
import java.util.ArrayList;

public class Example {

    public static void main(String[] args) {
        ArrayList<String> l = new ArrayList<String>();
        l.add("ice_cream");
        l.add("apple");
        l.add("raw_onion");
    }
}
```

If we were to print this ArrayList with `System.out.println(l)`, we'd get the output `[ice_cream, apple, raw_onion]`, which is the order in which we inserted the elements into the ArrayList.

But now, what if we wanted to sort our elements and then print them? Calling `Collections.sort(l)` and then printing would sort our items lexicographically, so our output would be `[apple, ice_cream, raw_onion]`. Why does this happen? Because the string class has a built-in `compareTo` method, which sorts lexicographically.

It turns out that we can define our own `compareTo()` method and sort our elements according to how we want. As mentioned previously, this can be done by implementing the `Comparable` interface.

Consider the following code segment, which illustrates how we can use the `Comparable` interface:

```
public class Student implements Comparable<Student> {
   private String name;
   private int id;

   public Student(String name, int id) {
      this.name = name;
      this.id = id;
   }

   public String toString() {
      return "Name: " + name + ", Id: " + id;
   }

   public int compareTo(Student other) {
      if (id < other.id) {
         return -1;
      } else if (id == other.id) {
         return 0;
      } else {
         return 1;
      }
   }
```

```
  public boolean equals(Object obj) {
     if (obj == this) {
        return true;
     } else if (!(obj instanceof Student)) {
        return false;
     } else {
        return compareTo((Student) obj) == 0;
     }
  }

  /* If we override equals we must have correct hashCode */
  public int hashCode() {
     return id;
  }
}
```

Note that the `Student` class implements the `Comparable` interface (whatever comes in the angled brackets `< ... >` specifies what we will be comparing). Consequently, the `compareTo` method is overriden so that we can specify that we want to sort the students based on their IDs (we retunr a negative value when the student's ID is less than the other students, zero when they're equal, and a positive value otherwise). Now, if we use `Collections.sort()` on an ArrayList of `Student` objects, we will sort based on ID.

An even simpler implementation of `compareTo` that would do just what we want would be to only have the statement `return id - other.id`.

Finally, we have a `hashCode` method, which will be discussed in-depth at a later time. But for now, it's important to know that, whenever the `equals` method is overriden, we must provide a `hashCode` method with the same header as shown in the code segment above. Java's enforcement of this "policy" is known as a **hash code contract**.

Annotations

In Java, **annotations** are used to help find runtime errors at compile time, and also provide additional information to others as to what you are trying to do. The general syntax for an annotation is `@[annotation name]`. Some of the common ones used by the compiler are listed below:

- `@Test` indicates that a test follows.
- `@Override` indicates that an overridden function follows.
- `@SuppressWarnings` indicates that the compiler should suppress warnings surrounding the code that follows.

What's an example in which annotations can help us?

Suppose we're overriding a function. If we make a mistake and write the function header for the overridden wrong, our compiler won't interpret this as an error (assuming everything else is right). But, if we were to put the `@Override` annotation before we override the function, our compiler will notice that we've got the wrong function header, and it will give us an error.

This is particularly helpful because it helps us identify otherwise latent bugs.

4 Friday, September 6, 2019

An Introduction to Inheritance

As mentioned briefly last class, **inheritance** is a technique that allows us to capitalize on some features that have already been implemented in a class. Ultimately, this allows us to reduce code duplication, and it also increases readability.

Suppose, for the sake of example, that we have a class `Shape`, and we want to implement several other related classes, like `Square`, `Rectangle`, and `Circle`. Since squares, rectangles, and circles are all examples of shapes, this is a clear example in which we can use inheritance. In computer science terminology, we would say that the `Square, Rectangle,` and `Circle` classes are **subclasses** or the `Shape` **superclass**. These subclasses **extend** their superclass.

In essence, by putting the variables and methods that are common to all shapes in the `Shape` class, we can avoid having to re-add them in every class that extends the `Shape` class (the subclasses can inherit what we want them to). We can subsequently define new instance variables and new methods that are specific to the shape we are implementing. For example, the subclasses might inherit a floating-point `area` field from the `Shape` class, but they must implement their `getArea()` methods in different ways.

The following classes, used to represent a University Database, clearly depict how some features of inheritance work. There are three classes that compose this program.

First, the `Person` class:

```
package university;

public class Person {
   private String name;
   private int id;

   public Person(String name, int id) {
      this.name = name;
      this.id = id;
   }

   public Person() {
      this("Unknown", 0);
   }

   public Person(Person person) {
      this(person.name, person.id);
   }

   public String getName() {
      return name;
   }

   public int getId() {
      return id;
   }

   public void setName(String name) {
      this.name = name;
   }

   public void setId(int id) {
```

```
      this.id = id;
   }

   public String toString() {
      return "[" + name + "] " + id;
   }

   public boolean equals(Object obj) {
      if (obj == this)
         return true;
      if (!(obj instanceof Person))
         return false;
      Person person = (Person) obj;

      return name.equals(person.name);
   }

   public int hashCode() {
      return id;
   }

   public static void main(String[] args) {
      Person p1 = new Person("Paul", 10);
      Person p2 = new Person("Mary", 20);
      Person p3 = new Person(p2);

      System.out.println(p1);
      System.out.println("Same?: " + p1.equals(p2));
      System.out.println("Same?: " + p2.equals(p3));
   }
}
```

The `Person` class, shown above, is a very general class used to represent an individual person. There are some fields, like `name` and `id`, which characterize a person attending the university we are representing. Also, there are a few different ways in which we can instantiate a `Person` object, as seen by the different constructors provided. Note that we have also override the `equals` method, which simply compares the names of the people. As we've learned, we can prevent some compile-time errors by adding the `@Override` annotation before this method. Finally, we have provided a `hashCode` method in order to satisfy Java's hash code contract (as mentioned before, this will be discussed in-depth at a later time).

Next, we have the `Student` class:

```
package university;

public class Student extends Person {
   private int admitYear;
   private double gpa;

   public Student(String name, int id, int admitYear, double gpa) {
      super(name, id); /* calls super class constructor */
      this.admitYear = admitYear;
      this.gpa = gpa;
   }

   /* What would happen if we remove the Person default constructor? */
   public Student() {
      super(); /* calls base case constructor (what if we remove it?) */
```

```
    admitYear = -1;
    gpa = 0.0;
  }

  public Student(Student s) {
    super(s); /* calls super class copy constructor */
    admitYear = s.admitYear;
    gpa = s.gpa;
  }

  public int getAdmitYear() {
    return admitYear;
  }

  public double getGpa() {
    return gpa;
  }

  public void setAdmitYear(int admitYear) {
    this.admitYear = admitYear;
  }

  public void setGpa(double gpa) {
    this.gpa = gpa;
  }

  public String toString() {
    /* Using super to call super class method */
    return super.toString() + " " + admitYear + " " + gpa;
  }

  public boolean equals(Object obj) {
    if (obj == this)
      return true;
    if (!(obj instanceof Student))
      return false;
    Student student = (Student) obj;

    /* Relying on Person equals; passing student */
    return super.equals(student) && admitYear == student.admitYear;
  }

  public static void main(String[] args) {
    Student bob = new Student("Bob", 457, 2004, 4.0);
    Student robert = new Student(bob);
    Student tom = new Student("Tom", 457, 2004, 4.0);

    System.out.println(bob);
    System.out.println("Same:" + bob.equals(robert));
    System.out.println("Same:" + tom.equals(robert));
  }
}
```

Firstly, we can observe that the `Student` class extends the `Person` class, meaning that it extends the public methods and fields that `Student` has.

The **super** class, when used in a method, calls the method with the same method header in the superclass

of the class `super` is being called in.

Now that we've already implemented various constructors in the `Person` class, it isn't necessary to re-implement them in the `Student` class since we're using inheritance. In each of our `Student` constructors, we just make a `super()` call to access the corresponding constructor in the parent `Person` class, which does what we want it to do.

Furthermore, we see that, in the `Student` constructor, we call `super(name, id)`, which calls the superclass's constructor. Whenever we're making `super()` call in a constructor, it is necessary for the `super()` call to be the first statement in a constructor.

In the default `Student()` constructor, which takes no parameters, it's fine to remove the `super()` call. Java will automatically add it for you, and it will call the default constructor of the superclass.

It's also important to note how Java can recognize which constructor to use from the parent class. For example, just `super()` will call the default constructor in the parent class, whereas `super(s)` will call the copy constructor in the parent class. It is particularly interesting to note how the superclass takes a `Person` object in its copy constructor, whereas the `Student` subclass takes a `Student` object. This is perfectly fine since every student "is-a" person.

If we didn't want the `Student` class to access some method of the `Person` class, we could just make that method private.

Finally, we have the `Faculty` class:

```
package university;

public class Faculty extends Person {
   private int hireYear; /* year when hired */

   public Faculty(String name, int id, int hireYear) {
      super(name, id);
      this.hireYear = hireYear;
   }

   public Faculty() {
      super();
      hireYear = -1;
   }

   public Faculty(Faculty faculty) {
      /* Why are we using get methods for the first two ? */
      this(faculty.getName(), faculty.getId(), faculty.hireYear);
   }

   int getHireYear() {
      return hireYear;
   }

   void setHireYear(int year) {
      hireYear = year;
   }

   public String toString() {
      return super.toString() + " " + hireYear;
   }

   public boolean equals(Object obj) {
      if (obj == this)
         return true;
```

```
    if (!(obj instanceof Faculty))
      return false;

    Faculty faculty = (Faculty) obj;

    /* Relying on Person equals; passing student */
    return super.equals(faculty) && hireYear == faculty.hireYear;
  }

  public static void main(String[] args) {
    Faculty john = new Faculty("John", 20, 2004);
    Faculty jack = new Faculty(john);
    Faculty mary = new Faculty("Mary", 101, 2010);

    System.out.println(mary);
    System.out.println("Same:" + john.equals(jack));
    System.out.println("Same:" + jack.equals(mary));

    System.out.println("Id: " + john.getId());
    System.out.println("HireYear: " + john.getHireYear());
  }
}
```

Similar to the `Student` class, the `Faculty` class extends the `Person` class and uses its constructors with `super()`. Also similar to the `Student` class, the `super()` call in the default `Faculty()` constructor is optional (Java will automatically add one if you don't have it).

Here are a few key points that you should remember about inheritance:

- The `extends` keyword is used to specify that a class is a subclass of another class. For example, `public class Student extends Person { .. }` indicates that `Student` is a subclass of `Person`.
- The `this` keyword is used to refer to the current class we're in.
- Subclasses inherit everything from their superclasses that isn't private.
- Superclasses and subclasses illustrate an "is-a" relationship — as seen by the copy constructor example, we can use the subclass objects wherever Java is expecting a superclass object without error.
- The `super()` call can be invoked when initializing a subclass object in order to use the superclass constructor.
- A `super()` call must be the first statement in your constructor.
- If you don't use a `super()` call, Java will automatically invoke the base class's default constructor. What happens if the base class doesn't have a default constructor? We get a compile-time error.

A consequence of the "is-a" relationship between the subclass and superclass is seen by the validity of various assignments. Going back to our university example, since a `Student` is a `Person`, if we declared a `Student` object `s` and a person object `p`, it would be fine to do something like, `p = s`. More specifically, it's fine to assign a person to a student since a student is a person. However, the other way around is not allowed.

Also, if we don't like a method that the superclass provides to us, we can override it in order to get a new one. This is seen in the `toString()` method in the university example.

5 Monday, September 9, 2019

The Protected Keyword

Recall that when a subclass extends a superclass, it cannot access any private fields in the superclass. What happens if we want to permit subclasses of a superclass to access various fields, but we also want to retain some level of privacy? This is where the **protected** keyword can help us. Like `public` and `private`, the `protected` keyword is used to specify what can access a variable.

When do we declare a variable to be `protected`? We should use the `protected` keyword when we need to access it from the enclosing class (the class it is in) subclasses that extend that class, and other classes in the same package (folder) as the enclosing class.

Essentially, declaring variables as `protected` allows us to limit the visibility of a variable, while still permitting subclasses to access that variable.

Early and Late Binding

Consider the following code segment:

```
Faculty carol = new Faculty("Carol Tuffteacher", 458, 1995);
Person p = carol;
System.out.println(p.toString());
```

Given that both the `Person` and `Faculty` classes implement a `toString()` method, which one should be used when executing the print statement on the third line? Should it be the `Faculty` object's `toString()` method? Or, should it be the `Person` object's `toString()` method? There are good arguments for either choice:

- One argument is that the variable `p` was initially declared to be of type person. Thus, we should use the Person's `toString()` method. This is known as **early (static) binding**. The name makes sense since we know early on (as soon as we've declared the object) what method will be used.
- The other argument says that the object to which `p` is referring to was created as a `Faculty` object, so we should use the `Faculty` object's `toString()` method. This is known as **late (dynamic) binding**. Again, this name is fitting since we don't figure out which method will be called when the person is initially declared.

There are pros and cons to both early binding and late binding. On one hand, early binding is more efficient since the decision of what set of methods should be used is determined at compile-time. On the other hand, late binding allows for more flexibility.

By default, **Java uses late-binding**, so in our example, the `Faculty`'s `toString()` method will be called. Essentially, late binding says that the method that is called is dependent on the object's actual type, and not the declared type of the referring variable.

Polymorphism

Java's use of late binding enables us to use a single reference variable to refer to objects of many different types. In the following code segment, we illustrate this fact by creating an array of various university people:

```
Person[] list = new Person[3];
list[0] = new Person("Col. Mustard", 10);
list[1] = new Student("Ms. Scarlet", 20, 1998, 3.2);
```

```
list[2] = new Faculty("Prof. Plum', 30, 1981);
for (int i = 0; i < list.length; i++) {
    System.out.println(list[i].toString());
}
```

The above code segment is perfectly valid, and it will use different `toString()` methods depending on type of the object it at each entry.

This is an example of **polymorphism**, which can more concretely be defined as a method of using a single symbol to represent multiple different types. Note that each of `list[1], list[2],` and `list[3]` are polymorphic variables since we can assign to them a `Person, Student`, or `Faculty` type without error.

getClass and instanceof

In Java, objects can obtain information about their types dynamically. This is done using the `getClass` and `instanceOf` methods that Java provides.

These methods do exactly what they sound like. The `getClass()` method returns the runtime class of an object. Internally, this is represented as a location in memory. It is also perfectly valid to compare the return-value of `getClass()`. For example, if we have two objects `b1` and `b2`, we could use the conditional `if (b1.getClass() == b2.getClass()) {...}` in order to check whether or not `b1` and `b2` are instances of the same class. This conditional will evaluate to "true" if both `x` and `y` are of the same type (i.e. both are Strings).

Now, consider the following code segment in which `getClass()` returns false:

```
Person bob = new Person(...);
Person ted = new Student(...);

if (bob.getClass() == ted.getClass()) { ... } // false (ted is really a Student).
```

The conditional inside of the `if` clause will evaluate to "false." Why? Because the `getClass()` method compares the runtime class of an object. Here, `ted` was initially declared to be of type `Person`, but it refers to a `Student` type. This example further illustrates the fact that Java uses late (dynamic) binding.

Next, we'll discuss the `instanceof` keyword.

The `instanceof` keyword is is used to determine whether one object is an instance of some other class. In terms of inheritance, Class A is an instance of Class B if Class A extends Class B. It's important to remember that `instanceof` is an **operator** in Java, not a method call. Thus, we do not invoke `.instanceof` on the object (there shouldn't be a period!).

The following example illustrates how both `getClass` and `instanceof` operate:

```
package university;

public class InstanceGetClass {
   public static void main(String[] args) {
      Person bobp = new Person("Bob", 1);
      Person teds = new Student("Ted", 2, 1990, 4.0);
      Person carolf = new Faculty("Carol", 3, 2016);

      /* Notice we are using Faculty variable rather than Person */
      Faculty drSmithf = new Faculty("DrSmith", 4, 2010);

      if (bobp.getClass() == teds.getClass()) {
         System.out.println("1. bob and ted associated with same getClass value");
```

```
    }

    if (bobp instanceof Person) {
       System.out.println("2. bob instance of Person");
    }

    if (teds instanceof Student) {
       System.out.println("3. ted instance of Student");
    }

    if (teds instanceof Person) {
       System.out.println("4. ted instance of Person");
    }

    if (bobp instanceof Student) {
       System.out.println("5. bob instance of Student");
    }

    if (carolf instanceof Person) {
       System.out.println("6. carol instance of Person");
    }

    if (carolf instanceof Student) {
       System.out.println("7. carol instance of Student");
    }

    /* Following will not compile (compare against previous one) */
    /*
     * if (drSmithf instanceof Student) {
     * System.out.println("drSmith instance of Student"); }
     */
  }
}
```

In this class, we're declaring three objects, `bobp, teds`, and `carolf` each of which have type `Person`. The variable `bobp` refers to a person, `teds` refers to a `Student`, and `carolf` refers a `Faculty` object. We also declare `drSmithf` to be of type `Faculty`, and `drSmithf` also refers to a `Faculty` object.

Now, which of these `if (...)` clauses will evaluate to "true," and which will evaluate to false?

- The conditional `bobp.getClass() == teds.getClass()` evaluates to false. Why? Because `bobp` is a `Person`, whereas `teds` is a `Student`. Thus, the first print statement isn't printed.
- The conditional `bobp instanceof Person` evaluates to true since `bobp` is a Person.
- The conditional `teds instanceof Student` evaluates to true since `teds` refers to a `Student`.
- `teds instanceof Person` evaluates to true since `teds` refers to a `Student`, and a `Student` extends the `Person` class.
- `bobp instanceof Student` is false since `bobp` has a `Person` object, and `Person` does not extend `Student`.
- `carolf instanceof Person` evaluates to true since the `Faculty` class extends `Person` class.
- `carolf instanceof Student` is false since `Faculty` does not extend `Student`.
- `drSmithf instanceof Student` is false since `drSmithf` is a `Faculty` object and also refers to a `Faculty` type. In fact, a `Faculty` object can never refer to a `Student` object, so the Java compiler recognizes this and issues a compile-time error (the conditional can never be true).

6 Wednesday, September 11, 2019

Upcasting and Downcasting

When casting in the context of inheritance, there are two different types of casting: upcasting and downcasting. What are the differences?

- **Upcasting** is when we cast a subtype to a supertype. For example, we could cast a `Student` object to a `Person` object in order to perform a task that operates on people.
- **Downcasting** is just going in the other direction: casting a supertype to a subtype.

Upcasting is casting an object to something more general, while downcasting is casting an object to something more specific.

Upcasting is always allowed. There is no problem in treating a subclass object as its superclass type due to the "is-a" relationship exhibited by the sub and superclasses. Upcasting can automatically be done by the compiler in some instances. For example, if we have a function that takes in a `Person`, we could pass in a `Student`, and the compiler would automatically cast the `Student` as a `Person`.

On the other hand, downcasting isn't always allowed. How do we know when it's allowed? With the `instanceof` keyword. Consider the following example:

```java
package university;

/* This code generates a java.lang.ClassCastException */

public class UpCastingDownCasting {
   public static void main(String[] args) {
      Person personRefVariable;
      Person teds = new Student("Ted", 2, 2000, 4.0);
      Student StudentRefVariable;
      GradStudent GradStudentRefVariable;

      // Same type
      personRefVariable = teds;

      // Does not compile as teds may not be a Student; notice we defined teds
      // as a Person variable not a Student
      // StudentRefVariable = teds;

      // Downcasting
      // OK as teds is actually a Student; no run-time error
      StudentRefVariable = (Student) teds;

      // Downcasting
      // run-time error (ClassCastException); ted isn't a graduate student
      GradStudentRefVariable = (GradStudent) teds;
   }
}
```

Note that it's valid to assign `personRefVariable` to `teds` since they have the same type. On the other hand, we wouldn't be allowed to downcast `StudentRefVariable` to `teds` without a cast since they are not of the same time. Doing so would result in a compile-time error. In a similar manner, it wouldn't be valid to assign `GradStudentRefVariable` to `teds` since `ted` is not a graduate student. This would result in a

ClassCastException at runtime.

Safecasting is a process by which we perform various checks in our code in order to ensure that we aren't performing any illegal casts. We want to perform safecasting whenever possible.

The following code illustrates how we can perform safecasting:

