Interfaces + static modifier

Advanced Java Programming 2012-13

OO Design Patterns and Principles

Lecture Outline

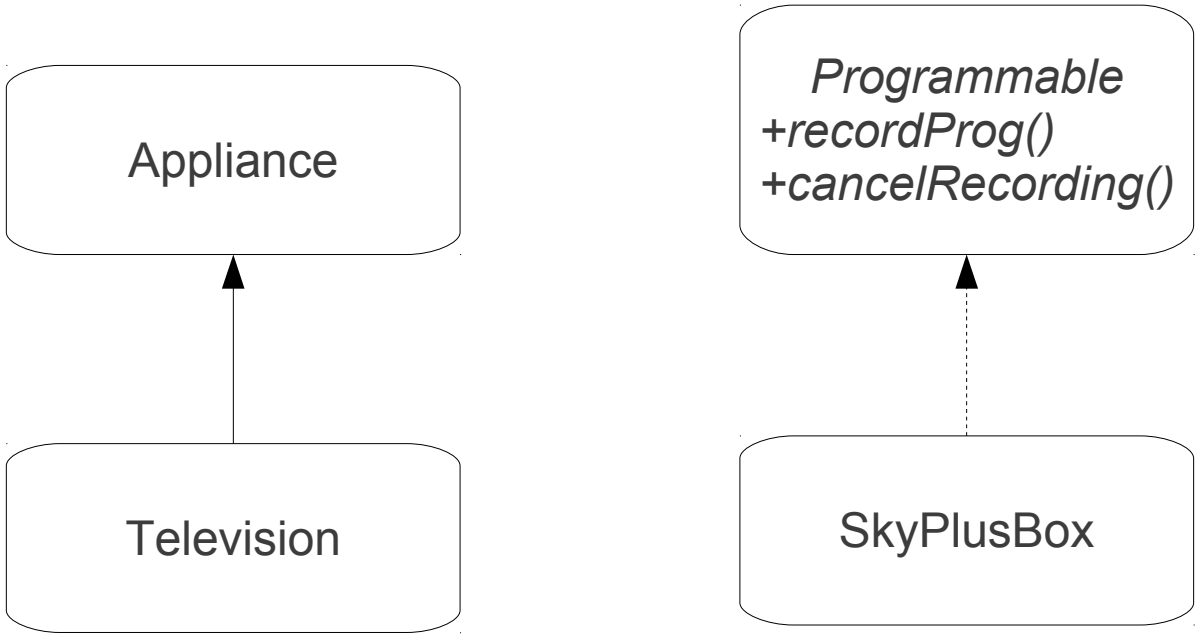
- Interfaces
  - What is an interface?
  - How do I declare an interface?
  - Methods and variables in interfaces
  - Why are interfaces useful?
  - Considerations when modifying interfaces
  - Interfaces in the JDK
  - Why are we covering interfaces now?
- The static modifier
  - Why do we need static variables/methods?

Interfaces

- What is an interface?
  - An interface specifies what an implementing class can do, not how it should do it
- You might write an interface called *Programmable* that contains the empty methods *recordProg*() and *cancelRecording*()
  - Any class implementing this interface (e.g. class TiVo, class SkyPlusBox, class V+) must agree to write the code for the *recordProgram*() and *cancelRecording*() methods
  - The actual details of the implementation will differ for each implementing class due to hardware, software etc.

Interfaces

- In UML, the relationship between an interface and an implementing class is shown using a dotted arrow
  - Use italics on the interface name
  - Include methods and instance variables as normal



Class and superclass

Implementing class and interface

Interfaces

- **Where have we used interfaces before?**
  - You used the *ActionListener* interface when you were listening for button presses
    - Any class implementing *ActionListener* had to have an *actionPerformed*() method
  - We used the *MouseListener* interface when we were listening for mouse actions
    - Any class implementing *MouseListener* had to have methods named *mouseEntered*(), *mouseExited*() etc.
  - Failure to provide these methods resulted in a compile time error

Interfaces

- **How do I declare an interface?**
  - An interface must be defined with the keyword *interface*

    `e.g. public interface Programmable {…}`
  - An interface is a 100% abstract class!
  - It has no implementation code whatsoever
  - Compare this with an abstract class, which can contain a mix of abstract and concrete methods
  - Every interface is implicitly abstract, and treated as such whether you label it or not

    `e.g. public abstract interface Programmable {…}`
  - An interface has package level access by default, public if you explicitly declare it

Interfaces

- When defining the methods of an abstract class there is no need to include empty parenthesis (curly brackets)
- There is also no need to declare the methods as abstract – again, this it is implicit
  - The methods of an interface are also implicitly public
  - An interface with private abstract methods does not make sense

```
public abstract void cancelRecording();
```

Interfaces

- The following method declarations in an interface are identical
  - `void Foo();`
  - `public void Foo();`
  - `public abstract void Foo();`
- The following method declarations would not compile
  - `final void Foo();` // final is the opposite of abstract
  - `private void Foo();` // makes no sense, also protected
  - `static void Foo();` // Static methods (e.g. println()) must have bodies

Interfaces

- All variables declared in an interface are implicitly public static and final
- This means that interfaces can only declare constants, not instance variables
- Any class implementing the interface has access to the variable, just as if the class had inherited it
- Obviously, any attempt the change the value of such a variable in an implementing class will generate a compile time error

```
public static final int ageRestriction = 15;
```