```
package university;

import java.util.*;

public class SafeDownCasting {
   public static void main(String[] args) {
      ArrayList<Person> list = new ArrayList<Person>();

      list.add(new Person("John", 1));
      list.add(new Student("Laura", 20, 2000, 4.0));
      list.add(new Faculty("DrRoberts", 30, 1970));

      for (int i = 0; i < list.size(); i++) {
         Person obj = list.get(i);
         System.out.print(obj.getName());
         if (obj instanceof Student) {
            Student student = (Student) obj;
            System.out.println(", admission year is " + student.getAdmitYear());
         } else {
            System.out.println();
         }
      }
   }
}
```

Note how we use the `instanceof` operator to verify the "is-a" relationship between the objects we are casting between. Ultimately, this helps us prevent ClassCastExceptions. There is no need to check anything when we're upcasting.

Abstract Classes

Motivating Abstract Classes

Suppose we want to define several objects to represent different shapes, each of which have several methods whose implementation will be identical, like, perhaps, `getColor()` could be used to obtain the color of the shape. However, suppose we want to add a `draw()` method inside of each of the classes, which draws the shape. Clearly, this method would be dependent on what shape the function is being called on. This means that it wouldn't make sense to make a single implementation for all of the shapes.

But, if we didn't have a `draw()` method in the `Shape` superclass, our code wouldn't compile if all of our shapes were stored in an array. This is due to the compiler not being able to tell whether or not the object stored at that array index truly does have a `draw()` method. Thus, we would need to add a `draw()` method to our `Shape` superclass for the sole purpose of it being overriden at a later point.

One solution to this problem is using interfaces; however, implementing an interface would *require* a class to implement all of the interface's methods. This isn't exactly what we want, because we might not want to be able to draw every shape.

Another problem that might arise comes from the fact that classes can be instantiated. Thus, one would be able to instantiate the `Shape` class, ultimately "creating" a new shape. This could lead to complications if one were to use the `draw()` method on that newly created shape.

This is an example in which it is a good idea to use **abstract classes**, which can be created using the `abstract` keyword. Abstract classes permit us to provide the implementation of common methods, like `getColor()`, while also only providing the function header for methods that might have different implementations. Unlike interfaces, abstract classes aren't implemented — they are extended with the `extends` keyword. Also, an abstract class cannot be instantiated directly; this means that a user wouldn't be able to "create" a new shape.

We can also use the `final` keyword on a method in order to prevent it from being overridden. This can be helpful when we're using abstract classes; however, it doesn't have to be used with them.

Inheritance versus Composition

There are two primary ways in which we can create a complex class from another: inheritance and composition. In the inheritance case, we're extending a parent class in order to solve the problem at hand, whereas in the composition case, we're defining an instance of an object to solve the problem at hand.

As an example, suppose we have `Permit` and `Person` classes. Would it be better to make the `Person` class have a `Permit` object, or would it be better to make the `Person` class extend the `Permit` class? The former describes composition, whereas the latter describes inheritance.

A straightforward way to determine which of inheritance or composition should be used is to establish whether there is an "is-a" or a "has-a" relationship. If there is an "is-a" relationship (i.e. a `Circle` is a `Shape`), we should be using inheritance. On the other hand, if there is a "has-a" relationship (i.e. a `Person` has a `Permit`), we should be using composition.

In general, composition is a better design choice. This is particularly true when we have a choice between either design pattern — we should always prefer composition over inheritance.

In composition, we explicitly have an instance variable of the given object type, which isn't the case for inheritance.

Multiple Inheritance

Multiple inheritance is a feature of some object-oriented programming languages in which one class can inherit characteristics and features from more than one parent class. Essentially, multiple inheritance would allow a single class to extend more than one parent class. How can this be useful? Suppose we have a class called `StudentAthlete`, and we have two other classes called `Student` and `Athlete`. It would be useful to have the `StudentAthlete` class extend both `Student` and `Athlete` to inherit those classes' properties.

Unfortunately, Java doesn't support multiple inheritance. However, it does permit implementing multiple interfaces. We can simulate the notion of multiple inheritance in Java by simultaneously extending a class and implementing an interface. Essentially, we could just make one of the classes that we would be extending into an interface, and we would be able to implement that interface without restriction. It does not matter which class becomes the interface.

7 Friday, September 13, 2019

Introduction to Exceptions

In Java, **exceptions** are used in the event that execution of a program that disrupts the normal, expected flow of code execution (typically, we say an exception is "thrown"). For example, attempting to divide by zero or attempting to open a non-existent file would cause an exception. Internally, an exception is represented as an object, which has various values and built-in method. When an error occurs, an exception object is created, and this object is sent to a **catch clause**, which performs the necessary processing. The mapping of the exception to the catch clause is done internally by Java.

What are the benefits of throwing exceptions?

- Allows for debugging and error differentiation.
- Typically, exception handling code is separate and easy to distinguish.

The following code segment illustrates a basic exception-throwing example:

```
package exceptionsEx;

import javax.swing.JOptionPane;

public class Fundamentals {
   public static void evenOdd() {
      String prompt = "Enter integer value";

      try {
         int value = Integer.parseInt(JOptionPane.showInputDialog(prompt));
         String message = "odd";
         if (value % 2 == 0)
            message = "even";
         System.out.println(message);
      } catch (NumberFormatException e) {
         JOptionPane.showMessageDialog(null, "You must provide an integer value");
      }
   }

   public static void main(String[] args) {
      evenOdd();
   }
}
```

In this program, we simply call a function called `evenOdd()`, which takes in a user-inputted integer, and determines whether it is even or odd. But what happens if the user *doesn't* input an integer? Typically, our code would crash. But in this code segment, we foresee this error, and we put the "dangerous" portion of the code (that is, the code segment that can lead to the program crashing) into a `try { ... }` block. This is immediately followed by a `catch ( .. ) { .. }` block, where the type of error that could occur is specified.

Essentially, when we're in the `try { ... }` block, our compiler will always be on the lookout for a `NumberFormatException`. If this exception is ever thrown, our flow of execution will be interrupted, and nothing after the line that caused the exception to be thrown will be executed — we will jump and execute the code in the `catch( ... ) { ... }` block. The code will continue to execute after the `catch ( ... ) { ... }` block afterwards (but in our code segment, there's nothing afterwards anyways).

To summarize,

- The code in the `try { ... }` clause should be the code that can potentially throw an exception. If an exception is thrown, the flow of execution is interrupted (we jump to the catch block).
- The code in the `catch ( .. ) { ... }` clause specifies what exception to look out for (we can look out for more than one exception by having multiple catch clauses). The code inside of the `catch ( ... ) { ... }` clause's code block will be executed if this type of exception thrown.

A comprehensive list of exceptions can be found here: https://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html.

In the above example, we wait for exceptions to be thrown by the compiler, and we catch them in our code block. However, as the program writer, you are able to throw exceptions yourself with the general syntax `throw new [exception]`. Why would we ever want to do this? For example, if you wanted to create an empty method called `getIntValue()` that you planned to implement at another time, you could use the following code:

```
public static int getIntValue() {
    throw new UnsupportedOperationException("You must implement this method");
}
```

This would allow you to continue implementing other portions of the code (without a compile-time error) without returning a value.

The Finally Block

There is another component to exceptions that we haven't discussed yet: **finally blocks**. The finally block follows the `try { ... } catch ( ... ) { ... }` blocks, and it contains code that is always executed (we can put code common to both the `try { ... }` and `catch ( ... ) { ... }` blocks inside of our `finally { ... }` block).

Here's an example:

```
package exceptionsEx;

import javax.swing.JOptionPane;

public class ReadNegativeValue {
  public static int totalAttempts = 0;

  public static int getNegativeInteger() {
     String prompt = "Enter negative integer value";
     String errorMessage = "You must provide a negative integer value";

     while (true) {
        int value = 0;
        try {
           value = Integer.parseInt(JOptionPane.showInputDialog(prompt));
           if (value < 0) {
              return value;
           }
           JOptionPane.showMessageDialog(null, errorMessage);
        } catch (NumberFormatException e) {
           JOptionPane.showMessageDialog(null, "In method catch clause: " + errorMessage);
        } finally {
           totalAttempts++;
        }
```

```
    }
  }

  public static void main(String[] args) {
    String report = "Value Provided: " + getNegativeInteger();

    report += "\nTotal Attempts: " + ReadNegativeValue.totalAttempts;
    JOptionPane.showMessageDialog(null, report);
  }
}
```

In this example, we keep on reading integers until a negative integer is entered. The static integer variable `totalAttempts` is incremented whether an exception is thrown or not. Another example in which a `finally` block could be used is when we're dealing with file I/O. We should close the file being processed in the `finally` block, whether an exception was thrown or not.

Checked and Unchecked Exceptions

Exceptions can be divided into two distinct categories: **checked exceptions** and **unchecked exceptions**. What are the differences?

- Checked exceptions are "forced" by the compiler, and we should be able to continue executing our program if they are provided. As an example, suppose we try to open a file that doesn't exist. This is a checked exception, and we should be able to handle this exception (i.e. continue executing our program), if it comes up.
- Unchecked exceptions are not forced by the compiler; they are typically more serious exceptions, like `NullPointerException` and `IndexOutOfBoundsException`, and they don't need to be caught. Pretty much, these are errors caused by the programmer writing bad code.

We can create our own exceptions by extending the `Exception` class. By default, this creates a checked exception. Here's an example that demonstrates this:

```
package exceptionsEx;

/* Checked exception */
public class NotPayingAttentionException extends Exception {
  private static final long serialVersionUID = 1L;

  public NotPayingAttentionException(String message) {
    super(message);
  }
}
```

Here, we have defined a `NotPayingAttentionException`, which is a checked exception (as it extends the `Exception` class). By default, the constructor provides us with the message that is displayed when this exception is thrown. How do we use this exception? The key thing to remember is that checked exceptions *must* be thrown in a `try` block. Why? Because Java recognizes that programs should be able to handle checked exceptions, and there should be some way to continue execution of the program (with the `catch` block). The line `throw new NotPayingAttentionException("Exception");`! itself would not compile. It needs to be placed inside of a `try` block.

8 Monday, September, 16, 2019

Nested Types

Java permits us to place classes inside of other classes in order to indicate that the two classes are distinct but closely related. Consider the following code segment:

```
public class MyOuterClass {
    private int x;
    private class MyInnerClass {
        private int y;
        void foo () { x = 1; }
    }
    void bar() {
        MyInnerClass ic = new MyInnerClass();
        ic.y = 2;
    }
}
```

This is valid code, and it compiles. `MyInnerClass` is a class that's placed inside of `MyOuterClass`, which indicates that the two classes are closely related. Once we've created an inner class, we can create instances of the inner class inside of the outer class as shown in the function `void bar() { ... }`. This is done in the usual way — with the `new` keyword.

Something that might seem strange, however, is the outer class's ability to access private members of the inner class. For example, as shown in the `bar()` function, we're able to access the private field `y` outside of the inner class. This would not be possible if `MyInnerClass` were not an inner class. This allows the inner and outer classes to easily share and exchange data.

Scoping can sometimes be confusing, so here is another example:

```
public class MyOuter {
    int x = 2;
    private class MyInner {
        int x = 6;
        private void getX() { // Inner class method.
        int x = 8;
        System.out.println(x); // Prints 8
        System.out.println(this.x); // Prints 6
        System.out.println(MyOuter.this.x); // Prints 2
        }
    }
}
```

Here, we have an outer class `MyOuter` and an inner class `MyInner` The inner class is able to access everything in the outer class, whether it is private or not. The first `System.out.println(x)` statement accesses the declaration of `x` as 8 since that is the most recent declaration in its current scope. If we use the `this` keyword, we can refer to properties of the current object and access the value 6. In a similar manner, if we need to access the value of x in the outer class, we can use the `this` keyword on the outer class.

Why do we need inner classes? The following example illustrates why inner classes can be so useful.

Below, we will present a program that uses `Player` and `Team` objects. Firstly, here's an implementation of the `Player` class:

```
package teamV2;
```

```
public class Player {
   private String name;
   private String address;

   public Player(String name, String address) {
      this.name = name;
      this.address = address;
   }

   public String toString() {
      return "Player [name=" + name + ", address=" + address + "]";
   }
}
```

There's not too much to this: there's a constructor that takes in a name and an address, and there's a `toString()` method to print the relevant details about a player.

Now here's the `Team` class:

```
package teamV2;

public class Team {
   public Player[] list; /* bad: it should not be public */
   public int size; /* bad: it should not be public */

   public Team(int maxSize) {
      list = new Player[maxSize];
      size = 0;
   }

   public boolean add(Player player) {
      if (size < list.length) {
         list[size++] = player;
         return true;
      }

      return false;
   }

   public String toString() {
      String answer = "";

      for (int i = 0; i < size; i++) {
         answer += list[i] + "\n";
      }

      return answer;
   }
}
```

This is a little bit longer than the `Player` class. As we can see, each `Team` is composed of an array of `Player` objects. There's a constructor that initializes the `Team` object with its maximum size, and there's an `add()` method that adds a `Player` object to the array in the `Team` object. Finally, there is a `toString()` method, which simply calls the `toString()` method on each `Player` object by using `System.out.println()`.

Now here is a driver program that uses both of these classes:

```
package teamV2;

import java.util.Iterator;

public class Driver {
   public static void main(String[] args) {
      int maxSize = 10;
      Team team = new Team(maxSize);

      team.add(new Player("John", "College Park"));
      team.add(new Player("Rose", "Silver Spring"));
      team.add(new Player("Linda", "Bethesda"));

      Iterator<Player> it = new TeamIterator(team);
      while (it.hasNext()) {
         System.out.println(it.next());
      }
   }
}
```

Here, we create a `Team` object named `team`, and we add three `Player` objects to our team. Finally, we use our `team`'s `toString()` method to print the details about our team through a `TeamIterator` object. Below is an implementation of the `TeamIterator` class:

```
package teamV2;

import java.util.Iterator;

public class TeamIterator implements Iterator<Player> {
   private Team team;
   private int pos;

   public TeamIterator(Team team) {
      this.team = team;
      pos = 0;
   }

   public boolean hasNext() {
      return pos < team.size;
   }

   public Player next() {
      return team.list[pos++];
   }

   public void remove() {
      throw new UnsupportedOperationException("iterator remove not implemented");
   }
}
```

Here, we've implemented a `TeamIterator` class, which implements the `Iterator` interface. As enforced by the interface, we implement the `hasNext()` and the `next()` methods. The integer variable `pos` represents where we are currently pointing to in the array.

While this program compiles and works fine, it is not the optimal implementation. Why? Firstly, the `TeamIterator` class has to access the `list` and `size` variables from the `Team` class, which prevents these

variables from being private (they are currently declared as public). Secondly, the `TeamIterator` class is different from the `Team` class, while the two classes are actually strongly related to each other. One way to fix the first issue would be by making the variables private and adding public getter methods to retrieve these values, but this still leaves the `Team` and `TeamIterator` classes as if they are two completely distinct classes.

With inner classes, we can resolve both of these issues. More specifically, we can place the `TeamIterator` class inside of the `Team` class as an inner class as follows:

```
package teamV3;

import java.util.Iterator;

public class Team {
   private Player[] list; /* good: now is private */
   private int size; /* good: now is private */

   public Team(int maxSize) {
      list = new Player[maxSize];
      size = 0;
   }

   public boolean add(Player player) {
      if (size < list.length) {
         list[size++] = player;
         return true;
      }

      return false;
   }

   public String toString() {
      String answer = "";

      for (int i = 0; i < size; i++) {
         answer += list[i] + "\n";
      }

      return answer;
   }

   /* We need this method to provide access to the Iterator object */
   public Iterator<Player> iterator() {
      return new TeamIterator();
   }

   public class TeamIterator implements Iterator<Player> {
      private int pos = 0;

      /* Notice that we no longer has to pass team */
      /* as we can access team object data directly. */
      /* Notice how the code has been simplified dramatically */

      public boolean hasNext() {
         return pos < size;
      }
```

```
    public Player next() {
       return list[pos++];
    }

    public void remove() {
       throw new UnsupportedOperationException("iterator remove not implemented");
    }
  }
}
```

Now, we don't need to create additional getter methods to access the variables since inner classes have access to the variables in their outer classes. Also, our `TeamIterator` is now *part of* our `Team` class, which clearly indicates that the two classes are strongly related to each other and makes our code more readable. If we wanted to be able to use the enhanced for-loop and write something like, `for (Player p : team)`, we could implement the `Iterable` interface. This interface requires us to provide a method called `iterator()` which simply returns an Iterator object. The relevant portion of code is shown below:

```
public class Team implements Iterable<Player> {
   /* ..... Other methods not shown ..... */

   public Iterator<Player> iterator() {
      return new TeamIterator();
   }
   /* ..... Inner class not shown ..... */
}
```

Side-note: It is possible for an inner class to extend its outer class.

9 Wednesday, September 18, 2019

More on Inner Classes

Last time, we introduced the notion of inner classes. Recall that inner classes are important because it allows us to access data directly in a class that is strongly related to another class. In the previous lecture, we improved a `Team` class by moving its iterator inside of that class and subsequently implementing the `Iterable` interface so that we can use the enhanced for loop with this object.

We can actually simplify this code even further by declaring the `TeamIterator` that we are

```
package teamV6;

import java.util.Iterator;

public class Team implements Iterable<Player> {
   private Player[] list; /* good: now is private */
   private int size;   /* good: now is private */

   public Team(int maxSize) {
      list = new Player[maxSize];
      size = 0;
   }

   public boolean add(Player player) {
      if (size < list.length) {
         list[size++] = player;
         return true;
      }

      return false;
   }

   public String toString() {
      String answer = "";

      for (int i = 0; i < size; i++) {
         answer += list[i] + "\n";
      }

      return answer;
   }

   /* Relying on anonymous inner class */
   public Iterator<Player> iterator() {
      return new Iterator<Player>() {
         private int pos = 0;

         public boolean hasNext() {
            return pos < size;
         }

         public Player next() {
            return list[pos++];
         }

         public void remove() {
```

```
            throw new UnsupportedOperationException("iterator remove not implemented");
        }

    };
  }
}
```

In our `iterator()` method, we're now declaring `TeamIterator` class as we return it. In Java, this is known as an **anonymous inner class**. The adjective "anonymous" comes from the fact that there is no way to reference this class since we haven't given it a name.

Anonymous Inner Classes

An **anonymous class** is a class that is both defined and instantiated in a single, succint expression. Similar to an anonymous inner class (see above), anonymous classes aren't given names, which means that we can't reference them later on. This is particularly useful when we only want to create one instance of a class (so it wouldn't be very helpful to explicitly define a class, which is usually done for later references).

We can demonstrate the uses of anonymous classes with an example. Consider the following `Tetris` class:

```
package anonymousClasses;

public class Tetris {
  private int score;

  public int getScore() { return score; }
  public void setScore(int score) { this.score = score; }
  public void rotatePiece() {
    System.out.println("Rotating");
  }
  public void dropPiece() {
    System.out.println("Dropping");
  }
}
```

This class simply simulates some basic functions that we might want to have when playing a game of Tetris. Now suppose we wanted to create a `SuperTetris` object, which is very similar to the `Tetris` class, except that it should say "Super Tetris Dropping" instead of "Dropping" when a piece is being dropped. This can be done as follows:

```
package anonymousClasses;

public class SuperTetris extends Tetris {
  public void dropPiece() {
    System.out.println("Super Tetris Dropping");
  }
}
```

But what happens if we only want one instance of the `SuperTetris` class? It seems pointless to create an entire new class just to create one instance of the class and never use the class again. This is where we can use an anonymous class so that we can define the class implicitly. Here's an example of how this can be done in a driver:

```
package anonymousClasses;
```

```
public class Driver {
    public static void main(String args[]) {
        /* Anonymous class, based on the tetris class. */
        Tetris superTetris;
        superTetris = new Tetris() {
            /* Overriding method. */
            public void dropPiece() {
                System.out.println("Super Piece Dropping");
            }
        };

        /* Using the object. */
        superTetris.dropPiece();
    }
}
```

The above code compiles, and it works just how we want: the output is "Super Tetris Dropping." Here, we've declared `superTetris` with an anonymous inner class, which has allowed us to avoid explicitly creating a new class. Note that the declaration of an anonymous inner class requires a semicolon after the final closing bracket.

It isn't necessary for an anonymous class to be an inner class.

Static Nested Classes

We've already seen inner classes (also known as non-static nested classes), but it turns out that static nested classes have different functionalities than non-static ones. In particular, non-static nested classes cannot be instantiated without creating an instance of the outermost class. This is not true in the case of a static nested class. That is, we are able to instantiate a static nested class without an instance of the outer class. So, why would we ever use a static nested class if they can be instantiated and used independently from their outer class? One of the primary reasons is the scope of the nested class is contained in the outer class. Another reason is that, if the inner class is supporting a functionality of the outer class, it is much more readable to have the two classes together.

Copying Objects

In Java, there are three ways in which we can copy an object. To demonstrate what is happening in each of these three cases, let's suppose the variable y is a reference to the object z, and we want x to be a copy of y.

- A **reference copy** of y would simply make x point to the same memory location as y. This is done in Java with the expression `x = y`. Any modification to either one of x or y will be reflected in both references.
- A **shallow copy** duplicates as little as possible. For example, a shallow copy of a collection would copy y's structure into x, but it wouldn't copy the elements. This means that x and y will represent two different collections, but they will share the individual elements. One way in which this is done in Java is the `.clone()` method. This performs a "pure-byte" copy.
- A **deep copy** is very straightforward: it makes duplicates *everything*. If we're duplicating a list, the structure will be copied and so will the individual elements. One way in which this type of copy can be performed in Java is by iterating through a list and inserting elements into the copy list manually.

Let's suppose we're duplicating an array of strings. We want to duplicate the array of strings so that modifications to one of the arrays isn't reflected in the other array. We don't need a deep copy to do this — since strings are immutable, a shallow copy is enough.

As mentioned above, we can create shallow copies of an object using the `.clone()` method. The `Object` class provides a `clone()` method. In order to clone methods of a particular class, we need to override the `clone()` method and call the `Object` class's clone method.

Below is an example of cloning. Consider the following `Mouse` class:

```
package cloning;

public class Mouse implements Cloneable {
   private String type;
   private int xPos, yPos;

   public Mouse(String type) {
      this.type = type;
      xPos = yPos = 0;
   }

   public void moveMouse(int xPos, int yPos) {
      this.xPos = xPos;
      this.yPos = yPos;
   }

   public String toString() {
      return type + "-> xPos: " + xPos + ", yPos: " + yPos;
   }

   /* Notice the return type */
   @Override
   public Mouse clone() {
      Mouse obj = null;

      try {
         obj = (Mouse) super.clone();
      } catch (CloneNotSupportedException e) {
         e.printStackTrace();
      }

      return obj;
   }
}
```

Here, we've defined the `clone` method, which overrides the `clone()` method provided to us by the `Object` class. When overriding a method in Java, we're able to define the return type to be a subtype of the return type of the method that's being overridden. This is known as a **covariant return type**, and it describes what's happening in our `clone()` class aboves.

Note that we've also called the `Mouse` superclass's `clone()` method, which performs a shallow copy of the primitive types when it can.

10 Friday, September 20, 2019

Exam 1 is today.

11 Monday, September 23, 2019

Lambda Expressions

Suppose we have the following interface that provides us with the prototype of a `compute` function:

```
package lambda;

public interface Task {
   public int compute(int x);
}
```

One way to use this interface would be to implement it, which would make the compiler force the class to provide a declaration of the function `compute`. This is a great idea if we're planning to create many instances of the class, but what if we only plan to instantiate this class once? As we've learned, we can solve this problem with anonymous classes. An alternative way to solve this problem is with **lambda expressions**, which permit us to write anonymous functions in a simpler way. This means that lambda expressions aren't "necessary" in the sense that there isn't something that can only be done with lambda expressions, but they can be very convenient to use.

The general syntax for a lambda expression is `(type1 parameter1, type2 parameter2, ...) -> expression` or `(type1 parameter1, type2 parameter2, ...) -> {statements;}`. The first form is used when we're returning a value, and the second form is used otherwise.

The following code demonstrates how we can use lambda expressions in the context of our `Task` interface.

```
package lambda;

public class LambdaBasics {
   public static void main(String[] args) {
      Task anonymousVersion = new Task() {
         public int compute(int x) {
            return x + x;
         }
      };
      System.out.println(anonymousVersion.compute(10));

      Task lambdav1 = (int x) -> {
         return x + x;
      };
      System.out.println(lambdav1.compute(10));

      Task lambdav2 = (x) -> x + x;
      System.out.println(lambdav2.compute(10));

      Task lambdav3 = x -> x + x;
      System.out.println(lambdav3.compute(10));

      AnotherTask lambdav4 = () -> 10;
      System.out.println(lambdav4.analyze());

      Processor lambdav5 = (int x, float y) -> x * y;
      System.out.println(lambdav5.increase(10, 5));

      pdata((x, y) -> x * y);
   }
```

```
  public static void pdata(Processor p) {
     System.out.println(p.increase(10, 60));
  }
}
```

The above code firstly declares an anonymous inner class that implements the `Task` interface. This is done by providing a declaration of the `compute` function. Subsequently, we use a lambda expression to declare `lambdav1` using the first "general syntax" form that was specified earlier. Next, we declare a second lambda expression named `lambdav2` using the second "general syntax" form specified earlier. We can note that the key difference between these two declarations is that the first form contains a single expression after the arrowhead, while the second form can contain several statements after the arrowhead (if there's a single statement, like above, the compiler can infer that it needs to return it). Why isn't the `int` type specified in the second declaration? It turns out that our compiler can infer the type to be an `int` based on the context in which the lambda expression is used in.

We can simplify `lambdav2` even more. The lambda expression `lambdav3` drops the parentheses around the parameter of `lambdav2`; parentheses are not necessary when there is only a single parameter.

Now let's introduce another interface named `AnotherTask`:

```
package lambda;

public interface AnotherTask {
  public int analyze();
}
```

This interface provides us with the prototype for an `analyze()` function that returns an integer and doesn't take in any parameters.

As shown in the declaration of `lambda4` in our driver code, we can declare a lambda expression for `analyze` as well by using an empty open-closing parentheses pair, which indicates that there are no parameters.

Finally, let's look at one last interface named `Processor`:

```
package lambda;

public interface Processor {
  public float increase(int x, float y);
}
```

In our driver code, we've declared a lambda expression named `lambdav5` that can be used for the `increase` function.

Finally, in our driver, we show an application of lambda expressions. The `pdata` function takes in a `Processor` type, and it prints out the return value of the `p.increase(10, 60)` call. Instead of creating an instance of a `Processor` and subsequently passing that object into the function call, we can instead use a lambda expression (as done in the driver) in order to avoid writing so much code. This is also helpful since we don't need to declare one-time-use variables.

Revisiting Shallow Copies

Recall the following `Mouse` class that we used to discuss shallow copying:

```
package cloning;
```

```
public class Mouse implements Cloneable {
   private String type;
   private int xPos, yPos;

   public Mouse(String type) {
      this.type = type;
      xPos = yPos = 0;
   }

   public void moveMouse(int xPos, int yPos) {
      this.xPos = xPos;
      this.yPos = yPos;
   }

   public String toString() {
      return type + "-> xPos: " + xPos + ", yPos: " + yPos;
   }

   /* Notice the return type */
   @Override
   public Mouse clone() {
      Mouse obj = null;

      try {
         obj = (Mouse) super.clone();
      } catch (CloneNotSupportedException e) {
         e.printStackTrace();
      }

      return obj;
   }
}
```

When we create a `Mouse` object, and we use the `.clone()` method to create a copy of that mouse, a shallow copy is created. This means that primitive types are copied, while only the memory addresses of non-primitive types are copied.

In our `Mouse` class, we call `super.clone()` inside of our `.clone()` method. This call returns an `Object` type, but we cast it as a `Mouse` and return a `Mouse` type. Note that this still counts as function overriding (even though we're returning a `Mouse` type instead of an `Object` type) as this is a case of covariant return types. At this point, we're done creating our copy, and we can return this value. This returned value will be independent of the original object since the strings in the class are immutable, and the remaining variables are all primitives, so they are all copied.

So when's an example in which the default `.clone()` method is not enough to produce two independent copies (i.e. copies with the property that changing one does not affect the other)? One example in which `.clone()` wouldn't be enough is if we have a new class named `Computer` that has a `Mouse` object as one of its instance variables. In this case, the `.clone()` method will make both `Computer` objects point to the `Mouse` object. However, since the `Mouse` object contains two primitive types (integers) that can be changed, changing an integer in the `Mouse` object will be reflected as a change in both `Computer` objects.

How do we fix this problem? We can override the `.clone()` method in the `Computer` class, check if a `Mouse` object is present (not null). If it is present, we can call the `.clone()` method on the mouse (which we've already explained does what we want).

Garbage Collection

One of the benefits of using Java as a programming language is that we don't have to worry about freeing memory that we are no longer using in the program. More precisely, if we remove all references to an object, this object becomes **garbage** since it is useless and can no longer affects the program. This is not true in some other programming languages, like C.

This entire process is automated for us in Java, and the process is called **garbage collection**. Essentially, we're able to reclaim memory used by unreferenced objects when we're running low on memory. This process is performed periodically by Java, but it is not guaranteed to occur.

One way in which garbage collection is performed is through the use of **destructors**, which are void methods with the name `finalize()` that contain the necessary actions to be performed when an object is freed. Destructors are only invoked if garbage collection occurs. Here's an example of what a destructor might look like:

```
class Foo {
    void finalize() { ... } // destructor for Foo
}
```

12 Wednesday, September 25, 2019

Initialization Blocks

An **initialization block** is a grouping of code to be executed at a pre-defined time. There are two primary types of initialization blocks:

1. **Static initialization blocks** are run whenever a class is loaded.
2. **Non-static initialization blocks**, which are also known as **instance initialization blocks** are run whenever a constructor is called.

When can static initialization blocks be helpful? Let's suppose we have a `Person` class whose implementation is shown below:

```
package staticBlock;

import java.util.Calendar;
import java.util.Date;
import java.util.TimeZone;

public class Person {
   private String name;
   private Date birthDate;
   private static final Date MILLENIUM;

   static {
      Calendar gmtCal = Calendar.getInstance(TimeZone.getTimeZone("GMT"));
      gmtCal.set(2000, Calendar.JANUARY, 1, 0, 0, 0);
      MILLENIUM = gmtCal.getTime();
   }

   public Person(String name, Date birthDate) {
      this.name = name;
      this.birthDate = birthDate;
   }

   @Override
   public String toString() {
      return "Person [name=" + name + ", birthDate=" + birthDate + "]";
   }

   public boolean bornBefore2000() { // FASTER!
      return birthDate.before(MILLENIUM);
   }
}
```

In this class, we've got a `bornBefore2000()` function that returns true if the `Person` object's birthdate is set before 2000. This is done using the a built-in Calendar package. We also have a static initialization block, which as described previously, is run whenever the class is loaded (note: a class is loaded only once — when a reference is made to it).

What's happening in the static initialization block? Essentially, we're pre-computing the result that's returned in the `bornBefore2000` function. While we could create an equivalent definition of this class by removing the static initialization block and placing its contents inside of the `bornBefore2000` method, it is

faster to just precompute the result whenever a person instance is created since their birthday won't change. Ultimately, this increases efficiency.

In summary, we can precompute various quantities of interest in a static initialization block. Since classes are only loaded once, we should only precompute quantities that don't tend to change (like our date example above).

Here's another illustrative example that clearly conveys how static initialization blocks work:

```
package staticBlock;

/**
 * Initializations executed in order of number
 *
 */
public class VariableInitialization {
   static {
      A = 1;
   }
   static int A = 2;
   static {
      A = 3;
   }
   {
      B = 4;
   }
   private int B = 5;
   {
      B = 6;
   }

   VariableInitialization() {
      System.out.println("A: " + A);
      System.out.println("B: " + B);
   }

   public static void main(String[] args) {
      new VariableInitialization();
   }
}
```

While we probably wouldn't need so many static initialization blocks in practice, this example is simply present to illustrate how static initialization blocks work. The number that the variable is initialized to represents the order of execution when the program is run. Note that even though the statement `A = 1` can occur prior to the declaration of `A` as an integer. This is valid syntax as long as the initialization occurs in a code block.

Next, we can look at the following example, which uses non-static initialization blocks:

```
package nonStaticBlock;

public class Employee {
   private String name;
   private int age;
   private double salary = 100;

   public Employee(String name, int age) {
      this.name = name;
```

```
        this.age = age;
    }

    public Employee() {
        this.name = "NONAME";
        this.age = 0;
    }

    {
        System.out.println("salaryInit1: " + salary);
        salary = 200;
        System.out.println("end processing1");
    }

    @Override
    public String toString() {
        return "Employee [name=" + name + ", age=" + age + ", salary=" + salary + "]";
    }

    {
        System.out.println("salaryInit2: " + salary);
        salary = 400;
        System.out.println("end processing2");
    }

    public static void main(String[] args) {
        Employee employee1 = new Employee("John", 40);
        System.out.println(employee1);

        Employee employee2 = new Employee("Peter", 17);
        System.out.println(employee2);

        System.out.println(new Employee());
    }
}
```

Here, we have an initialization block that prints out the `salary` variable, sets `salary` to 400, and prints "end processing2." This initialization block is executed everytime an `Employee` object is instantiated. Why would we put code in an initialization block instead of just putting it directly in the constructor? It's mostly a matter of style; sometimes it can be messy to have too much code in the constructor.

Generic Classes

The `ArrayList` class permits us to specify what type of object we want to store inside of the list. We can create an `ArrayList` of `String` objects by writing, `ArrayList<String> = new ArrayList<String>()`, and we can similarly create an `ArrayList` of `Integer` objects with `ArrayList<Integer> = new ArrayList<Integer>()`. How does the `ArrayList` class adapt to the different types of objects we want to store? This is done very easily through the use of generics.

A **generic class** is a class that permits us to use one or more type variables. The benefit of using generic classes is clear — we can avoid rewriting duplicate code for very similar tasks. Essentially, we're able to reuse the same code for different data types.

In order to define a generic class, we just need to append the generic type variable(s) to the class name using angled brackets. The general syntax is `ClassName <type variable>` (note how this looks really similar

to the `ArrayList` syntax that we're used to). The `type variable` that is placed in-between the angled brackets can be any letter, but it's customary to use a single uppercase letter, like `E`, `K`, `T`, or `V`.

Once we've defined the generic variable in our class name, we are free to use it anywhere in the class. Here's an example of an implementation of a Queue data structure that uses generics:

```
package queueExample;

import java.util.NoSuchElementException;

public class Queue<T> {
   private T data[] = (T[]) new Object[4];
   private int front = 0, rear = data.length - 1, size = 0;

   public int size() {
      return size;
   }

   public boolean empty() {
      return size() == 0;
   }

   public int length() {
      return data.length;
   }

   public int front() {
      return front;
   }

   public int rear() {
      return rear;
   }

   public T dataAt(int index) {
      return data[index];
   }

   public boolean full() {
      return size() == data.length;
   }

   public T remove() {
      if (empty())
         throw new NoSuchElementException("Queue is empty");
      T result = data[front];
      data[front] = null;
      front++;
      size--;
      if (front == data.length)
         front = 0;

      return result;
   }

   public void add(T t) {
      if (full()) {
         resize();
```

```
      }
      size++;
      rear++;
      if (rear == data.length)
         rear = 0;

      data[rear] = t;
   }

   private void resize() {
      T newData[] = (T[]) new Object[data.length * 2];

      for (int i = 0; i < size; i++)
         newData[i] = data[(i + front) % data.length];
      front = 0;
      rear = size - 1;
      data = newData;
   }
}
```

This example clearly illustrates the usefulness of generic programming. Previously, we would have been able to create a `Queue` class that permits us to store a pre-defined data type, like integers, by creating an array of that type inside of the class. But now, by specifying the class uses a generic type, we're able to create a generic class that works for any non-primitive object.

What are some key things to note from the above implementation?

- We must specify the generic type that we're using when we're opening the class. This is done by "capturing" the generic variable `T` in angled brackets by writing `public class Queue<T>`.
- As shown through the `dataAt()` method, it perfectly fine to return a generic type from a function.
- As shown in the `remove()` and `resize()` functions, it is valid to create other variables with the generic type.

To summarize, without generic programming, we would need to have an implementation for each data type that we wanted to have a `Queue` object for. But with generic programming, we're able to put everything that's common to these implementations in a single class, ultimately reducing the amount of code that we must write.

Introduction to Linked Lists

A **Linked List** is a linear data structure that is composed of several **nodes**, each of which contain some data and a reference to the next node in the list. For example, the following figure illustrates a Linked List consisting of three integers: 12, 99, and 37.

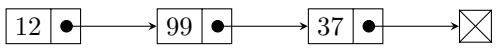


Figure 1: A Linked List

The entire figure above represents a Linked List, and each individual square represents a square. Visually, we represent the end of a Linked List by a null reference (above, this is represented with a crossed out square).

As shown in the above figure, it's important to remember that each node consists of two entities: the data stored in that node as well as a reference to the next node.

In Java, we can implement a Linked List using an inner class to represent the node. Part of an implementation is shown below:

```
package myLinkedList;

public class MyLinkedList<T extends Comparable<T>> { /* Notice the parameter */
   private class Node {
      private T data;
      private Node next;

      private Node(T data) {
         this.data = data;
         next = null; /* do we really need to do this? */
      }
   }

   /* List head pointer */
   private Node head;

   /* We don't actually need it */
   public MyLinkedList() {
      head = null;
   }

   /* Adding at the front of the list */
   public MyLinkedList<T> add(T data) {
      Node newNode = new Node(data);
      newNode.next = head;
      head = newNode;

      return this;
   }
```

As described earlier, each `Node` object has a generic `data` field to store some information as well as `next` field, which stores the next `Node` object in the Linked List. This may seem confusing at first since the definition of a `Node` object includes a `Node` field inside of it. This self-reference is permitted by Java.

13 Monday, September 30, 2019

x

14 Friday, October 4, 2019

Linked List Methods

Recall that last lecture, we started talking about the Linked List data structure. It is important to note that the head of a Linked List data structure isn't actually a node but rather just a reference to the first node in the Linked List. This means that an empty Linked List can be represented with a null head node. In memory, nodes are typically stored in the heap (they are dynamically allocated).

How do we print a Linked List?

Here is the relevant code to do so:

```
/* Prints a Linked List */
public String toString() {
    String result = "\" ";
    Node curr = head;
    while (curr != null) {
        result += curr.data + " ";
        curr = curr.next;
    }
    return result + "\"";
}
```

Essentially, we traverse the Linked List with a node called `curr`. We start traversing from the first node in the Linked List by setting `curr` equal to `head`. Subsequently we add the data of each node into our buffer that we will be printing, and we use the reference present in the current node to access the next node. This process is repeated up until we hit `null`, which indicates the end of a Linked List data structure.

Next, let's look at an `add()` function which appends a value to the front of our Linked List:

```
public MyLinkedList<T> add(T data) {
    Node newNode = new Node(data);
    newNode.next = head;
    head = newNode;
    return this;
}
```

This function be a little bit tricky to understand. The function takes in some `data` of type `T` (it's a generic type!), and it passses this data into the `Node` constructor. The `newNode` object is created with the `data` parameter that is passed in, and the `next` field of the newly created `newNode` object is set to null. We set the `next` field equal to the head of the Linked List (so that the new node is pointing to the front of the Linked List), and we subsequently move the `head` to point to the `newNode`, which ultimately makes the `newNode` the front of the Linked List.

In order to trace what various Linked List functions are doing, it is sometimes helpful to draw Linked List diagrams and modify the edges as they are modified in the function.

As we've seen so far, many Linked List methods must iterate through the entire list. Linked List traversals can be categorized into two noteworthy categories[1]:

- The "**print traversal**" is used when we need to visit every node in the Linked List, and perform some sort of processing on the node. This sort of traversal should be used when we don't need to access previous nodes to complete our task (for instance, if we're printing our list). The general setup for the print traversal is shown below:

[1] These two categories are not standard names.

```
Node curr = head;
while (curr != null) {
    /* Perform some processing. */
    curr = curr.next;
}
```

- The "**Tom and Jerry Traversal**" works by keeping two adjacent references to nodes. The left-most reference allows us to look back and access previous elements, and the right-most reference allows us to perform forward processing. These two pointers move in parallel. This type of traversal is particularly useful when we need a backreference, like when we're reversing a Linked List or adding a node either before or after some element. The general setup for the Tom and Jerry traversal is presented below:

```
Node prev = null;
Node curr = head;
while (curr != null) {
    /* Processing. */
    prev = curr;
    curr = curr.next;
}
```

One can see that the `toString()` method presented earlier uses the print traversal. However, the print traversal isn't enough to do everything that we want. For example, suppose we want to remove all nodes whose data is 10 in a given Linked List. This cannot be done with a simple print traversal since once we've found a node whose data entry is equal to 10, we wouldn't be able to go backwards and set the `next` field of the node that comes before the node with 10 as its data to the node after the node with 10 as its data. This problem can, however, be solved with the Tom and Jerry Traversal since we'll have always have a reference to the previous node.

15 Monday, October 7, 2019

Recall last time we introduced two types of Linked List traversals: The Print Traversal, and the Tom and Jerry Traversal. The first type of traversal consists of a single node that moves across the entire Linked List, and the second type of traversal consists of two nodes that move across the Linked List. These two types of traversals allow us to solve most of the problems we will encounter in this course.

In case it is not clear, the Tom and Jerry Traversal is given its name since we can view our reference to the left-most traversal node (often referred to as the "previous" node) as if it is "chasing" the right-most traversal node (often referred to as the "current" node).

Linked List Methods

Below is an implementation of a Linked List's `delete()` method, which takes in a generic `targetElement` variable and deletes the first node whose `data` entry is equal to `targetElement`. This is done using a Tom and Jerry Traversal through the Linked List. Once we've found a node whose data matches `targetElement`, we can use our reference to the previous node and set the `next` field of the previous node equal to the node after the node that the current node is pointing to.

```
public void delete(T targetElement) {
   Node prev = null, curr = head;

   while (curr != null) {
      if (curr.data.compareTo(targetElement) == 0) {
         if (curr == head) {
            head = head.next;
         } else {
            prev.next = curr.next;
         }
         return;
      } else {
         prev = curr;
         curr = curr.next;
      }
   }
}
```

We can trace this code with the following Linked List:

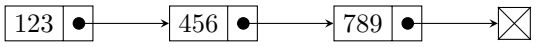


Figure 2: Linked List Pre-Deletion

Suppose we want to delete 456 from our Linked List.

- Before we enter the `while` loop, `curr` will point to the first node (i.e. the node with data 12), and `prev` will point to `null` (it will not be pointing to any node).
- Next, we'll enter the `while` loop since `curr` does not equal null. The comparison of `curr`'s data field with `targetElement` will show that the two quantities are not equal, and we will subsequently move to the next iteration of the `while` loop.
- In the second iteration of the while loop, `prev` will point to the first node (i.e. the node with data 123), and `curr` will point to the second node (i.e. the node with data 456). At this point, the comparison of

`curr`'s data filed with `targetElement` will show that the two quantities are equal, so we will enter the `if` clause. In there, we'll set the `next` field of the first node equal to the third node, which ultimately results in the following Linked List:

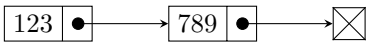


Figure 3: Linked List Post-Deletion

Next, we can look at a `getListWithDataInBetween` function, which takes in a lower and upper bound, and it returns a Linked List with only nodes whose `data` fields are between these bounds:

```
public MyLinkedList<T> getListWithDataInBetween(T start, T end) {
MyLinkedList<T> newList = new MyLinkedList<T>();

if (head != null) {
   Node curr = head, last = null;

   while (curr != null) {
      if (curr.data.compareTo(start) >= 0 && curr.data.compareTo(end) <= 0) {
         Node newNode = new Node(curr.data);
         if (newList.head == null) {
            newList.head = newNode;
         } else {
            last.next = newNode;
         }
         last = newNode;
      }
      curr = curr.next;
   }
}

return newList;
}
```

Once again, this method uses the Tom and Jerry Traversal. This method is fairly similar to the `delete` method. We keep on comparing nodes until we find one that lies between our two bounds. Once such a node is found, we create a new node with the data of the current node we're at, and this node is subsequently set equal to the `head` field of our newly created Linked List.

Note that there is a special case for when we're adding the first node into our Linked List (in this case, we should make the newly created node our head).

Queue Implementation

Recall that a queue is a first-in-first-out (FIFO) data structure. One of the easiest ways in which we can implement a queue is through a Linked List. This is true because we can keep a pointer to the first node, which allows us to remove it easily. Also, we can keep a reference to the last node since it allows us to add elements easily. An alternative implementation would use an array by using a "circular" array in which we keep pointers to where we should add and remove elements.

16 Wednesday, October 9, 2019

When you have a program in Java, there are four areas of memory: the **stack**, **heap**, **static section**, and **code**. The stack is where local variables and recursive function calls are stored.

Introduction to Recursion

Recursion is a strategy for solving problems in which solutions to larger problems depend on solutions to smaller instances of the same problem. More precisely, a **recursive function** is a procedure that calls itself. The "general" pseudocode to solving recursive problems is to solve the problem directly if it is simple or trivial enough, and otherwise simplify the problem into smaller instances of the original problem, small the smaller instances using the same algorithm, and combine the solutions to form the solution to the original problem.

A classical example for demonstrating recursive functions is a factorial function. Recall $n! = n(n-1)(n-2)\cdots(1)$. By rewriting this expression as $n! = n \cdot (n-1)!$ (note that this is an equivalent definition), we can write a program that computes factorials as follows:

```
/* Given an integer n, return n! */
int fact(int n) {
    if (n == 0) return 1; /* Base case */
    return n * fact(n - 1); /* Recursive step */
}
```

In this program, we've used the fact that $n!$ is equal to $n \cdot (n-1)!$. We reduce each problem (i.e. the task of computing a factorial) into a smaller-sized subproblem (computing a factorial of a smaller number) until our problem is very easy to solve (we stop recursing when $n = 0$ is true). The case(s) in which we stop recursing and solve the problem directly are called our **base cases** (for example, here we would say "$n = 0$ is our base case). On the other hand, the case(s) in which we continue recursing is called our **recursive step**. Another way to interpret recursive functions is a function that calls another function that does the same thing as the function we're currently in is doing.

We can prove that recursive algorithms work correctly through **induction**; however, we will not be proving correctness in this class.

What makes recursion possible? Recursion is made possible with the call stack, which is one of the four areas in memory. Essentially, the state of the current procedure or method is saved when the procedure is invoked so that we can easily restore back and continue from where we left off in a function upon encountering a recursive call. *Every* function call gets its own stack space. Here is a diagram that depicts the call stack for a `fact(3)` call:

The above diagram depicts what happens when we call the `fact` function with $x = 3$. The `args` and `x = 3` entries are depicted in the stack just for completeness — this is to emphasize that function calls share the stack space with local variables as well as the function "call" to the main function.

What are the pros and cons of using recursion? Some problems are recursive in nature, so it can be easier to implement a recursive algorithm. This means that recursive algorithms can also be simpler and easier to debug, understand, and maintain. On the other hand, having so many function calls can lead to **function overhead**, which includes the time needed to switch between functions.

Consider the following alternative approach to implementing the `fact` method:

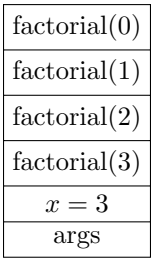


Figure 4: Call Stack for fact(3)

```
int fact(int n) {
    int res = 1;
    for (int i = n; i > 0; i--) { res *= i; }
    return res;
}
```

This implementation of `fact` is actually more efficient than the first recursive definition that we showed. The reason why we showed the first example is just for demonstration purposes.

17 Friday, October, 11, 2019

Recursive Array Functions

In order to get a better understanding of recursion, we can show a few array-based problems in which recursion can be used as solutions.

Given an array and a target value, the following `find()` method returns `true` if the provided target value is present in the array and `false` otherwise. This problem can be solved non-recursively very easily: iterate over all values, and compare each value to the provided target value. But for the sake of demonstration, we will look at the following recursive solution:

```
public static boolean findElementAuxiliary(int[] array, int index, int target) {
   if (index > array.length - 1) {
      /* Empty array segment */
      return false;
   } else {
      if (array[index] == target) {
         return true;
      } else {
         return findElementAuxiliary(array, index + 1, target);
      }
   }
}
```

In this example, we have a base case of when the Boolean expression `index > array.length - 1` is true. When this logical expression evaluates to `true`, our current index exceeds the size of the array. In other words, if we've gotten this far in the array without already having found the value, then we can conclude that the value does not exist in the array. On the other hand, we perform our recursive step when `index <= array.length - 1` is true. In this case, we simply compare the current value at our index to the target value. If there is an equality, we return `true` since we can conclude that our target value exists. If there isn't an equality, we make a recursive call to the same function with an increased index (so we will be looking at the next element in the array on the next recursive call).

Here's another example of a recursive function which counts the number of instances of a provided target element:

```
public static int instancesOfElementAuxiliary(int[] array, int index, int element) {
   if (index > array.length - 1) {
      /* Empty array segment */
      return 0;
   } else {
      if (array[index] == element) {
         return 1 + instancesOfElementAuxiliary(array, index + 1, element);
      } else {
         return instancesOfElementAuxiliary(array, index + 1, element);
      }
   }
}
```

Once again, our base case occurs when the Boolean expression `index > array.length - 1` is true. If this is true, our current index exceeds the capacity of the array, and there are no more instances of the target value. In our recursive step, we compare the current index value to the target. If there is a match, we add 1

to our return value and continue looking at other elements in our array for further matches. On the other hand, if there isn't a match, we simply continue looking at other elements in our array for any further matches.

Intuitively, this recursive function does exactly what we would do if we were counting the number of occurrences of an element in an array by hand. We would start a counter in our head, and we would look at each index of the array in a left-to-right manner. Upon encountering a value that matches our target value, we will increment this counter and continue looking at the next elements.

Something that is important to note is that both of the recursive functions we've looked at so far have had an `index` as a parameter. This parameter allows us to iterate through the array in a recursive fashion since we're able to keep track of where we currently are in the array. Since all initial calls to this function will have the `index` parameter set equal to 0, we can overload our recursive function as follows:

```
public static int instancesOfElement(int[] array, int element) {
     return instancesOfElementAuxiliary(array, 0, element);
}
```

From the `main`, we can now make use of the `instancesOfElement` function, which only takes in an array and a target value to count. For example, we can now write `System.out.println(findElement(data, 349));` instead of always having 0 as one of the third parameters. The `instancesOfElementAux` function, which performs the work to help another function, is known as an **auxillary function**.

Recursive Linked List Functions

In the two previous recursive array functions we've seen, we can note that we stopped iterating upon hitting the end of our array (i.e. when `index > array.length - 1` is true). This is a very common base case when implementing recursive array functions since it permits us to iterate through the array until we reach the end.

In a similar manner, we can create recursive linked list functions whose base case occurs when we reach the end of the list. But since we don't have a `.length` function anymore, we can just compare the current node that we are at to `null`.

Here is a very simple recursive Linked List function, which simply prints a Linked List:

```
public void printListAux(Node headAux) {
   if (headAux != null) {
      System.out.println(headAux.data);
      printListAux(headAux.next);
   }
}
```

We just check whether the current node we are it is `null` (if this is true, then we've hit the end of the Linked List). If it is `null`, we don't do anything — no action is necessary here since we aren't returning anything. On the other hand, if the node is not null, then we print the contents of the current node, and we look at the next node with a recursive call. Consider the following Linked List:

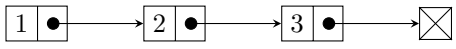


Figure 5: Linked List Recursive Print Demonstration

With the use of our `printListAux` function, we would end up printing `1 2 3` if we passed in a representation of the above Linked List into our function. How can we modify this function to make the Linked List print in

reverse order? In other words, how can we print `3 2 1` instead of `1 2 3`? This can be done by switching the order of the print statement and the recursive call. This way, we will recurse to the base case prior to performing any printing.

Now here is the Linked List analogue of the recursive `find` method that we implemeneted earlier for arrays:

```
public boolean findAux(Node headAux, T target) {
  if (headAux != null) {
     int result = headAux.data.compareTo(target);
     return result == 0 ? true : findAux(headAux.next, target);
  }

  return false;
}
```

This is doing the exact same thing as what our recursive array function performed. Our base case occurs once we've reached the end of the Linked List (i.e. when `headAux` is `null`). When we haven't hit the end of our Linked List, we check whether the current Linked List contains our `target` value. If it does, we return true. Otherwise, we look at the next node. Note that in this implementation, we use the `compareTo` method since our Linked List is implemented using generics.

Something else to keep note of is that it is important to keep our parameter name for the node (which is currently `headAux`) different from the name that we have for our Linked List's `head` in the Linked List class. Otherwise, the compiler will instead be referencing the `head` instance variable.

18 Monday, October 14, 2019

More Recursion with Linked Lists

Now, we can look at a more intricate example than what we have seen before. We will look at a function that takes returns an ArrayList of node entries in which the entries lie in between two provided values:

```
 public ArrayList<T> getDataBetween(T start, T end) {
   ArrayList<T> answer = new ArrayList<T>();

   getDataBetweenAux(head, start, end, answer);

   return answer;
}

private void getDataBetweenAux(Node headAux, T start, T end, ArrayList<T> answer) {
   if (headAux != null) {
      if (headAux.data.compareTo(start) >= 0 && headAux.data.compareTo(end) <= 0) {
         answer.add(headAux.data);
      }
      getDataBetweenAux(headAux.next, start, end, answer);
   }
}
```

Here, we have an auxillary function that takes in a `headAux` node variable, which is used to tell us where we currently are in the array. Furthermore, we take in a `start` and `end` variable, and we want to add all nodes whose value lies between `start` and `end` into our ArrayList.

Similar to the previous functions we've encountered, our base case consists of checking whether we've reached the end of our Linked List. This is done by comparing `headAux` to null. If we've hit the end of our Linked List, no action is necessary since our function is `void` (there is nothing to return). On the other hand, if we haven't hit the end of our Linked List, we can compare the current node's data field and check whether it lies between the `start` and `end` values provided. If it does, we can add this value into our `ArrayList`.

The approach taken by this recursive method differs from what we've seen before. In previous recursive functions, when we were returning an integer, boolean, or simply printing values, we would have a `return` statement that would give us what we wanted from the function. In this function, however, we pass in an empty ArrayList which gets modified into what we want. This implementation allows us to not have to keep track of all of the elements that lie between the `start` and `end` values provided since we're adding the value immediately after this condition is satisfied (this gives our function a "memoryless" property).

Tail Recursion

Tail recursion is a type of recursive function in which we do not need to perform any further actions on a recursive call. In other words, a tail recursive function is a recursive function in which the function calls itself at the end (the "tail" of the function) of the function in which no computation is done after the recursive call. For example, the `getDataBetweenAux` function that we just saw is tail recursive because no further processing is necessary once we've made the last recursive call. On the other hand, the `instancesOfElementAuxillary` function that we saw for arrays is not tail recursive since we might need to add 1 to the value returned by the recursive call.

Why do we care about tail recursion? Mostly because using tail recursion allows us to write more optimized code. Since tail recursive functions don't have any processing after their recursive calls, there is no need to actually store the current function in the stack. Instead, we can save some memory and only store the recursive call.

Common Recursion Problems

Some common recursion-related problems that one may encounter are listed below:

- Infinite recursion: This problem can occur if our recursive step does not simplify the original problem into a smaller-sized subproblem. We can also encounter infinite recursion when we forget to include a base case. For example, something like, `int bad(int n) { if (n == 0) return bad(n-1); }` is an example of infinite recursion since we will keep calling this function without terminating. Eventually, our program halts when our stack runs out of memory from our recursive calls. This is known as a **stack overflow**.
- Efficiency: Just because recursion works doesn't mean it's the best solution. This is particularly true since recursive functions often perform the same work over and over again.

Introduction to Hashing

Hashing is a technique used for storing key-value entries into the array. The process of hashing is utilized by **hash tables**, which are data structures that allow us to ideally insert and retrieve values in constant time. For example, let's say there's a collection of objects and a data structure needs to answer queries of the form "Is this object in the data structure?" (e.g. is this word in the dictionary?). By using hashing, we can do this in less time than it takes to perform a binary search.

How is hashing performed? The idea is to first find a function (known as a **hash function**) that maps the elements of the provided collection to an integer between 1 and N, where N is some number larger than the number of elements in your collection. For example, if you want to store the characters `'a'`, `'b'`, and `'c'`, then we require $N \geq 3$. Now, we can keep an array indexed from 1 to N (so we have an array of size N), and we can store each element at the position that the function evaluates the element as. So if our hash function is $f(x)$ and $f(a) = 1$, $f(b) = 2$, and $f(c) = 3$, then we would store `'a'` in index 1, and so on. Now, what if we want to determine if an element is present in our data structure? This is also simple: just plug in the value we want to check into our function, and check whether the element is present in the index computed.

What are some desirable properties of hash functions?

- A hash function should be quick to compute. This is true because we want our hash table data structure to support quick retrieval and storing of data. If it takes too much time to compute the index of where a value of interest resides, our hash table itself will run slowly when computing indices of where data should reside.
- A hash function should **collision-free**. This means that it should be difficult to produce two distinct keys that map to the same value. On the extreme end, suppose we had a function that *always* produced 1, no matter what value was provided to it. This means that multiple values would get mapped to the first index in the array, which is not desirable. A function that produces *no collisions* is called a **perfect hash function**.

So, what are some examples of good hash functions? Typically, it's a good idea to create a relatively huge value and use the modulus operator in order to reduce it to the size of our array (the process of reducing this

value to the size of the array is referred to as **scaling**). This works particularly well when our hash table's size is a prime number.

In Java, we can generate a **hash code** (the value produced by the hash function, prior to reducing it to the size of the array with the modulus operator) for a string by using the built-in `.hashCode()` function:

```
System.out.println("Java".hashCode()); // Prints 2301506.
```

How does `"Java"` produce 2301506?. The ASCII value for J, a, and v are $74, 97$, and 118, respectively. The hashCode is then computed by $74 \cdot (31)^3 + 97 \cdot (31)^2 + 118 \cdot 31 + 97 = 2301506$.

This is an example of how hash codes can be computed on strings. For primitive types, like floats or doubles, hash functions typically manipulate the internal binary representation to produce a hash code. For practical purposes, it's usually not plausible to produce a perfect hash function that produces distinct values for every key. In pratice, it's good to pick something that seems fairly random and verify that it works.

19 Wednesday, October 16, 2019

Last time, we introduced hash tables in which certain keys are mapped to indices where their corresponding values reside with the use of a function. What happens if our hash table runs out of space? In order to resolve this issue, we can create a new hash table with greater size and rehash all of our values.

Resolving Collisions

As we have mentioned, it is desirable to have a hash function that prevents collisions. But what happens if a collision *does* occur? We can't store two values in the same index in the array, and our data structure would be unreliable if we just replaced the old value (or didn't store the new value). There are two primary ways in which we can resolve collisions known as **open addressing** and **separate chaining**. In open addressing, we look for an unused entry in the table. In separate chaining, each element in the table becomes associated with more than one search key (i.e. each element becomes a bucket or a list).

Open Addressing

Open addressing can further be subdivided into several variations. A common theme between these variations is that they all use **probing**, which is the process for locating an open element or position in the hash table. Suppose our hash table has size n.

One type of open addressing is known as **linear probing**. In this variation, when a collision occurs at some index position k, we look to see whether position $k+1 \pmod n$ is available (not in use) If it is in use, we look at $k+2 \pmod n$, and so on. Suppose we want to keep a hash table of strings, and our hash function evaluates the following:

- `hash("apple") = 5`
- `hash("watermelon") = 3`
- `hash("kiwi") = 0`
- `hash("mango") = 6`
- `hash("banana") = 2`
- `hash("orange") = 3`
- `hash("pear") = 3`

Under linear probing, we would end up with the following hash table:

How do we get this hash table?

- First, we insert `apple` into index 5 (our indices start from 0, where index 0 is the top-most entry), `watermelon` into index 3, `kiwi` into index 0, `mango` into index 6, and `banana` into index 2.
- Next, we encounter a collision between `orange` and `watermelon`. But using linear probing, we can resolve this collision by noting that index 4 is empty. Thus, `orange` is moved into index 4.
- Next, we encounter a collision between `watermelon` and `pear`. We find the next available index, which is index 7, so we place `pear` into index 7.

kiwi
banana
watermelon
orange
apple
mango
pear

Figure 6: Linear Probing Example

As you may have noticed, resolving collisions with linear probing generates groups of consecutive elements in the hash table. Each group is called a **cluster**, and this phenomenon is known as **primary clustering**. Bigger clusters means longer search times since for each cluster is a sequence of probes that we must search when adding, removing, or retrieving.

Another type of open addressing is known as **quadratic probing**. In this case, we consider elements at indices $k + j^2 \pmod{n}$. For example, we'd first look at index $k + 1 \pmod{n}$ followed by $k + 4 \pmod{n}$, $k + 9 \pmod{n}$, and so on. Unlike linear probing, quadratic probing does not result in primary clustering. Instead, it results in **secondary clustering**, which is a phenomenon in which filled slots are mapped to values that are far away from their keys.

Finally, one last variant is **double hashing** in which the increment of 1 for linear probing and j^2 for quadratic probing is replaced with the result of a second, different hash function that determines the increment.

Separate Chaining

Separate chaining is a second approach to resolving collisions. In this case, each element of the table represents more than one value. Each element in the hash table is called a **bucket**, which can be represented internally with a list, sorted list, or a linked list, etc. Buckets should support searching: we should be able to determine the bucket by hashing the search key and look through the list to find the element or determine that it does not exist. Also, buckets should support insertion: we should look for an item and insert it into the found bucket if it is not found. Finally, buckets should support removal: we should be able to look for the item and remove it from the bucket.

It is important that the number of elements in each bucket is very small. Equivalently, it is important that we do not have too many collisions. Otherwise, we might end up having to search a large list, which completely defeats the purpose of using a hash table. For example, consider the extreme case in which *every* value resides in index 1. In this case, we'll need to search a list in the worst case, which makes a hash table no better than just using an array or a Linked List.

Load Factor

We can quantify the cost of collision resolution with a statistic known as the **load factor**. The load factor is denoted by λ and it is defined as follows:

$$\lambda = \frac{\text{number of entries in hash table}}{\text{size of the hash table}}.$$

As λ increases, the number of comparisons needed to resolve collisions also increases. The performance of linear probing degrades the load factor increases. In order to keep reasonable efficiency, we should maintain $\lambda \leq 0.5$ (i.e. the hash table should be less than half full).

Once we have $\lambda > 0.5$, it is typically useful to resize the hash table and compute a new hash index for every key. This process is known as **rehashing**.

Hash Code Contract

In Java, when we're using hash functions, we must satisfy something known as the **hash code contract**. Essentially, this contract states that if `a.equals(b)` evaluates to true, then we must guarantee `a.hashCode() = b.hashCode()`. However, the inverse or converse of this statement do not necessarily have to be true. That is, it is not necessary for `!a.equals(b)` to imply `a.hashCode() != b.hashCode()`, and it is not necessary for `a.hashCode() == b.hashCode()` to imply `a.equals(b) == true`.

Note that these conditions imply that hash functions must be deterministic in the sense that we should always get the same hash whenever we compute the hash code of an object.

HashSets

Java provides us with a a `HashSet` as a part of the `Collections` framework. The `HashSet` creates a collection that internally uses a hash table for storage. Here is an example of how we can use a `HashSet`:

```
package setIncorrect;

import java.util.*;

public class Roster {
   private HashSet<Person> roster = new HashSet<Person>();

   public void addPerson(String name, int id) {
      roster.add(new Person(name, id));
   }

   public boolean findPerson(String name, int id) {
      Person person = new Person(name, id);

      return roster.contains(person);
   }
}
```

Here, we are using a `HashSet` to represent a collection of `Person` objects. The `addPerson` function adds another `Person` object into our `HashSet` with the provided name and ID. Internally, this `Person` object is added into our hash table. The `findPerson` method performs an internal look-up in our hash table, and checks whether a `Person` object with the same name and ID already exists.

Now consider the following driver program:

```
package setIncorrect;

public class Driver {
   public static void main(String[] args) {
```

```
      Roster section0101 = new Roster();

      section0101.addPerson("Mary", 10);
      section0101.addPerson("Peter", 20);
      section0101.addPerson("Jose", 7);

      if (section0101.findPerson("Peter", 20)) {
         System.out.println("Found Peter");
      } else {
         System.out.println("Peter not found");
      }
   }
}
```

In this driver, we add three different `Person` objects into the `HashSet` corresponding to our `Roster`. In other words, we compute a hash code for each of our objects, and we store them into a large array. By default, the hash code corresponding to a user-made object is just the object's memory address — we can easily verify that this satisfies the hash code contract since `a.equals(b)` implies `a` and `b` reside in the same memory location.

Next, in our code, we check whether our `HashSet` contains a `Person` object with name `"Peter"` and Id `20`. In other words, we compute the hash code for this object, and we check whether there is an entry in our hash table at this index. Assuming our hash function satisfies the hash code contract, this expression evaluates to true, so we print `"Found Peter"`.

It is important to emphasize that, ideally, this is more efficient than using a simple array-based implementation. We are able to retrieve and store elements in expected constant time, which is better than having to iterate through the entire array to find a `Person` object.

20 Friday, October 18, 2019

Exam 2 today.

21 Monday, October 21, 2019

Recap on Java's Hash Code Contract

In order to recap what we learned last week, we can review Java's Hash Contract.

Java's Hash Code Contract states that if two objects are equal (using the `.equals()` method), then we must guarantee that their hash codes are also equal. What happens if we don't satisfy the hash code contract? Classes that rely on hashing will fail.

When we write our own classes, we must write classes that satisfy Java's Hash Code Contract. We will run into problems if we don't satisfy the Java Hash Code Contract and use classes that rely on hashing (like `HashSet` or `HashMap`). A possible problem that we might run into is being able to add elements to a set but not being able to find it during a lookup operation.

It is also useful to be able to recognize when valid hash functions are bad. For instance, overriding the `hashCode()` method to make it **always** return 5 is a a valid hash code. Why? Because if `a.equals(b)` is true, then `a.hashCode() = b.hashCode()` is also true (`a.hashCode()` and `b.hashCode()` will always evaluate to -5). But this hash code is not good because we will have many collisions – many elements are being mapped to index 5.

Sets and Maps

A **set** is a collection that cannot contain duplicate elements. Two set instances are considered to be equal if they contain the same elements. Java's Collections library contains three general-purpose set implementations:

- The `HashSet` set is implemented with hashing. We have already seen this in use in our `Roster` example. The elements in the `HashSet` must implement the `hashCode()` method. This type of set provides us with the most efficiency (in terms of time).
- The `LinkedHashSet` is a `HashSet` that supports the ordering of elements. This allows us to retrieve elements in the order in which we inserted them. On the other hand, the `HashSet` does not guarantee any type of ordering.
- The `TreeSet` guarantees that the elements in a set are in sorted order. The elements being stored in a `TreeSet` must be comparable. We can also pass in custom comparators when using a `TreeSet`.

A **map** is an abstract data type that holds (key, value) pairs. We can think of maps as arrays except we aren't constrained to using integers as our indices. Instead, we can use any other type to index other values, like strings to other strings. The map interface provides methods like `put(K key, V value)` to insert an element, and `get(Object key)` to retrieve an element. There is also a `remove(Object key)` and `clear()` function to remove a single element or completely clear the map. The map concrete classes are listed below. Note that the `Map` interface is not a subtype of the `Collection` interface:

- The `HashMap` map is implemented using hashing. Elements must satisfy the `hashCode()` contract. The `HashMap` is the most efficient in terms of time.
- The `LinkedHashMap` supports ordering of elements (we can retrieve elements in order of insertion), but it is slower than the `HashMap`, which does not guarantee any type of ordering.
- The `TreeMap` requires elements to be comparable, and it guarantees that elements can be retrieved in sorted order.

The following example demonstrates the differences between the three types of maps, and it also illustrates some of the primary methods that we use with maps:

```
/*
 * Example that shows how we can use maps.
 */
import java.util.*;
public class ClassesImpMaps {

   public static void main(String[] args) {
      System.out.println("************* HashMap test *************");
      test(new HashMap<>());

      System.out.println("\n\n************* TreeMap test *************");
      test(new TreeMap<>());

      System.out.println("\n\n************* LinkedHashMap test *************");
      test(new LinkedHashMap<>());
   }

   /* Notice the parameter is a Map, which is an interface */
   public static void test(Map<String, Department> map) {

      /* adding <key,value> pairs to the map */
      map.put("Mary", new Department("Electronics", 5000));
      map.put("Peter", new Department("Music", 4500));
      Department shoeDepartment = new Department("Shoe", 6000);
      map.put("Zoe", shoeDepartment);
      map.put("Laura", shoeDepartment);

      /* printing the contents */
      Set<String> nameSet = map.keySet();
      for (String name : nameSet) {
         /* finding the dept that maps to the name */
         Department dept = map.get(name);
         System.out.println(name + " " + dept);
      }

      /* Membership test */
      if (map.containsKey("Mary"))
         System.out.println("Mary found");

      if (!map.containsKey("Laura"))
         System.out.println("Laura not found");

      /* Getting all the values */
      System.out.println("All departments");
      Collection<Department> collection = map.values();
      for (Department dept : collection) {
         System.out.println(dept);
      }

      /* Another alternative */
      System.out.println("*Another alternative*");
      for (Map.Entry<String, Department> elem : map.entrySet()) {
         System.out.println(elem.getKey() + " " + elem.getValue());
      }
```

```
    }
}
```

In the `test` function, we take in a `Map` that maps `String` objects to `Department` objects. Since we only specify that the parameter is a `Map`, it is valid for us to pass in a `HashMap`, `TreeMap`, or even a `LinkedHashMap`. In this example, we are mapping employee names to departments in which they work in. The `test` function adds four entries to this map. Firstly, we add `Mary` and `Peter` who work in the `Electronics` and `Music` departments, respectively. Next, we add `Zoe` and `Laura` both of whom work in the `shoeDepartment`.

Next, we store the map's `.keySet()` in the variable `nameSet`, which is a function returns a `Set` of keys stored in the map. By using an enhanced for-loop, we can iterate over `nameSet`, but we are not guaranteed any type of ordering.

In our "membership test" section, we will find that the Boolean expression `map.containsKey("Mary")` evaluates to true, whereas the Boolean expression `map.containsKey("Laura")` evaluates to false. Finally, we provide two different ways in which we can print the values in the map.

22 Wednesday, October 23, 2019

Last time, we started looking at the classes that implement the maps, and we looked at a word frequency counter that counts the number of times a word appears in an array of strings.

Applications of Maps and Sets

The following example demonstrates a basic application of maps in which we count the number of occurrences of words in an array of strings:

```
public class WordFrequencyCounter {
   public static void main(String[] args) {
      Map<String, Integer> map = new TreeMap<>();

      for (String word : args) {
         if (!map.containsKey(word)) {
            map.put(word, 1); /* First instance seen */
         } else {
            map.put(word, map.get(word) + 1);
         }
      }

      System.out.println("Words Frequency:\n");
      for (String word : map.keySet()) {
         System.out.println(word + "\t" + map.get(word));
      }
   }
}
```

Here, we create a `Map` that maps `String` objects to `Integer` objects. More precisely, we will be mapping words to the number of times they appear in an array. Next, we process every element in the array `args` with an enhanced for-loop. For every word in the provided string array, we check whether we've seen the word before. If so, we increment the integer that it is being mapped to by one. Otherwise, we set its entry to 1 since it has only occurred once at the given time. Note that it is important that we handle these two cases separately since it would be an error to increment the value if it has not yet been declared (the default value is not 0!).

Finally, we print out the frequency of all of the words. Since we are using a `TreeMap`, we are guaranteed that we will print the words in a sorted manner.

Next, here's an implementation of a `Course` object in which we use a maps and sets:

```
package maps;

import java.util.*;

public class Course {
   private Map<Integer, Set<String>> allSectionsMap = new HashMap<>();

   public void addStudent(Integer sectionNumber, String name) {
      Set<String> sectionSet = allSectionsMap.get(sectionNumber);

      if (sectionSet == null) {
         sectionSet = new TreeSet<>();
         allSectionsMap.put(sectionNumber, sectionSet);
```

```
      }
      sectionSet.add(name);
   }

   public boolean removeStudent(String name) {
      for (Integer sectionNum : allSectionsMap.keySet()) {
         Set<String> sectionSet = allSectionsMap.get(sectionNum);

         if (sectionSet.contains(name)) {
            sectionSet.remove(name);
            if (sectionSet.isEmpty()) {
               allSectionsMap.remove(sectionNum);
            }
            return true;
         }
      }

      return false;
   }

   public void printAllStudents() {
      for (Integer sectionNum : allSectionsMap.keySet()) {
         Set<String> sectionSet = allSectionsMap.get(sectionNum);

         for (String name : sectionSet) {
            System.out.println(name);
         }
      }
   }

   public static void main(String[] args) {
      Course course = new Course();

      course.addStudent(201, "Jose");
      course.addStudent(101, "Mary");
      course.addStudent(101, "Kelly");
      course.printAllStudents();
   }
}
```

In this implementation, we use a `Map` to map integers to a `Set` of strings. In the context of this class, we are mapping section numbers (which fully determine a course) to a `Set` of student names, representing the people taking that course.

In the `addStudent` method, we take in a section number as well as the student to add. We retrieve the current set of students by using our `Map`'s `.get()` method. Subsequently, we check to see if the retrieved value is `null`. This is important because, as mentioned in the previous example, there is no default value for what the keys are being mapped to. This means that we can obtain a `NullPointerException` if we don't handle the case in which the `sectionNumber` has no students separately.

If the `sectionSet` retrieved is null, then the class is currently empty. This means that we need to instantiate a set, make the `sectionNumber` map to the `sectionSet`, and finally add the student's name to the `setctionSet`. On the other hand, if the `sectionSet` is not null, then we can just add the student without any trouble.

In the `removeStudent` method, we iterate over all of the section numbers (i.e. all of the keys) by using the `keySet` method of our map. For every key we iterate over, we retrieve the set corresponding to that key, and we check whether that student is in that course. If so, we remove the student from that course. If the course becomes empty after removing that student (that is, the student was the only student in that course), then we remove the course as well.

Finally, the `printAllStudents()` function iterates over all section numbers. For each section number, retrieve the set of students in that course. Finally, we print the names of the students in each course.

Another example in which maps can be useful is by using lists as keys. Consider the following code segment:

```
package maps;

import java.util.HashMap;
import java.util.List;
import java.util.ArrayList;

public class ListsAsKeys {
   public static void main(String[] args) {
      HashMap<List<String>, String> favoriteDessertMap = new HashMap<>();

      ArrayList<String> johnSmith = new ArrayList<>();
      johnSmith.add("John");
      johnSmith.add("Smith");
      favoriteDessertMap.put(johnSmith, "Chocolate");

      ArrayList<String> rosePeterson = new ArrayList<>();
      rosePeterson.add("Rose");
      rosePeterson.add("Peterson");
      favoriteDessertMap.put(rosePeterson, "Ice Cream");

      for (List<String> list : favoriteDessertMap.keySet()) {
         System.out.println(list + " " + favoriteDessertMap.get(list));
      }

      /* Retrieving */
      ArrayList<String> temp = new ArrayList<>();
      temp.add("John");
      temp.add("Smith");
      if (favoriteDessertMap.containsKey(temp)) {
         System.out.println("Favorite Dessert John Smith: " + favoriteDessertMap.get(temp));
      }

      /* Notice what happens when we cleared the entry */
      johnSmith.clear();
      System.out.println("Listing Map Contents After Clearing: ");
      for (List<String> list : favoriteDessertMap.keySet()) {
         System.out.println(list + " " + favoriteDessertMap.get(list));
      }
   }
}
```

In this example, we created a `HashMap` that is mapping a list of people's names (represented as `String` objects) to another `String` object representing their favorite food. There aren't too many new concepts

going on, except that it might seem counterintuitive to use a `List` as our key. Near the bottom of the code segment, after we've cleared `johnSmith()`, we print out `[], null`. The `[]` indicates an empty array, and the `null` indicates that the corresponding value is not present. This emphasizes that it is important not to mess with the keys that are being used.

23 Friday, October 25, 2019

Algorithmic Complexity

Time Functions and Big-$\mathcal{O}$ Notation

While we have already looked at the basics of Big-$\mathcal{O}$ notation, we will now learn how to analyze programs and determine their asymptotic complexity on our own.

The **critical section** of a program is another word for the "heart" of an algorithm. In essence, it indicates the portion of code where the most "work" is performed in terms of time needed. The critical section of an algorithm dominates its overall execution time, and its operation is central to the functioning of a program. Typically, the sources of a critical section comes from loops or recursion.

Our goal is to find the asymptotic complexity of various algorithms. The general approach for doing so is ignoring frequently executed parts of the algorithm, finding the critical of the algorithm, and determining how many times the critical section is executed as a function of the problem size.

Here's an example of some code in which we can easily identify the critical section:

```
A
for (int i = 0; i < n; i++) {
    B /* This is the critical section. */
}
C
```

Suppose A, B, and C are sequences of constant-time operations (for example, print statements). Then, A is executed exactly once, B is executed n times (it is inside of a loop that loops through n times), and C is executed exactly once. Therefore, the total number of operations we perform is $T(n) = 1 + n + 1 = n + 2$. The high-order term of $T(n)$ is n, which implies that this algorithm runs in $\mathcal{O}(n)$ time. The function $T(n)$, which gives the exact number of operations, is referred to as our **time function**. Using this terminology, our asymptotic complexity is the high-order term of our time function.

It is important to be able to compute the time function exactly. Thus, it is important to be careful, particularly with the bounds of our loops. In the previous example, it is clear that the loop iterates through exactly n times (once for $i = 0$, once for $i = 1$, all the way up to $i = n - 1$). If we instead wrote our loop as `for (int i = 1; i <= n; i++)`, then our loop would still iterate through n times. But `for (int i = 0; i <= n; i++)` iterates through $n + 1$ times. Another example of a for-loop is given by `for (int i = 0; i < n; i += n)`, which executes only once.

Here's another example:

```
A
for (int i = 0; i < n; i++) {
    B
    for (int j = 0; j < n; j++) {
        C
    }
}
D
```

Once again, suppose A, B, C, D are sequences of constant-time operations. In this scenario, A is executed exactly once. B is executed exactly n times (it is in a for-loop that iterates through n times), C is executed

n^2 times (it is inside a for-loop that executes another for-loop that iterates n times exactly n times, so we have $n \cdot n = n^2$ executions). Finally, D is executed once. Therefore, our time function is given by

$$T(n) = 1 + n + n^2 + 1 = n^2 + n + 2.$$

The high-order term of $T(n)$ is n^2, so this algorithm runs in $\mathcal{O}(n^2)$ time.

Here is a third example:

```
A
for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
        B /* Critical section. */
    }
}
```

Here, A is executed exactly once. But now, the inner for-loop depends on the value of the outer for-loop. How do we determine the number of times B is executed? We can note that when $i = 0$, the inner for-loop starts at $j = 1$, and it goes up to n. This contributes a total of $(n - 1)$ executions. Next, when $i = 1$, the inner for-loop starts at $j = 2$, and goes up to n. This contributes a total of $(n - 2)$ executions. This process continues up until the inner for-loop contributes 0 executions. Therefore, our time function is given by

$$\begin{aligned} T(n) &= \underbrace{1}_{\text{Executions of Statement } A} + \underbrace{((n-1) + (n-2) + \cdots + 1 + 0)}_{\text{Executions of Statement } B} \\ &= 1 + \frac{n(n-1)}{2} \\ &= \frac{n^2}{2} - \frac{n}{2} + 1 \end{aligned}$$

where we used the identity $\sum_{k=1}^{n} k = n(n+1)/2$ to simplify the number of executions of Statement B. Since the high-order term of of $T(n)$ is n^2, we conclude our algorithm is $\mathcal{O}(n^2)$.

It is important to note that minor changes to algorithms won't affect the asymptotic complexity of the algorithm. For example, if we instead looped from $i = 0$ and $j = i + 1$ up to $n/2$ instead of n, our algorithms would still be $\mathcal{O}(n^2)$ (although, our time functions would decrease). Essentially, Big-$\mathcal{O}$ notation is mostly concerned with *long-run* behavior, whereas time functions are concerned with exact behavior.

Here is another example:

```
int i = 1;
while (i < n) {
    /* This is the critical section. */
    A
    i = 2 * i;
}
B;
```

In this case, the statement $i = 1$ contributes 1 to our time function (the time function accounts for *every* operation). Now, how many times does A execute? The answer is $\log_2(n)$. This can be seen easily by plugging in different powers of 2 for n, and tracing the code. Logarithmic time complexity typically indicates that we are increasing the loop variable by some constant factor, or we're reducing the problem at-hand by a constant factor. Thus, statement A contributes a $\log_2(n)$ term to our time function, and the re-assignment of i inside

of the for-loop contributes another $\log_2(n)$ term. Finally, the statement B executes one more operation, from which we obtain

$$T(n) = 1 + \log_2(n) + \log_2(n) + 1 = 2(\log_2(n) + 1).$$

The high-order term here is $\log(n)$ so we conclude that this algorithm is $\mathcal{O}(\log(n))$.

Complexity of Recursive Algorithms

How can we compute the complexity of a recursive algorithm? We can do this by writing the recurrence in terms of itself. For example, consider the following program:

```
int fact(n) {
    if (n == 0) return 1;
    return n * fact(n - 1);
}
```

If n is greater than 0, then we perform exactly 3 operations and perform a recursive call on $n-1$. On the other hand, if n is equal to 0, then we perform only one operation. In this case, our time function can be expressed by $T(n) = T(n-1) + 3$ with base case $T(1) = 0$. This type of equation is known as a **recurrence**. There are many ways to solve recurrences, but knowing how to solve them is not necessary in this class.

By performing iteration, we can solve the recurrence in a very informal way by identifying a pattern:

$$\begin{aligned} T(n) &= T(n-1) + 3 && (1)\\ &= T(n-2) + 6 && (2)\\ &= T(n-3) + 9 && (3)\\ &= \dots && (4)\\ &= T(n-k) + 3k. && (5) \end{aligned}$$

We keep on iterating until $k = n$ in which case we obtain $T(n) = T(0) + 3n$. But since we have the base case $T(0) = 1$, we find $T(n) = 3n + 1$. Therefore, we can conclude that the algorithm runs in$\mathcal{O}(n)$ time.

Here's another example of an algorithm (pseudocode is shown):

```
MergeSort(Array) {
    if (n == 1) return;
    MergeSort(first half of array)
    MergeSort(second half of array)
    Merge the two halves. /* Requires n operations. */
}
```

In this algorithm, we perform n operations per recursive call. In addition, we make two calls to the algorithm with subproblem size $n/2$, so we obtain $T(n) = 2T(n/2) + n$ as our recurrence equation. The base case occurs when $n = 1$ in which case we perform just one operation. Thus, our base case is given by $T(1) = 1$. This recurrence can be a little bit more tricky to solve, but the solution's high order term is $n \log_2(n)$. Thus, this algorithm runs in $\mathcal{O}(n \log(n))$ time.

Additional Complexity Measures

So far, we have only discussed Big-$\mathcal{O}$ notation, which is an upper bound on the number of operations we perform. However, we can also use Big-Ω notation, which is a *lower* bound on the number of operations we perform (e.g. what is the least number of operations we perform?).

Suppose we want to find the maximum element in an array with n numbers. One easy way to do this is to keep track of the maximum value seen so far, and update this value when we see a new maximum. This algorithm runs in $\mathcal{O}(n)$ time. But, it can also be shown that a lower bound for this algorithm is $\Omega(n)$ time. This is true because there cannot exist an algorithm that finds the maximum of n numbers without at least looking at what the n numbers are. But even the act of just looking at the n numbers requires linear time.

When the Big$-\mathcal{O}$ and Big-Ω of an algorithm are equal (i.e. $\Omega(f(n)) = \mathcal{O}(f(n))$ for some function $f(n)$), then we say that the algorithm is $\Theta(f(n))$. This is a stronger result.

Trees

A **tree** is a hierarchical, non-linear data structure that can be used to represent a one-to-many relationship between elements. The data structure consists of a **root node** which is typically the topmost node in a data structure as well as many other **internal nodes**. Each node stores values and references to other nodes in the tree. Below is an example of a tree:

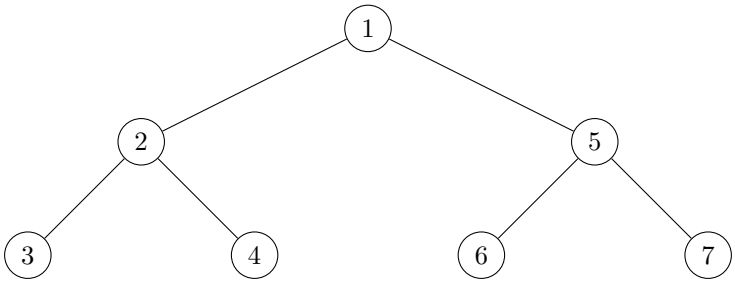


Figure 7: An Example of a Tree

Here, the node labelled 1 is the root node, and every other node is an internal node.

What are trees useful for? Trees are good for clearly depicting relationships. For example, we can model ancestry trees, or relationships between people. We can also model a network of cities with vertices representing cities, and an edge between two cities representing a roadway or path between the two cities.

In the diagram above, we see that every node other than $3, 4, 6$ and 7 have two nodes coming out, underneath them. We call these nodes the **child nodes** of the node that they are coming out of (so nodes 3 and 4 are the children of node 2, and node 7 has no children). Also, we can differentiate between the two children by using the terminology **left child** and **right child**.

The **level** of a node is a measure of the node's distance from the root. More precisely, if the node is the root of the tree, then its level is 1. Otherwise, its level is its parent's level plus one. The **height** of a tree is the maximum level of any node in the tree.

A **binary tree** is a special kind of tree in which each node has at most 2 children. We will place a special focus on this type of tree, starting next lecture.

24 Monday, October 28, 2019

Last time, we introduced **trees**, which are non-linear data structures that can be used to represent relationships between elements. A **binary tree** is a special type of tree in which each node has at most two children.

Tree Traversal

Unlike like linear data structures, like arrays, we cannot just use a for-loop to iterate over a tree since the data structure is non-linear. Thus, we are motivated to find a new way to traverse the data stored in trees. There are two primary ways in which we can perform tree traversal.

Breadth-First Search

Breadth-first traversal starts from a source node, and it visits all nodes that are closest to that source node prior to visiting any nodes that are farther out. In order to illustrate breadth-first traversal, consider the following tree:

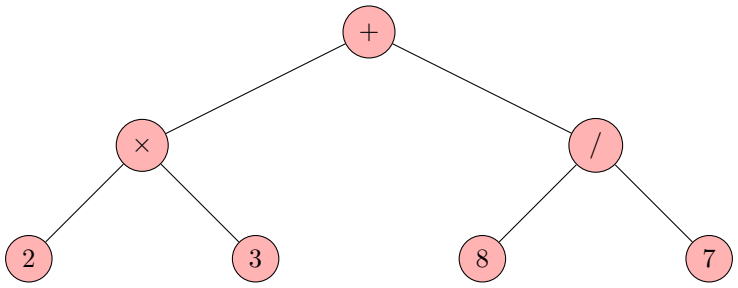


Figure 8: An Expression Tree

Now suppose we begin a breadth-first search traversal starting at the root (the node with the plus sign). In this case, we first visit the root, and we subsequently visit either the node with the multiplication sign or the node with the division sign. The actual order does not matter; the only thing that matters is that the next node visited is the smallest distance away from the root node. Suppose we visit the node with the multiplication sign next. Afterwards, we'll visit the node with the division sign (there is no other node whose distance is only one away from the root, so this is the only candidate). Finally, we'll visit the the nodes whose distances are 2 away from the root (i.e. the nodes $2, 3, 8, 7$ in any order). The key thing to remember when performing a breadth-first search traversal is that the nodes are visited in ascending order of their distance from the root node.

What are some applications of breadth-first search traversal? One simple example is shown through social networking sites. We can represent all social media profiles with nodes, and we can place an edge between two nodes if the two people represented by those nodes are friends. By performing a breadth-first search, we can figure out how many friends away a given person is from yourself. This can be useful when suggesting friends for you to add.

Depth-First Search

A **depth-first search traversal** can be further separated into three distinct categories:

- In a **pre-order traversal**, we visit the node that we are currently at first. Subsequently, we recursively visit the left subtree, and we finally recursively visit the right subtree.

- In an **in-order traversal**, we recursively visit the left subtree followed by the node we are currently at, and we finally visit the right subtree.
- In a **post-order traversal**, we recursively visit the left subtree followed by a recursive visit to the right subtree. Finally, we visit the node we are currently at last.

In order to keep track of the three different types of depth-first search traversal, note that the left subtree is *always* visited before the right subtree. Also, the order in which we visit node we are currently at can be determined from the name of the traversal (in a "pre-order traversal," we visit the node we are currently at first, whereas in a "post-order traversal", we visit the node we are currently at last).

Once again, consider the same tree from the breadth-first search example:

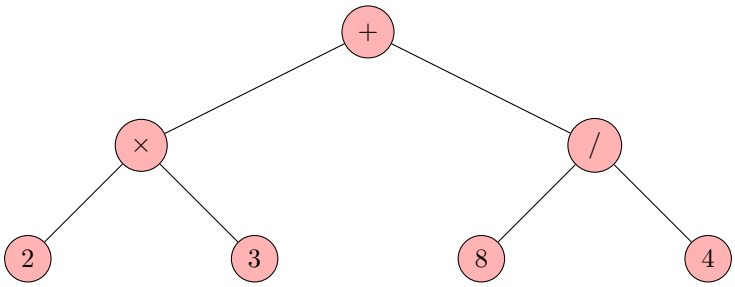


Figure 9: An Expression Tree

Let's trace what happens under the three different types of depth-first search traversal:

- In a pre-order traversal, we visit the node we are currently at first, and we subsequently process the left subtree followed by the right subtree. Since we are starting the search at the root, the node with the plus sign is processed first. Subsequently, we make a recursive call on the left subtree. Now to process the node with the multiplication sign, we perform the exact same procedure. We process the node we are currently at first (which is the node with the multiplication sign), and we make a call to the left and right subtrees. This processes the nodes 2 and 3 in that order. Finally, there are no more nodes left in the left subtree of the plus symbol, so now we perform a recursive call on the right subtree of the plus node. Similar to the left subtree, we end up processing the division node followed by the 8 node, and finally the 4 node. Thus, the final order in which nodes are processed in a pre-order traversal is given by $+, \times, 2, 3, /, 8, 4$.
- In an in-order traversal, we recursively visit the left subtree, then the node we at, and finally the right subtree. In the tree above, we make recursive left calls until we hit the 2 leaf node. Next, we process the $\times$ node as well as the 3 node. At this point, we have finished processing the $+$ node's left subtree call. Next, we process the $+$ node, and we finally process 8 node followed by the $/$ node and 4 node. Thus, the final order in which the nodes are processed is given by $2, \times, 3, +, 8, /, 4$.
- In a post-order traversal, we process the left subtree, the right subtree, and finally the node we are currently at. In this case, note that the root is always processed last. First, we make a recursive call to the left subtree rooted at $\times$. When processing this subtree, we make another recursive call to the left, and we end up processing 2 first. Next, we perform the $\times$ node's right subtree call, which results in processing the node 3. Next, we process the $\times$ node. Finally, we procses the right subtree of $+$ in a similar manner, and we finally process the root node. The final order in which nodes are processed is given by $2, 8, \times, 8, 4, /, +$.

Binary Tree Terminology

Here, we introduce some more terminology that is used to describe binary trees.

The **level** of a node is a measure of the node's distance from the root. Observe that there is a natural recursive definition for the level of a node. If the node is the root of the tree, then its level is 1. Otherwise, the node's level is one plus the level of its parent's level.

The **height** of a tree is the maximum level of any node in the tree.

A tree is called **degenerate** provided that most of the nodes only have one child. Here is an example of a degenerate tree:

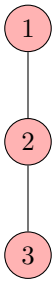


Figure 10: A Degenerate Tree

A degenerate tree is similar to a Linked List in the sense that it is "approximately linear" (or, like in the case above, it can be linear). Typically, a binary tree being degenerate is an undesirable property to have. The height grows approximately linearly since usually adding a new node increases the height by one. The performance of the tree typically degrades when a tree is degenerate.

A tree is called **balanced** provided that most of its nodes have two children. Here is an example of a tree that is balanced:

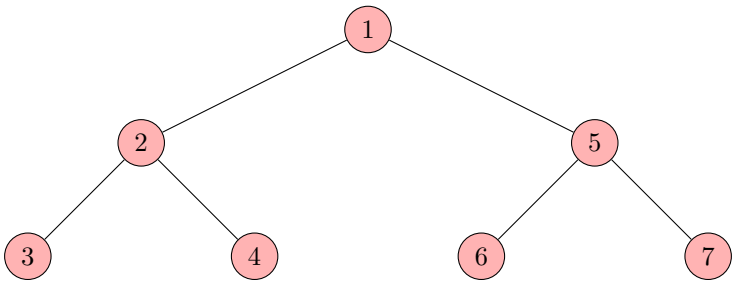


Figure 11: A Balanced Binary Tree

Typically, a binary tree being balanced is a desirable property to have. the height grows logarithmically since usually we need to add approximately twice as many nodes as before to increase the height by one.

A **full binary tree** is a tree in which every node other than the leaves have two children. The tree above is also a full binary tree, and every full binary tree is balanced. In fact, we can find an explicit formula for the number of nodes in a full binary tree: $n = 2^h - 1$, where h is the height of the tree. For example, in the image above, the height of the tree is 3 (assuming height starts at 1). Thus, our formula tells us that the tree has $2^3 - 1 = 7$ nodes, which agrees with our picture above.

Binary Search Trees

A **binary search tree** is a data structures that is organized, as its name suggests, in a binary tree. However, they differ in one primary way: the values stored in a binary search tree are always stored in such a way as to satisfy the **binary-search-tree property**:

- For any node n, all nodes in the left subtree of n are less than or equal to n.
- For any node n, all nodes in the right subtree of n are greater than or equal to n.

Here's an example of a binary search tree:

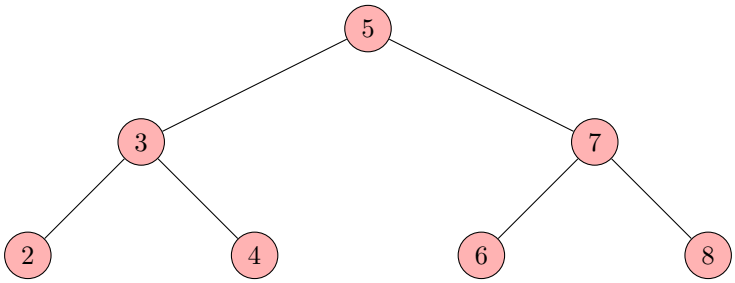


Figure 12: A Binary Search Tree

Note that the right subtree the root node contains the values $6, 7$, and 8. All three of these values are greater than the root node. On the other hand, all values in the left subtree of the root node are less than the root node. Note that this property does not only apply to the root node but rather every node in the tree. If we choose some arbitrary node in the tree and look at the nodes to the left of it, we should only find smaller values. Likewise, if we choose some arbitrary node in the tree, and we look at nodes to the right of it, we should only find larger values.

Due to the binary search tree property, performing an in-order traversal on a binary search tree processes the nodes in sorted order. For example, performing an in-order traversal on the tree above processes the nodes in the order $2, 3, 4, 5, 6, 7, 8$.

How do we search for an element in a binary search tree? We can find an element in a binary search tree by comparing the target value we are searching for with the current node we are at. At each step, this tells us whether our target value should lie in the left subtree or the right subtree (if it exists). For example, suppose we want to look for 4 in the tree provided above. We start at the root, and we make a comparison with 4 and 5. Since $4 < 5$, we know that if 4 exists, it must lie in the left subtree of 5. We make a recursive call to the left subtree, and we compare 4 with 3. Since $3 < 4$, we recurse on the right subtree of 3, which happens to be 4. Note that this is very similar to performing a binary search. We expect the runtime of searching in a binary search tree to be $\mathcal{O}(\log(n))$; however, in the worst case (when the tree is degenerate), the search algorithm runs in $\mathcal{O}(n)$ time.

A helpful tool to visualize binary search trees is `http://btv.melezinek.cz/binary-search-tree.html`

25 Monday, October 30, 2019

Insertion in Binary Search Trees

Last class, we described how we can search for an element in a binary search tree: making comparisons at each step tells us whether we should make a recursive subcall on the left or right subtree. Following a similar procedure, we can easily insert values into binary search trees. Suppose we have the following binary search tree:

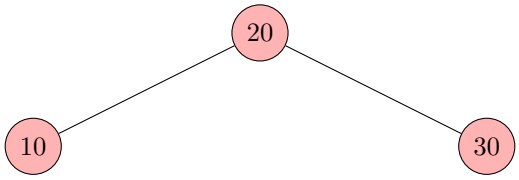


Figure 13: A Binary Search Tree

Suppose we want to insert the value 35. We start off by making a comparison with the root node (20), and we determine that the value 35 belongs in the right subtree of 20. Next, we make a comparison with 30, and we determine that the value 35 belongs in the right subtree of 30. But since 30 has no right subtree, we stop recursing, and we add the value 35 where it belongs. A newly inserted value will **always** end up as a leaf node in the new tree. Similar to searching, we note that insertion requires $\mathcal{O}(\log(n))$ time in the average case, but it takes $\mathcal{O}(n)$ time in the worst case (when the tree is degenerate).

What's an example in which the worst case behavior is exhibited? Suppose we insert a sequence of values in ascending or descending order. This will surely result in a degenerate tree, and our tree's insertion method will operate similar to a Linked List's insertion method. This is bad since we will end up performing in $\mathcal{O}(n)$ time rather than $\mathcal{O}(\log(n))$ time.

Deletion in Binary Search Trees

So far, insertion and searching in binary search trees has just involved starting at the root node, and recursing on the left or right subtree, depending on what a comparison yields. However, deleting a node in a binary search tree can be a little bit trickier.

The general outline for deleting a node X in a binary search tree is described below:

1. Perform a search for value X in the binary search tree.
2. If X is a leaf node, delete X.
3. Otherwise, X must be an internal node. In this case, replace X with the largest value Y on the left subtree or the smallest value Z on the right subtree. Now recursively call this delete method on the replacement value (either Y or Z) from the subtree.

The reason why deletion in a binary search tree might seem complicated is that we must preserve the binary search tree property when we are removing a node from the tree. The method described above ensures that we do not violate the binary search tree property even after a node has been removed. Our base case here is deleting a leaf node since our binary search tree property can never be violated if we just remove a leaf.

Binary Search Tree Implementation

So far, we've mostly discussed the theory behind trees and binary search trees. But how do we actually implement a binary search tree in Java? In order to do this, we need to create a class that represents a Binary Search Tree, and this class needs to have an inner class representing a node inside of a binary search tree (this is very similar to how we implemented a Linked List).

Here's one way in which we can begin the implementation of a generic binary search tree:

```
public class BinarySearchTree <K extends Comparable<K>, V> {
    private class Node {
        private K key;
        private V data;
        private Node left, right;
    }

    public Node(K key, V data) {
        this.key = key;
        this.data = data;
    }
}
```

Note that our we require the generic type `K` to extend `Comparable` since our binary search tree property imposes an ordering on its elements. Each node has a reference to its left and right subtree. The `key` entry in the `Node` class represents the quantity that we are using to determine the node's relative order to the other nodes in the tree. On the other hand, the `data` field represents the actual data that the node is storing (in some cases, these two quantities might be the same).

Here is an implementation of the `add` method which incorporates the ideas that we have already discussed:

```
  public boolean add(K key, V data) {
     if (root == null) {
        root = new Node(key, data);
        return true;
     } else {
        return addAux(key, data, root);
     }
  }

  private boolean addAux(K key, V data, Node rootAux) {
     int comparison = key.compareTo(rootAux.key);

     if (comparison == 0) {
        rootAux.data = data;
        return false;
     } else if (comparison < 0) {
        if (rootAux.left == null) {
           rootAux.left = new Node(key, data);
           return true;
        } else {
           return addAux(key, data, rootAux.left);
        }
     } else {
        if (rootAux.right == null) {
           rootAux.right = new Node(key, data);
           return true;
        } else {
```

```
            return addAux(key, data, rootAux.right);
        }
    }
}
```

Firstly, in the `add` function, we check whether `root` is null. If it is, our tree is currently empty, which means that we just need to make the node that we are adding equal to the root. On the other hand, if we enter the `else { ... }` clause, our tree is non-empty, and we must make use of our recursive auxillary function. In the auxillary function, we compare the value that we wish to add with the node that we are currently at. If the two values are equal, we just update the data (and do not add a new node). On the other hand, if the value we are adding is less than the current node we are at, we either add the node as the left child of the current node (if there is no left child), or we recurse on the left subtree. The case in which the comparison yields that the value we are adding has a key larger than the current node's key is similar, except we recurse and add onto the right subtree.

Next, we'll discuss an implementation of a `toString()` method, which prints the values in a binary search tree:

```
    public String toString() {
    return toStringAux(root);
}

private String toStringAux(Node rootAux) {
    return rootAux == null ? ""
          : toStringAux(rootAux.left) + "{" + rootAux.key + ":" + rootAux.data + "}" +
              toStringAux(rootAux.right);
}
```

This method is fairly short. We use a recursive helper function `toStringAux()` to perform an in-order traversal on our tree. We check whether the current root we are at is null (in this case, there is nothing to print). If it isn't null, we print the data associated with the left subtree followed by the data associated with the current node, and we finally print all of the data in the right subtree. As we mentioned before, an in-order traversal prints the values in a binary search tree in ascending order of the node's keys.

Finally, here is an implementation of a `find` method, which returns true provided that an inputted node exists in the binary search tree:

```
    public boolean find(K key) {
    return find(key, root);
}

public boolean find(K key, Node rootAux) {
    if (rootAux == null) {
        return false;
    } else {
        int comparison = key.compareTo(rootAux.key);
        if (comparison == 0) {
            return true;
        } else if (comparison < 0) {
            return find(key, rootAux.left);
        } else {
            return find(key, rootAux.right);
        }
    }
```

This implementation is very similar to the insertion method. We make comparisons at each step, which allows us to easily target where the value should be present in the binary search tree (if it exists at all).

26 Monday, November 1, 2019

Binary Search Tree Implementation

Recall that the **size** of a tree is the total number of nodes in the tree. We can define a recursive function to determine the size of a tree as follows:

```
public int size() {
   return sizeAux(root);
}

public int sizeAux(Node rootAux) {
   return rootAux == null ? 0 : 1 + sizeAux(rootAux.left) + sizeAux(rootAux.right);
}
```

This recursive function adds 1 if the current node we're at isn't null, and it recurses on the left and right subtrees. Each node contributes 1 to the size, which is exactly what we want.

Finally, here's a recursive function that computes the height of a tree (the maximum-lengthed path from the root to a leaf node):

```
 /* Note: The height of an empty tree is not defined. */
/* Source: https://xlinux.nist.gov/dads/HTML/height.html */
public int height() {
   if (root != null) {
      return heightAux(root);
   }
   return -1;
}

public int heightAux(Node rootAux) {
   if (rootAux.left == null && rootAux.right == null) {
      return 0;
   } else if (rootAux.left != null && rootAux.right == null) {
      return 1 + heightAux(rootAux.left);
   } else if (rootAux.left == null && rootAux.right != null) {
      return 1 + heightAux(rootAux.right);
   } else {
      return 1 + Math.max(heightAux(rootAux.left), heightAux(rootAux.right));
   }
}
```

27 Wednesday, November 6, 2019

Binary Heaps

A **heap** is a data structure represented by an underlying array A that can be interpreted as an "almost complete" binary tree. Each node in the tree corresponds to a single element in the array. The tree is "almost complete" in the sense that the tree is completely filled on all levels except possibly the lowest, which is filled from the left to right up to a point.

How do we represent our almost complete binary tree with an array? We can start our array at index 1 (using a sentinel at index 0), and we can store the root of the node in `A[1]`. Furthermore, given the index i of a node, we can place the indices of its parent, left child, and right child in the indices $\lfloor i/2 \rfloor$, $2i$, and $2i+1$, respectively. As an example, consider the following array:

32	24	22	23	7	2

The corresponding binary heap is presented below:

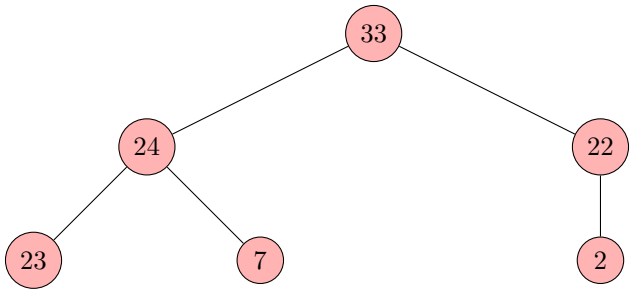


Figure 14: A Binary Heap

For instance, observe that the parent of the nodes storing 24 and 22 is the node storing 33. Furthermore, note that the the elements 24 and 22 are in the second and third indices of the array. By noting that $\lfloor 2/2 \rfloor = \lfloor 3/2 \rfloor = 1$, we see that the parent of these two nodes should be at index 1, which agrees with our array representation above.

There are two types of binary heaps: **max-heaps** and **min-heaps**. In both cases, the nodes satisfy a **heap property**, which differs depending on the type of heap. In a max heap, the **max-heap property** asserts that for every node v in the tree, we require `A[PARENT(i)] >= A[i]`. On the other hand, in a min-heap property, we require `A[PARENT(i)] <= A[i]`. In other words, the parent of a node in a max heap is always greater than or equal to the node itself, and the parent of a node in a min heap is always less than or equal to the node itself. Thus, we can conclude that the figure above is a max heap.

Note that the max heap depicted above does not have its entire last level full. This is okay — as we mentioned previously, the tree must be completely filled on all levels except possibly the lowest, which should be filled from the left to the right up to a point.

Here is an example of a tree that is **not** a heap:

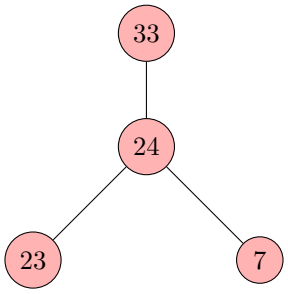


Figure 15: Not a Binary Heap

Although the tree depicted above satisfies the max heap property (each parent is greater than its children), the level directly after the root node (with value 33) is not completely filled, so it is invalid to begin the level after that. Thus, this is not a binary heap.

Due to the "almost completeness" of binary heaps, we are ensured that the binary tree represented by the heap is balanced. In other words, the height of the binary tree grows logarithmically in the number of nodes (it's impossible for heaps to exhibit degeneracy).

We will now discuss how to insert and extract elements from min heaps. Although our descriptions will specify how to insert and extract elements from a min heap, they can easily be modified to insert and extract elements from a max heap.

Insertion

When inserting an element into a min heap, we need to be careful not to mess up the min heap property. This can be done with the following insertion procedure:

1. Insert the new element to the end of the array (so it will be the bottom right-most element in the tree).
2. While the parent's key is larger than the inserted element's key, swap the parent node with the inserted element's node.
3. Insertion is complete.

For example, suppose we have the following min heap:

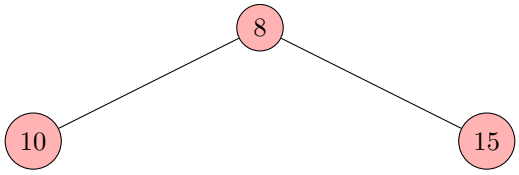


Figure 16: Min Heap

If we wanted to add the node with key 5 into the heap, we would first append the value to the last slot in the array (so 5 would be a child of the node 10). Subsequently, we would compare 5 with 10. Since $5 < 10$, we would swap the node 5 with its parent node, and we would repeat the procedure so that 5 is the root node, its immediate left child is 8, whose immediate right child would be 10.

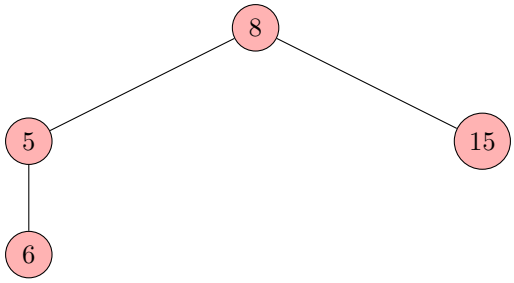


Figure 17: Min Heap

Extraction

What about deletion? When working with heaps, we do not extract any node other than the root. This allows us to get the smallest element (in a min heap) very quickly. But, how do we maintain the min heap property? This can be done with the following `extractSmallest()` procedure:

1. Replace the root with the smallest node in the tree (but store the value of the old root — this is the value that we are extracting). We will henceforth refer to this new root as node v.
2. Compare v to both children. If v is larger than either child (i.e. the min heap property is violated), then swap the root with the smallest child. Repeat the swapping procedure on the subtree rooted at v until v is smaller than both of its children. At this point, the min heap property will be satisfied.

28 Monday, November 11, 2019

Introduction to Multithreading

A **thread** is an execution path in a process, almost like a program inside of another program. Our environment has the ability to rapidly switch between threads giving the illusion that multiple tasks are being performed at once.

Why would we want to create multiple threads? One reason is efficiency. Suppose there's one task that needs to be performed, but this task is dependent on another task's completion. Instead of having the computer hosting the second task do nothing but wait for the first task, we can utilize the thread that would be used to complete the second task to help complete the first task (or some other task).

As a motivating example, suppose we've designed a clock that works in our own timezone, but now we want to design four clocks, each of which represent a different timezone. Instead of copy-and-pasting the same code over and over again, we can instead use our one working clock and spawn four threads, each of which represent a different timezone. The program will allow each thread to run for the correct amount of time. Even if you only have one CPU, it will look like the clocks are moving at the same time since the environment rapidly switches between the threads.

The minimalistic representation of a thread consists of the **stack** and a **program counter**. The stack allows each thread to store its own variables and call functions, while the program counter points to the next instruction to execute.

Just about everything else is shared by threads. For example, threads share files. If one thread opens a file, that same file can be utilized by any other thread in our process. Moreover, threads share the heap. If one thread creates an object in the heap, another thread can use the same object.

Using Threads in Java

There are two primary ways in which we can use threads in Java. The first way defines a class that extends the `Thread` class. Once we extend this class, we must subsequently override the `run` method in which we define what the thread does. The thread

Here's an example of a thread whose task is to just print a message a variable number of times:

```
public class MessageThread extends Thread {
   private String msg;
   private int times;

   public MessageThread(String msg, int times) {
      this.msg = msg;
      this.times = times;
   }

   /* We could have called print() in run */
   public void run() {
      for (int i = 0; i < times; i++) {
         System.out.println(msg);
      }
   }
}
```

Now to actually make a thread start doing its role, we must declare the object that extends the `Thread` class, and we must call the `.start()` method on the object. Here is a driver program that illustrates how we can use the `MessageThread` class:

```
public class Driver {
   public static void main(String[] args) {
      int times = 1000;

      MessageThread msg1 = new MessageThread("Hello", times);
      MessageThread msg2 = new MessageThread("Bienvenidos", times);

      msg1.start();
      msg2.start();

      System.out.println("Driver done");
   }
}
```

In the code above, we create two `MessageThread` objects, namely, `msg1` and `msg2`. We then run the `.start()` method in order to signal that we want the thread to start performing their task (this is important — we do **not** directly call `.run()`; we call `.start()`, and the `run` method will get executed for us). It's important to note that just calling `.start()` doesn't necessarily start the thread's execution sequence either; it simply tells the compiler that we want the thread to do its job as soon as possible. What gets printed out in this example? During execution, the environment will rapidly switch back and forth between which thread gets execution time. There is no one deterministic output that gets printed out every time. With high probability, we can expect the `Hello` and `Bienvenidos` messages to get interweaved with each other.

While the previous technique to define threads (by extending the `Thread` class) is fairly simple, it does not scale well particularly because Java classes can only extend one parent class. This can lead to complications if we're working with more intricate objects that have a large hierarchy. As an alternative, we can make a class implement the `Runnable` interface and implement the `run` method. Here's the same `MessageThread` class, except now it implements the `Runnable` interface:

```
public class DisplayMessageTask implements Runnable {
   private String msg;
   private int times;

   public DisplayMessageTask(String msg, int times) {
      this.msg = msg;
      this.times = times;
   }

   public void run() {
      for (int i = 0; i < times; i++) {
         System.out.println(msg);
      }
   }
}
```

Note that a class that implements `Runnable` is *required* to implement the `run` method (since `Runnable` is an interface). Now another key difference is that, to start threads, we no longer call the `start` method. Instead, we create a `Thread` object, and we pass in the object implementing the `Runnable` interface as a parameter. Here's an example:

```
public class Driver {
   public static void main(String[] args) {
      int times = 1000;

      DisplayMessageTask msg1Task = new DisplayMessageTask("Hello", times);
```

```
      Thread msg1Tread = new Thread(msg1Task);

      DisplayMessageTask msg2Task = new DisplayMessageTask("Bienvenidos", times);
      Thread msg2Thread = new Thread(msg2Task);

      /* Using lambda */
      Thread msg3Thread = new Thread(() -> {
         for (int i = 1; i <= 100; i++)
            System.out.println("Hola");
      });

      msg1Tread.start();
      msg2Thread.start();
      msg3Thread.start();

      System.out.println("Driver done");
   }
}
```

Unlike before, once we create a `DisplayMessageTask` object, we cannot just call `.start()` directly on the `DisplayMessageTask` object. Instead, we create `Thread` objects with the `DisplayMessageTask` object as a parameter, and we call `.start()` on that thread object. The code above further illustrates that we can use pass in lambda expressions to instantiate `Thread` objects just fine.

Daemon Threads

We can further divide threads into two categories based on the duration for which the thread is doing its job:

- A **daemon** thread has a task that is being performed for the entire duration of the program. For instance, if we boot up our computer, there are some really important tasks that always need to be worked on. he threads working on these tasks are daemon threads. In Java, daemon threads are eventually terminated by the JVM all user threads are completed.
- A thread that is not a daemon thread is called a **user** thread.

In Java, we can specify whether a thread is a daemon thread by calling `setDaemon()` before `start()`. If we don't call this method, then the thread will be a user thread by default.

Thread Scheduling

How does Java figure out which thread to execute, for how long, and when? This is done with a **scheduler**. While we aren't concerned with the specific implementation details of the scheduler, there are a couple of policies that some schedulers follow that we should be aware of:

1. One type of scheduling is **non-preemptive scheduling**. This means that once we've given a thread the chance to run, the thread can hold the CPU for as long as it wants. In our printing example, this would mean that we would never achieve the interleaving of statements since a single thread will execute until completion before giving the other thread a chance to get ahold of the CPU. So how is non-preemptive scheduling even multithreading? Pretty much, the only way in which the execution of a program can jump between threads is by the thread explicitly telling the program to give another thread ahold of the CPU (in Java, this can be done with the `.yield()` and `.sleep()` functions).

2. **Preemptive scheduling** is a scheduling discipline in which the CPU can be taken away by a thread depending on what the scheduler deems to be important. Preemptive scheduling is often achieved by assigning priorities to the incoming threads and executing the threads from highest to lowest priority.

Why should we use one type of scheduling over the other? There are pros and cons to both scheduling disciplines. In preemptive scheduling, the order in which threads are selected is indeterminate — it is completely dependent on the scheduler. However, scheduling may not be fair; some threads may execute far more often than others. Also, some threads might indefinitely block other threads if the other threads always execute first (formally, this is known as the **starvation** of threads).

Sleeping, Joining, and Interrupting Threads

Consider the following code segment:

```
public class ThreadNoJoin extends Thread {

   public void run() {
      for (int i = 1; i <= 3; i++) {
         try {
            sleep((int) (Math.random() * 3000));
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
         System.out.println(i);
      }
   }

   public static void main(String[] args) {
      Thread thread1 = new ThreadNoJoin();
      Thread thread2 = new ThreadNoJoin();

      thread1.start();
      thread2.start();

      System.out.println("Done");
   }
}
```

The `run` method above prints the numbers `1`, `2`, and `3`, while waiting a random amount of time between each printing operation. Why do we call `sleep`? Because it permits the scheduler to pick another thread. Ultimately, this permits us to see an interleaving of our messages.

It is important to keep in mind that `sleep` is a static method which means that it is shared with all instances of `Thread` objects. As a consequence, this means that the instance variable that is invoking the `sleep` method may not be the one going to sleep — the current `Thread` object executing its task will be the one to go to sleep. This is a common mistake that occurs when programmers are new to multithreading. To be more explicit, if we declare `t1` to be a `Thread` object, calling `t1.sleep(2000)` will make whichever thread is currently executing (not necessarily `t1` itself) go to sleep for 2000 milliseconds.

There is an issue with the code we just presented: we don't know when the threads are being finished. This means that the `Done` statement that we are printing our after both `start()` calls can possibly (and will probably) appear before both threads finish execution. The solution to this problem is the use of the `.join()` method, which tells the `main` function to wait until a thread finishes executing.

Consider the following modified code:

```
public class ThreadJoin extends Thread {

   public void run() {
      for (int i = 1; i <= 3; i++) {
         try {
            sleep((int) (Math.random() * 3000));
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
         System.out.println(i);
      }
   }

   public static void main(String[] args) {
      Thread thread1 = new ThreadJoin();
      Thread thread2 = new ThreadJoin();

      thread1.start();
      thread2.start();

      try {
         thread1.join();
         thread2.join();
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
      System.out.println("Done");
   }
}
```

This code will now execute how we want it to. More specifically, we know that both threads will have finished execution after our `.join()` calls, which means that the "`Done`" that gets printed will happen after both threads finish execution.

It is very important to note that all of the threads must be started at once and subsequently, all of the threads must be joined at once. If we instead wrote `thread1.start()`, `thread1.join()`, `thread2.start()`, and `thread2.join()` in that order, we would no longer be multithreading. Instead, we would just be letting `thread1` finish its task deterministically, and subsequently letting `thread2` finish its task deterministically. Just to emphasize:

> If you want to run many threads concurrently, start them all at once and subsequently join them all at once. Do NOT start one thread, call join on that thread, start the next one, call join that thread, etc.

By default, a thread ends when its `run()` method ends. But, what happens if we need to stop a thread before it ends (maybe we've created several threads to find a solution and once a solution is found, there is no need for the other threads)? In the past, there was a `stop()` method; however, it caused many problems. It is now a deprecated method, and we should not use it. Instead, we should use the `.interrupt()` method.

Here's an example that illustrates where interrupting threads might be useful:

```
import javax.swing.JOptionPane;

public class InterruptExample extends Thread {
   private int power;
```

```
   public InterruptExample(int power) {
      this.power = power;
   }

   public void run() {
      int value = 1;
      String answer;

      while (!interrupted()) {
         answer = getName() + value + " raised to power " + power + ": ";
         answer += Math.pow(value, power);
         System.out.println(answer);
         value = ++value % 100;
      }
   }

   public static void main(String[] args) {
      Thread t1 = new InterruptExample(2);
      t1.setName("FIRST-->");

      Thread t2 = new InterruptExample(3);
      t2.setName("SECOND-->");

      t1.start();
      t2.start();

      JOptionPane.showMessageDialog(null, "Press OK to stop.");
      t1.interrupt();
      t2.interrupt();

      JOptionPane.showMessageDialog(null, "Thank you for using our system.");
   }
}
```

In this example, our two threads will keep on computing powers until we click the `OK` button in our pop-up box. At that point, we will interrupt both of our threads in order to terminate them.

29 Wednesday, November 13, 2019

Synchronization

Last time, we introduced threads. To recap, recall that the minimal representation of a thread consists of a stack and a program counter. Just about everything else (including the heap) is shared by all threads. This means that if one thread creates an object in the heap, every other thread can access this object as well. Today, we will talk about the correct way to access data shared between threads since doing so in an unsafe way might corrupt the data being accessed.

In Java, there is a concept of **thread safe** classes, which means that when threads share an object of this kind, we do not have to worry about corrupting data. Thread safe classes are also commonly referred to as **synchronized classes** — synchronization in Java guarantees that no two threads can modify the object at the same time. An example of a synchronized class is the `StringBuffer` class. On the other hand, the `StringBuilder` class is **not** synchronized; it is not thread safe. In addition, the `ArrayList` and `HashMap` classes are not thread safe.

Data Races

A **data race** occurs when two or more threads in a single process are accessing the same memory location concurrently, and one of the threads attempts to modify the resource. This "single resource" can even be a single variable, like an `int` or `ArrayList`.

In order to better clarify what a data race is, consider the following example:

```
public class DataRace extends Thread {
    static int common = 0;
    public void run() {
        int local = common; // data race
        local = local + 1;
        common = local; // data race
    }
    public static void main(String[] args) throws InterruptedException {
        int max = 3;
        DataRace[] allThreads = new DataRace[max];
        for (int i = 0; i < allThreads.length; i++)
            allThreads[i] = new DataRace();
        for (DataRace thread : allThreads)
            thread.start();
        for (DataRace t : allThreads)
            thread.join();
            System.out.println(common); // may not be 3
        }
    }
```

In this example, we create a `run` method that makes a copy of a shared `int` variable into a local variable, increments the local variable, and subsequently sets the shared variable equal to the local variable after incrementation.

In the `main` method, we create three threads, call `start` on them all, and finally `join` them all. This means that all of the threads have finished executing after the `join` statements. While we would expect the the shared `common` variable to store 3 after all three threads have finished running (we increment the variable three times). However, it turns out that this is not necessarily the case.

For example, suppose Thread 1 successfully increments `common` to 1, and Thread 2 is in the process of performing its incrementation. If Thread 2 executes the statement `int local = common` and the scheduler

switches to the third thread, then Thread 3 may also execute `int local = common`. At this point, when Thread 2 finishes executing, Thread 3 will still have the old value of `common`, and the final value of `common` will be 2 rather than 3.

Surprisingly, changing the three lines of code to just `common++` won't fix the problem either — the problem will just be hidden in assembly code (it won't be as visible to us). Some operations in programming languages run completely independently of other threads (such operations are called **atomic operations**), but as we've shown, incrementation is not an atomic operation.

In order to prevent data races, we can use **locks**, which help enforce synchronization. Locks are entities that can be helped by only one thread at a time, and they help enforce mutual exclusion so that we can protect **critical sections** of codes (i.e. portions of code that we do not want more than one thread accessing at once). Threads can acquire and release locks, but only one thread can acquire a lock at a time. Other threads must wait to acquire a lock (halting execution) if the lock is held by another thread.

Here is the same code segment that we just saw but now with a lock:

```
public class DataRace extends Thread {
    static int common = 0;
    static Object lockObj = new Object(); // all threads use lockObjs lock
    public void run() {
        synchronized(lockObj) { // only one thread will be allowed
            int local = common; // data race eliminated
            local = local + 1;
            common = local;
        }
    }
    public static void main(String[] args) {

    }
}
```

In this code segment, our critical section is the portion of code in which we're incrementing `common` by one (we only want one thread to execute this portion of code at once). Thus, we are motivated create a lock in order to enforce this policy.

We declare a global variable `lockObj`, which is an `Object`. Since `lockObj` is a global variable, the object is shared between all of the threads. Next, we've placed the critical section in a `synchronized() { ... }` block, which is how we acquire a lock. If a thread begins executing the synchronized block, it acquires the lock specified, and it begins to execute the code in the code block. No other thread can acquire the lock while the thread is executing this code block. After execution, the lock is returned, and other threads are permitted to acquire the block.

It's also important not to place *everything* in a synchronized block – otherwise, we wouldn't be multi-threading at all (only one thread would be able to execute their `run` method at a time). It's also important to remember that threads can be performing different tasks. In our examples

30 Monday, November 18, 2019

More on Locks

Last time, we saw the concept of locks, which can be used to prevent data races when multithreading. In our previous example, we used Java's `Object` class to represent a lock; however, it turns out that we can actually use *any* object to represent a lock (like a `String` object, or a user-defined object). This can be helpful since we can often use the object that we only want a single thread accessing at a time as a parameter to our `synchronized` block rather than creating a new lock `Object`. This is demonstrated in the example below.

Consider the following code segment which models the bank accounts of two types of shoppers — normal buyers and excessive buyers.

Firstly, here is the `NormalBuyer` class:

```
public class NormalBuyer extends Thread {
   private Account account;

   public NormalBuyer(Account account) {
      this.account = account;
   }

   public void run() {
      System.out.println("Normal buyer about to deposit");
      synchronized(account) {
         account.deposit(999);
      }

      System.out.println("Window shopping by Normal buyer");

      System.out.println("Normal buyer is now buying");
      synchronized(account) {
         account.withdrawal(100);
      }
   }
}
```

Next, here is the `ExccessiveBuyer` class:

```
public class ExcessiveBuyer extends Thread {
   private Account account;

   public ExcessiveBuyer(Account account) {
      this.account = account;
   }

   public void run() {
      System.out.println("Excessive buyer about to deposit");
      synchronized(account) {
         account.deposit(2000);
      }

      System.out.println("Window shopping by Excessive buyer");
      System.out.println("Window shopping by Excessive buyer");
      System.out.println("Window shopping by Excessive buyer");

      System.out.println("Excessive buyer is now buying");
```

```
      synchronized(account) {
         account.withdrawal(1999);
      }
   }
}
```

As we can see, the `NormalBuyer` deposits 999, prints a statement, and it subsequently withdraws 100. On the other hand, the `ExcessiveBuyer` deposits 2000, prints a few statements, and finally withdraws 1999.

Now consider the following driver which creates one thread for the normal buyer and a second thread for an excessive buyer. Both of the buyers share the same `Account` object.

```
public class Driver {
   public static void main(String[] args) {
      Account sharedAccount = new Account();
      Thread excessiveBuyer = new ExcessiveBuyer(sharedAccount);
      Thread normalBuyer = new NormalBuyer(sharedAccount);

      excessiveBuyer.start();
      normalBuyer.start();
      try {
         excessiveBuyer.join();
         normalBuyer.join();
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
      System.out.println(sharedAccount.getBalance());
   }
}
```

Since both the `excessiveBuyer` and `normalBuyer` are sharing the `sharedAccount` object, we need to protect the object with a lock whenever we are making changes to the account balance. This is why our `run` methods have `synchronized (...) { ... }` blocks with a shared object. Note that the print statements aren't inside of the `synchronized` block since it isn't a good idea to synchronize when it isn't necessary (this allows the scheduler to switch between threads whenever it deems that to be a good choice). Ultimately, this allows us to maximize concurrency.

It turns out that there's actually an alternative solution that allows us to abstract our solution even further. This alternative solution is to move the `synchronized` block from the `ExcessiveBuyer` and `NormalBuyer` classes to the `Account` class. By doing so, whoever is using the `Account` class doesn't have to bother with synchronizing their use of `Account` objects themselves (everything is already done in the implementation of an `Account` for them). Under this alternative solution, we would be able to remove the `synchronized` blocks in our `ExcessiveBuyer` and `NormalBuyer` classes, and the `Account` class would be modified to look like this:

```
public class Account {
   private double balance = 0;

   public void deposit(double amt) {
      synchronized(this) {
         double local = balance;

         local = local + amt;
         balance = local;
      }
   }
```

```
  public void withdrawal(double amt) {
    synchronized(this) {
      double local = balance;

      local = local - amt;
      balance = local;
    }
  }

  public double getBalance() {
    synchronized(this) {
      return balance;
    }
  }
}
```

Take note of how this implementation of `Account` uses the `this` keyword in order to use itself as a lock. This is allowed, and it allows us to practice synchronization without having to introduce a new object.

To justify why this solution is better than the previous solution, suppose we want to pass our `Account` class to someone else, and they want to use the `Account` object for a variety of purposes. With this new implementation, the people who we pass our code to won't have to deal with data races themselves (the `Account` class now takes care of data races itself; the class is now thread safe).

Can we do even better? Yes — instead of using `synchronized(this) { ... }` blocks that cover entire functions, Java allows us to add the `synchronized` modifier to function headers. However, this modifier should only be used if *every* statement inside of the function needs to be protected by the lock. Here's our updated code with `synchronized` functions in practice:

```
public class Account {
  private double balance = 0;

  public synchronized void deposit(double amt) {
    double local = balance;

    local = local + amt;
    balance = local;
  }

  public synchronized void withdrawal(double amt) {
    double local = balance;

    local = local - amt;
    balance = local;

  }

  public synchronized double getBalance() {
    return balance;
  }
}
```

Common Mistakes with Multithreading

There are several things that can go wrong when using concurrency. In this section, we will discuss some of the common mistakes.

1. Using different locks for each thread can be a problem since the mutual exclusion of threads depends on threads acquiring the same lock. There cannot be synchronization if threads have different locks which means that data races may still occur. We can usually resolve this issue by making our lock object `static` and global.

2. Threads should not let go of a lock until they're completely finished with the critical section. It would be incorrect to break a single `synchronized` block into two `synchronized` blocks even if they are one after another. To demonstrate what is meant by this statement, consider the following code that is used by a thread to increment a shared variable:

```
public class DataRace extends Thread {
    static int common = 0;
    static Object lockObj = new Object(); // all threads use lockObjs lock
    public void run() {
        synchronized(lockObj) { // only one thread will be allowed
            int local = common; // data race eliminated
            local = local + 1;
            common = local;
        }
    }
}
```

   While the above code properly prevents data races, it would be incorrect to rewrite the code as follows:

```
public class DataRace extends Thread {
    static int common = 0;
    static Object lockObj = new Object(); // all threads use lockObjs lock
    public void run() {
        synchronized(lockObj) {
            int local = common;
        }
        synchronized(lockObj) {
            local = local + 1;
            common = local;
        }
    }
}
```

   As demonstrated, we must perform either all of the operations associated with a critical section at once or none of them. The process of acquiring the lock and performing all of the operations is known as an **atomic transaction**.

3. Suppose a thread holding a lock is unable to obtain the lock held by another thread and vice versa. This might occur if the thread holding the lock is waiting for an action to be performed by the other thread waiting for the lock, so it cannot let go of the lock. Such an occurrence is known **deadlock**, and in this case, the program is unable to continue execution.

31 Wednesday, November 20, 2019

Today, we'll finish up concurrency, and we'll start graphs.

Deadlock

Last time, we introduced the notion of **deadlock** — a state a situtation in which threads are blocked because each thread is holding onto a resource and waiting for another resource that has been acquired by another thread. Today, we will look at two examples of deadlock.

Firstly, consider the following deadlock situation (the code segment below isn't real Java code, but it illustrates one of the primary ways in which deadlock can occur):

```
/* Objects a and b are our locks. */
Object a = new Object();
Object b = new Object();

Thread1() {
    synchronized(a) {
        synchronized(b) {
            ...
        }
    }
}

Thread2() {
    synchronized(b) {
        synchronized(a) {
            ...
        }
    }
}
```

In this code segment, we see that there are two locks, `a` and `b`, which are to be shared between `Thread1` and `Thread2`. Furthermore, we see that there are two nested `synchronized` blocks in the code that `Thread1` and `Thread2` execute. Suppose the scheduler begins to execute `Thread1`'s code, and it enters the `synchronized(a) { ... }` block. However, suppose it is preempted by the scheduler before it can enter the `synchronized(b) { ... }` block. Now suppose the scheduler begins to execute `Thread2`'s code, and say it enters its `synchronized (b) { ... }` block. If this is the case, then we will enter a state of deadlock: `Thread2` cannot continue its execution path since `Thread1` has acquired the lock `a` (it won't be able to execute the `synchronized(a) { ... }` block), and the same goes for `Thread1` since `Thread2` has acquired the lock `b`.

Here's another example of deadlock:

```
void swap(Object a, Object b) {
    Object local;
    synchronized(a) {
        synchronized(b) {
            local = a; a = b; b = local;
        }
    }
}

Thread1() { swap(a, b); } // holds lock for a; waits for b.
Thread2() { swap(b, a); } // holds lock for b; waits for a.
```

Why might this code result in a deadlock? Suppose we begin executing `Thread1`'s code, and we enter the `synchronized(a) { ... }` block. At this point, suppose `Thread1` is preempted, and we enter `Thread2`'s code. If we enter the `synchronized(a) { ... }` block for `Thread2`, then both of our locks will have been acquired, and we will enter a state of deadlock. This issue can be fixed, however, by switching `Thread2`'s code to `swap(a, b)` because whichever thread enters the outermost `synchronized` block first will finish execution to completion.

What are some tips to avoid deadlock?

- Try to avoid acquiring many locks when they aren't needed.
- If you need to acquire several locks, acquire them in them in the same order in which you use them.

Introduction to Graphs

A **graph** consists of a set of points (called nodes or vertices), which are interconnected by a set of lines known as edges. Edges can be **directed**, which means that they can only be traversed in one direction (whichever direction they are pointing towards) or **undirected**, which means that they can be traversed in either direction.

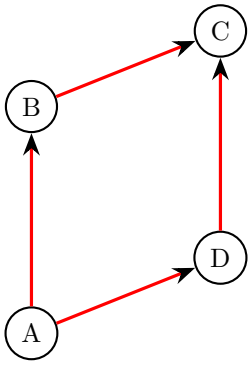


Figure 18: A Graph with Directed Edges

A graph can be **weighted**, which means that there's a cost associated with traversing each edge. This is usually represented in a graph by writing the **edge weight** corresponding to an edge directly beside the edge.

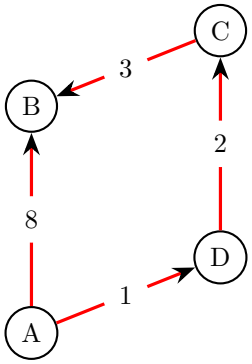


Figure 19: A Weighted Graph with Directed Edges

Weighted graphs are important since they help us model real-life scenarios. For example, suppose each vertex represents a city, and the edge weight between two vertices represents the distance between the two cities. We might be interested in finding the shortest path between two cities.

A **path** in a graph is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that an edge exists between each pair of vertices. For example, in the weighted directed graph above, A, D, C, B and A, D are paths; however, A, B is not a path.

A **cycle** is a path that starts and ends at the same starting vertex with no repeating intermediate vertices. Once again, looking at our previous graph, A, D, C, B, A is a cycle, whereas A, D isn't a cycle (but it's still a path).

A graph is called **acyclic** if there are no cycles in the graph. Here is an example of an acyclic graph:

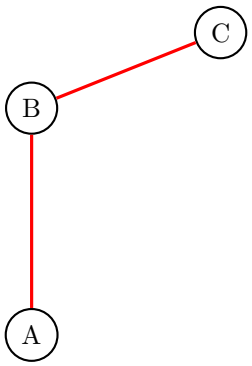


Figure 20: An Acyclic Graph

A graph is said to be **connected** if every node in the graph is reachable from every other node. A graph that is not connected is said to be **disconnected**.

Previously, we learned about trees, which are types of graphs (all trees are graphs but not all graphs are trees). The formal definition of a **tree** is an acyclic connected graph.

Graph Traversal

Earlier, we saw how to traverse trees using breadth-first and depth-first traversals. Surprisingly, these two traversals also allow us to efficiently traverse graphs. Recall the following:

- **Breadth-first search** visits from a source vertex, and it subsequently visits all vertices at distance k before visiting any nodes of distance $k + 1$. The general approach is to visit all of the neighbors of the source vertex first, and keep the list of nodes to visit in a queue. We can view this as a series of expanding circles. The actual order in which the vertices at a given distance are processed is dependent on the order in which they are stored in their Adjacency Lists.
- **Depth-first search** visits all node on a path first. It backtracks when the path ends, and it keeps a list of nodes to visit in a stack. This can be implemented by recursively calling the depth-first search function on neighbors we visit, or we can use a stack data structure to keep track of which vertices to visit next.

In both algorithms, it's important to keep track of which vertices we've already visited so that we don't end up cycling (which could result in infinite loops or stack overflows).

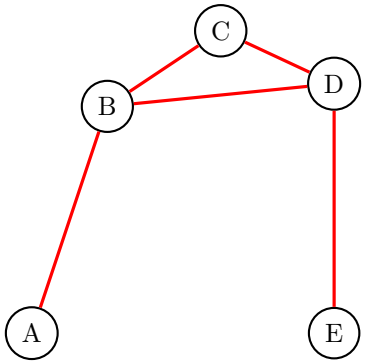
The general graph traversal procedure is essentially the same for both breadth-first and depth-first search traversals. The only major thing that we're changing is that we're extracting the next vertex to process from a queue when performing breadth-first search and a stack when performing depth-first search.

32 Friday, November 22, 2019

More on Graph Traversals

Last time, we mentioned that breadth-first search and depth-first search traversals can both be used to explore a graph.

As an example, we can trace out the two algorithms in the following graph:



First suppose we run the breadth-first search algorithm with source vertex C. At this point, vertex C enters our queue, and we mark C as "visited". At this point, we begin to process C's neighbors. We add both B and D to the queue. At this point, C has no more neighbors, so we're done processing C.

Next, we can dequeue the vertex at the front of our queue; this vertex will be processed next. Now, we will process vertex B. When looking at B's neighbors, we will add A to the queue, and we will continue processing. At this point, vertex B should be marked as visited.

On the next iteration, we dequeue D, and we add E to the queue to process. Now, D will be marked as visited.

Finally, when we dequeue A and E on the last two iterations, we will complete the iteration without processing any neighbors since there are no neighbors that have not already been visited. At this point, every vertex should be visited, so our breadth-first search traversal is complete. The order in which each vertex was processed was C, B, D, A, E.

Next, we will discuss the depth-first search traversal.

Starting with vertex C, we push B and D onto the stack. Since stacks are last-in-first-out data structures, the next vertex we will go to is D. This completes the processing of vertex C.

Next, we process the neighbors of D, and we add vertices B and E to the stack (vertex B gets added since it hasn't been marked as visited yet!). At this point, we mark vertex D as visited, this completes the processing of vertex D.

Third, we go to vertex E; however, it has no non-visited neighbors to process. Thus, we exit immediately, and we go to the next vertex at the top of our stack (vertex B) to visit.

When processing vertex B, we add vertex A to the top of our stack. At this point, vertex B has been fully processed.

Finally, we process vertex A. However, this vertex has no non-visited vertices to process, so we mark A as visited, and we have finished this iteration. At this point, the remaining entries in our stack are all visited, so our algorithm terminates. The depth-first search traversal is complete.

33 Monday, November 25, 2019

Dijkstra's Algorithm

x

34 Monday, December 2, 2019

Today, we will talk about several sorting algorithms and their properties. In this class, we aren't too concerned with how to implement these algorithms. We are more concerned about how each algorithm works at a high level, and what properties each sorting algorithm has.

Properties of Sorting Algorithms

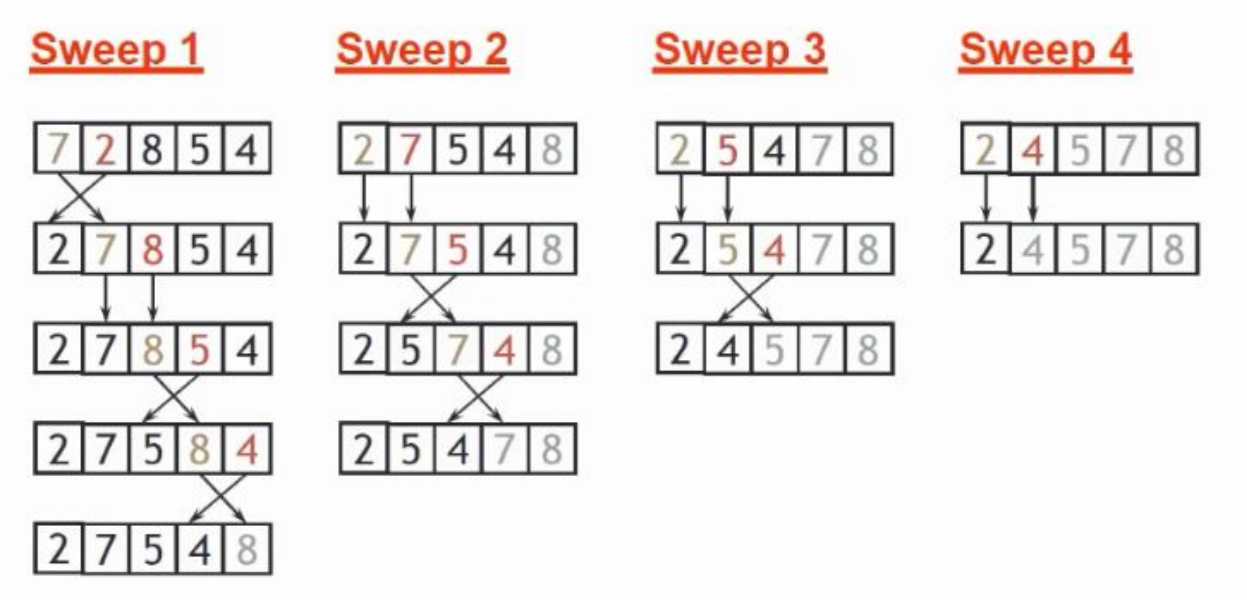
A **sorting algorithm** is an algorithm that puts the elements of a list into a predetermined order. The ordering of the elements is typically based on the key for each element, and the correctness of an algorithm is derived from the ability to compare two keys by size (e.g. if we're sorting a list of `int` types in ascending order, we know that 1 should come before 2 since $1 < 2$).

Below are some properties of sorting algorithms that we usually care about:

- A sorting algorithm is said to be **stable** if the relative order of equal keys remain unchanged. In order to demonstrate what this means, suppose we want to sort the array `[1, 3, 5, 2, 3, 4]` in ascending order (so the output should be `[1, 2, 3, 3, 4, 5]`). If we were to color the first 3 in the original array (the leftmost one) red and the rightmost 3 blue, then a stable sorting algorithm would keep the red 3 to the left, and it would keep the rightmost 3 to the right. While we clearly can't tell the difference in this example, sometimes the items we're sorting have an additional property which makes it easy to tell (one instance could be if we are sorting coordinates of the form (x, y) based on their x-coordinates).
- A sorting algorithm is said to be **in-place** if it only uses a constant amount of additional space. For instance, a sorting algorithm that creates a new array as a part of the algorithm is not in-place. The amount of additional space should not be a function of the size of the array inputted.
- A sorting algorithm is called **internal**, which means that all of the data we are dealing with is in the same memory we are working with. All of the sorting algorithms we will see in this class are internal; however, it is good to note that **external** sorting algorithms exist (these are sorting algorithms that require secondary storage devices, like disk arrays).
- A sorting algorithm is called **adaptive** if it takes into account whether the data is already sorted or near-sorted. These type of algorithms perform better and run faster when the data is already sorted or almost sorted. Adaptive algorithms are able to do less work by recognizing that the data they are processing is near the desired solution. Non-adaptive algorithms will carry out the same procedure they always perform, even when the algorithm is already sorted.
- An algorithm is **comparison-based** if it only relies on the comparison of pairs of numbers and not any other additional information (such as what is being sorted, or the frequency of each number). It has been shown that $\Omega(n \log(n))$ is a lower bound on the runtime of comparison-based sorting algorithms. There are some $\mathcal{O}(n)$ sorting algorithms that exist, but they are not comparison-based sorting algorithms, and they all assume some information about the numbers being processed.

Bubble Sort

The first algorithm that we will discuss is Bubble Sort. Bubble sort is an algorithm that iteratively sweeps through shrinking portions of the list, and it swaps element `x` with its right neighbor if `x` is larger. The following diagram illustrates how the algorithm works:



Bubble sort runs in $\mathcal{O}(n^2)$ in the best case and the worst case.

While we are not too concerned about the implementation of algorithms in this class, Bubble Sort is fairly simple to implement, so is it presented below:

```
void bubbleSort(int[] a) {
    int outer, inner;

    for (outer = a.length - 1; outer > 0; outer--) {
        for (inner = 0; inner < outer; inner++) {
            if (a[inner] > a[inner + 1]) {
                swap(a, inner, inner + 1);
            }
        }
    }

    void swap(int[] a, int x, int y) {
        int temp = a[x]; a[x] = a[y]; a[y] = temp;
    }

}
```

Selection Sort

The selection sort algorithm works by iteratively sweeping through shrinking portions of our list, selecting the smallest element found in each sweep, and swapping the element with the front of the current list. The image below demonstrates how the algorithm works:

Like Bubble Sort, Selection Sort also runs in $\mathcal{O}(n^2)$ time in the best case and the worst case. One possible implementation is selection sort is as follows:

```
void selectionSort(int[] a) {
    int outer, inner, min;

    for (outer = 0; outer < a.length - 1; outer++) {
        min = outer;
        for (inner = outer + 1; inner < a.length; inner++) {
            if (a[inner] < a[min] {
                min = inner;
            }
        }
        swap(a, outer, min);
    }
}
```

Insertion Sort

Insertion sort is very similar to the method that people typically use when they're sorting cards. Although this algorithm runs in $\mathcal{O}(n^2)$ worst case time, it is the preferred algorithm when data is nearly sorted or when the size of the data to sort is small (due to the low overhead). This algorithm has a simple implementation, and it's more efficient in practice than most simple quadratic sorting algorithms.

Insertion sort works by iteratively grows a sorted output list by removing one element from the data, finding its correct location, and placing it there. This happens until no input elements remain.

Once again, we are not too concerned with implemenetation details. This is a good illustration of how insertion sort works: https://upload.wikimedia.org/wikipedia/commons/9/9c/Insertion-sort-example.gif.

Insertion sort's best case runtime is $\mathcal{O}(n)$. One instance in which this best case behavior is exhibited is when the input list is already sorted. Moreover, since insertion sort performs better on already-sorted lists, we can conclude that insertion sort is an adaptive algorithm as well.

Tree Sort

The tree sort algorithm is fairly simple from a high-level perspective. We insert all of the elements into a binary search tree, and we perform an in-order traversal on the tree. The order in which we process the nodes is also the order that the nodes should be in the final output array. This algorithm runs in $\mathcal{O}(n \log(n))$ time in the average and worst case, and it runs in $\mathcal{O}(n)$ time in the worst case (when we make a degenerate tree).

Mergesort

Mergesort is more complicated than the sorting algorithms we've seen so far. The general approach to performing mergesort is as follows:

1. Partition the list of elements into two lists (the two lists should be equally-sized or off by one).
2. Recursively sort both lists.
3. Given the two sorted lists, merge them both into one sorted list. This is done by iteratively examining the head of both lists, and moving the smaller to the end of the new list.

The runtime of this algorithm is $\mathcal{O}(n \log(n))$ in the average case and the worst case. When we use Java's `Collections.sort`, Mergesort is the algorithm that's used. This algorithm can be implemented externally or internally, and it uses additional memory to perform the merging step (so the algorithm is NOT in-place). Mergesort is stable.

The heart of the MergeSort algorithm is demonstrated through this code:

```
/* x denotes the lower end of the array region to be sorted, and y denotes the upper end of the array
    region to be sorted. */
void mergeSort(int[] a, int x, int y) {
    int mid = (x + y)/2;
    if (x != y) {
        mergeSort(a, x, mid);
        mergeSort(a, mid + 1, y);
        merge(a, x, y, mid);
    }
}

void merge(int[] a, int x, int y, int mid) {
```

```
    /* This function merges two adjacent sorted lists in an array. Implementation not shown. */
}
```

35 Wednesday, December 4, 2019

Today, we'll continue looking at sorting algorithms.

Quicksort

The quicksort algorithm works by repeatedly picking "pivot elements" and partitioning the set of elements into three disjoint sets: the set of elements strictly less than the pivot, the set of elements equal to the element, and the set of elements strictly greater than the pivot element. We can subsequently place the elements less than the pivot to the left of the pivot, and the elements greater than the pivot to the right of the pivot. Ultimately, this results in an array in which the pivot is placed in the correct position (but the array isn't necessarily sorted). Next, we recursively call the quicksort algorithm on the elements that are less than the pivot, and the elements that are strictly greater than the pivot. Finally, we concatenate the three lists, and we're done.

Quicksort runs in $\mathcal{O}(n^2)$ in the worst case and $\mathcal{O}(n \log(n))$ in the best case.

Radix Sort

Radix sort is the first algorithm that we will discuss that is not a comparison-based sorting algorithm. The algorithm decomposes a key C into several components $C_1, C_2, \ldots, C_n$ (these could be digits if we're sorting integers or characters if we're sorting strings). Each possible component is assigned a bucket (for example, we could have 10 buckets for each of the 10 possible digits if we were sorting integers), and we distribute values into buckets according to the rightmost digit, second to rightmost digit, up until the first digit. However, it is very important to make sure that the distributing procedure that we are using is stable — each bucket must retain the order in which it receives values.

To demonstrate, suppose we wish to sort the array of numbers `[122, 397, 220, 017, 512]`. In order to do so, we first create 10 buckets $B_0, B_1, \ldots, B_9$ where bucket B_k stands for the bucket for digit k.

- First, we distribute the values into buckets based on their rightmost digits. This means that `220` gets placed into B_0, `122` and `512` get placed into B_2, and `397` and `017` get placed into B_7. The remaining buckets remain empty.
- Next, we distribute values into buckets according to the middle digit. B_1 now contains `512` and `017`. B_2 now contains `220` and `122`. Finally, B_9 contains `397`. All other buckets remain empty.
- Finally, we distribute the values into buckets according to their leftmost digit. At this point, B_0 contains `512`, B_1 contains `122`, B_2 contains `220`, B_3 contains `397`, and B_5 contains `512`. Our integers are now sorted; we can merge all of the buckets if we desire.

If each integer has d digits and we're sorting an array of n numbers, then this algorithm runs in $\mathcal{O}(d \times n)$ time. When d is fixed and much smaller than n, this reduces to $\mathcal{O}(n)$. Radix sort has some disadvantages since the number of buckets depends on the number of possible components of a key (10 buckets for integers, 26 buckets for words).

Algorithmic Paradigms

Many algorithmic problems can be subdivided into two categories: **satisfiability** problems (i.e. "is there a path from A to B?") and **optimization problems** (i.e "what is the shortest path from A to B?"). Today, we will discuss **algorithmic paradigms**, which are general approachs to solving problem. These paradigms typically have two different structures: an **iterative** structure (in which we execute actions in a loop), or a **recursive** structure (in which we reapply actions to subproblems).

Backtracking Algorithms

Backtracking is a general algorithm that incrementally builds candidates to the solutions and abandons candidates as soon as the algorithm determines that the candidate cannot possibly be completed to a valid solution. Backtracking is based on the depth-first recursive search algorithm, and such algorithms are generally structured like this:

1. Test whether a solution has been found; if it has, then return the solution.
2. Otherwise, for each choice that can be made, make that choice and recur. If recursion returns a solution, return it.
3. If no choices remain, return failure.

One way we might implement a method to check whether a path exists from vertex A to vertex F is with the following backtracking algorithm: Start at the vertex A. If the current vertex has an edge to F, return the path. Otherwise, select a neighbor node to be the current node, and recursively find a path from X to F. Return false if there is no path from any neighbor of X.

Divide and Conquer

The divide and conquer approach to solving problems simply divides a large problem into smaller subproblems. These subproblems must be of the same type, and the subproblems do not necessarily need to overlap. We solve each subproblem recursively, and we ultimately combine the solutions to solve the original problem.

Typically, divide and conquer algorithms contain two or more recursive calls; however, this is not always the case.

Mergesort and quicksort are examples of divide and conquer algorithms (we partition the array into two parts and recursively call our function on these two subarrays).

Dynamic Programming

Dynamic programming is an algorithmic paradigm that is very similar to divide and conquer; however, there are a few subtle differences. This paradigm is based on remembering past results and constructing solutions based on overlapping subproblems.

The general approach to solving a problem with dynamic programming is as follows:

1. Divide the problem into smaller subproblems. Like in a divide and conquer solution, these subproblems must be of the same type. However, unlike divide and conquer solutions, these subproblems must overlap.
2. Solve each subproblem recursively. If this subproblem is identical to a problem we've previously solved, this might just consist of looking up a solution from a table.
3. Combine solutions to solve the original problem, and store the solution to the problem.

Dynamic programming is often used to solve optimization problems (i.e. counting the number of solutions, or finding a maximum or minimum).

Dijkstra's algorithm is an example of dynamic programming since we store the shortest path for later use. Another example is of dynamic programming can be a Fibonacci number calculator, which stores previously computed results.

Greedy Algorithms

A greedy algorithm is a problem-solving strategy that uses the heuristic of making the locally optimal choice at each stage with the intent of finding the best global solution.

While greedy algorithms are usually easy to implement, they often fail to find the correct solution. For example, suppose we live in a country which has a coin system consisting of 30 cent coins, 20 cent coins, 15 cent coins, and 1 cent coins. If we want to find the minimum number of coins to get to a number, one approach might be the greedy approach in which we take the highest number that still fits into our number.

For example, if we are trying to make 30, the greedy algorithm would produce the correct answer: we would take one 30 cent coin, and we'd be done. However, in this problem, the greedy algorithm does not always produce the correct answer — if we were trying to make 40 cents, then the optimal solution would be to take two 20 cent coins. But the greedy solution will take one 30 cent coin, and ten 1 cent coins, which is clearly not optimal.

A greedy algorithm that attempts to find the shortest path between two vertices would always pick the lowest-cost neighbor. However, this isn't guaranteed to be the best solution either.

Kruskal's algorithm and Prim's algorithm can be used to find the minimum spanning tree of a graph. These are algorithms in which which greedy solutions do work. While these are algorithms were not covered in this class, they demonstrate that we cannot always rule out greedy solutions.

36 Friday, December 6, 2019

More Algorithm Strategies

Brute-force Algorithms

A brute-force algorithm is exactly what it sounds like — it tries all possible candidates in order to find a solution. The general outline to a brute-force algorithm is as follows:

1. Generate and evaluate possible solutions until one of the following conditions is true:
2. A satisfactory solution is found.
3. A best solution is found.
4. All possible solutions have been exhausted (in this case, we return the best solution, or return failure if there is no satisfactory approach).

As one may expect, brute-force solutions are typically the most expensive approach to solving problems.

While brute-force algorithms are computationally expensive, they are still important to study since they guarantee a correct solution (if any), and there are sometimes no better solutions available.

One instance in which a brute-force algorithm is used is the **travelling salesman problem**, which is stated as follows:

> "Given a weighted undirected graph, find the lowest cost path visiting all of the nodes once."

No polynomial-time algorithm exists to solve this problem, and the brute-force algorithm which runs in $\mathcal{O}(n!)$ time is a costly but correct way to solve this problem.

Branch and Bound Algorithms

The branch and bound algorithm design paradigm systematically enumerates candidates of the state space, and it tracks the best current solution found. Subsequently, when examining a new solution, it eliminates ("prunes") partial solutions that cannot improve upon the best current solution.

As an example, suppose we are finding the shortest path between two vertices A and B in a graph. The brute-force algorithm would enumerate all possible paths from A and B and pick the minimum cost path among all of these paths. However, a branch and bound algorithm would improve on this algorithm by keeping track of the current shortest path and pruning any partial paths whose cost already exceeds the best solution.

Branch and bound algorithms are good since if the solution is found early, the amount of searching necessary can be greatly reduced. However, they can be bad in the worst case where we don't find the best solution until we've almost entirely exhausted the search space.

Heuristic Algorithm

A heuristic algorithm is a technique that is typically used for solving a problem quickly when classical methods are far too slow. Heuristic algorithms often give us good approximations when classical methods fail to find any exact solution.

Heuristic algorithms are based on trying to guide the search for a solution (heuristic $\implies$ "rule of thumb"). Pretty much, we use our outside knowledge of the problem in attempt to find an approximate solution to a problem.

Heuristic algorithms are good since they typically outperform brute-force algorithms. However, they can be bad in the sense that they sometimes return *approximations* rather than exact solutions.

Effective Java

At this point, we're done with all of the course material. To wrap everything up, we will discuss some important Java programming conventions to keep in mind. These patterns to emulate and pitfalls to avoid come from [1].

1. Always make sure you satisfy Java's `hashCode` contract. This becomes a concern whenever you override the `equals` method. If the `hashCode` contract is not satisfied, any classes that use hashing (e.g. maps and sets) may exhibit undefined behavior.
2. What does `System.out.println('H' + 'a')` print? Somewhat surprisingly, it doesn't print out "Ha" – it prints out 169. This happens because `'H' + 'a'` is evaluated as an `int` (the sum of their ASCII values), and subsequently converted to a string. In order to avoid this, we must use string concatenation with care. At least one of the operands must be a `String` to get what we want.
3. What does `System.out.println(2.00 - 1.10)` print out? The answer is 0.89999999. Since decimal values can't be represented exactly in computers, there is a small margin of error when dealing with floats. Try to use `BigDecimal`, `int`, or `long` types whenever floating point values aren't needed.
4. Favor immutability over mutability, composition over inheritance, and interfaces over abstract classes. Always override the `toString` method when implementing classes in order to make your classes more pleasant to use and to make debugging easy.
5. Prefer lambdas to anonymous classes, and omit the type of lambda parameters unless their presence improves the program's clarity.
6. When implementing methods, check parameters for validity . Use overloading judiciously, and return zero-length arrays over null.
7. Prefer primitive types to boxed primitives. Avoid floats and doubles if exact answers are required. Adhere to generally accepted naming conventions. Refer to objects by their interfaces.
8. Use exceptions only for exceptional conditions. Use checked exceptions for recoverable conditions and run-time exceptions for programming errors. Favor the use of standaard exceptions, and document all exceptions thrown by each method.
9. Avoid duplicate object creation. Try to reuse existing objects instead. This improves clarity and performance. In loops, the savings can be substantial. When writing an immutable class, don't provide any mutators. Ensure that no methods may be overriden by defining classes to be `final`, fields to be `final`, and fields to be `private`.
10. Make classes immutable unless there's a good reason not to.

References

[1] Bloch, Joshua. Effective java. Pearson Education India, 2016.