Interfaces

```
interface Programmable
{
   void cancelRecording();
   void recordProg()
}

class SkyPlusBox implements Programmable
{
  public void cancelRecording() {...};
  public void recordProg(){...};
}
```

Interfaces

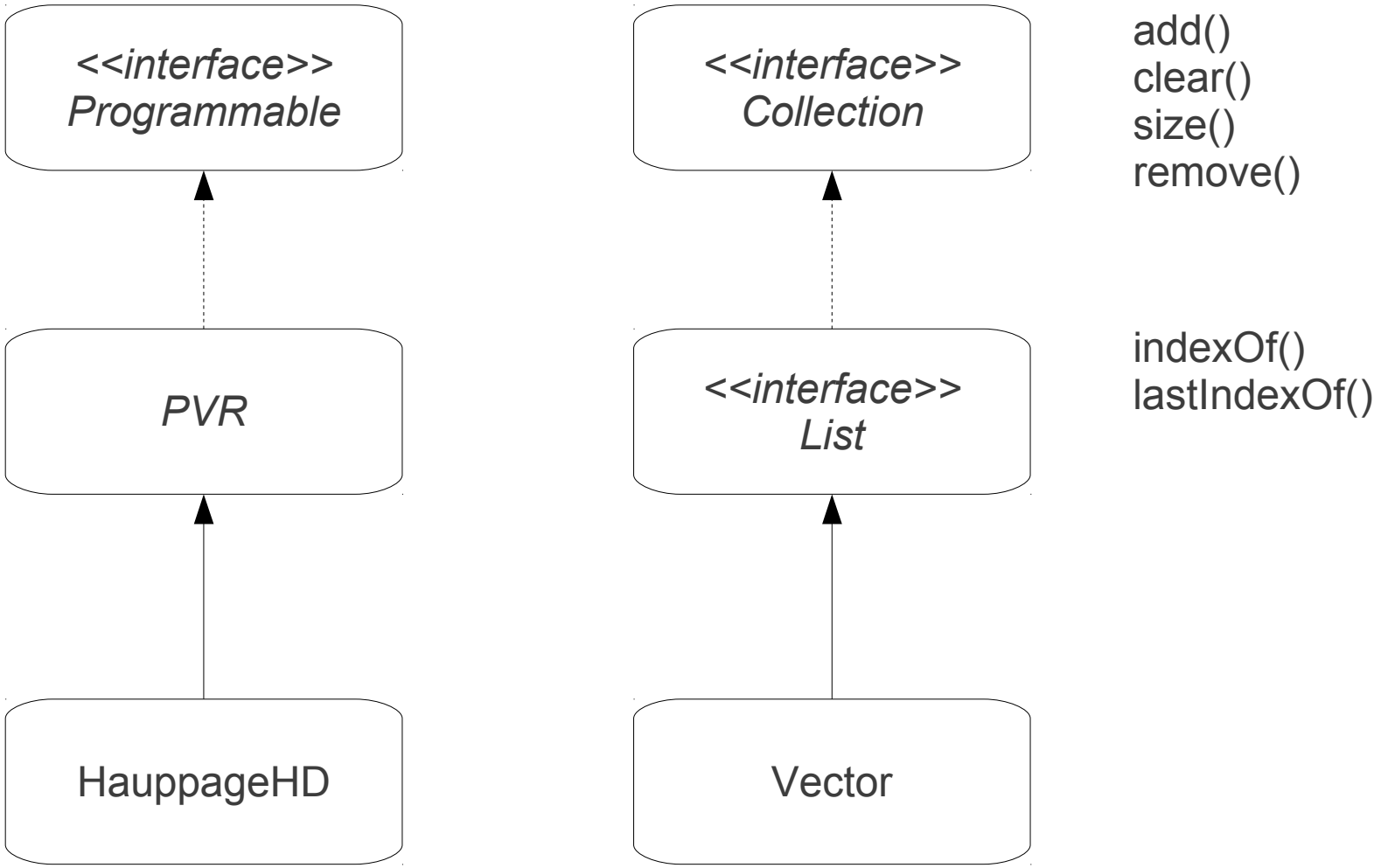
```
interface Programmable
{
   boolean unicodeCompliant = false;
   void cancelRecording();
   void recordProg()
}

class SkyPlusBox implements Programmable
{
   public void cancelRecording() {…};
   public void recordProg(){...};
   public void init(){unicodeCompliant=true;}
}
```

Interfaces

- Other important rules
  - A class can extend only one class (no multiple inheritance) but it can implement many interfaces
  - Interfaces can extend one or more interfaces
    - For example, *ActionListener* has a super-interface called *EventListener*
  - Interfaces cannot extend a class
  - Interfaces cannot implement an interface
  - An abstract implementing class does not have to implement the interface methods (but the first concrete class in the inheritance chain does)

Interfaces



Interface, abstract implementing class and concrete class

Superinterface, interface and implementing class

Interfaces

- Interfaces are not classes. In particular, you can never use the new operator to instantiate an interface:

```
x = new Programmable(. . .); // ERROR
```

- However, even though you can't construct interface objects, you can still declare interface variables

```
Programmable p; // OK
```

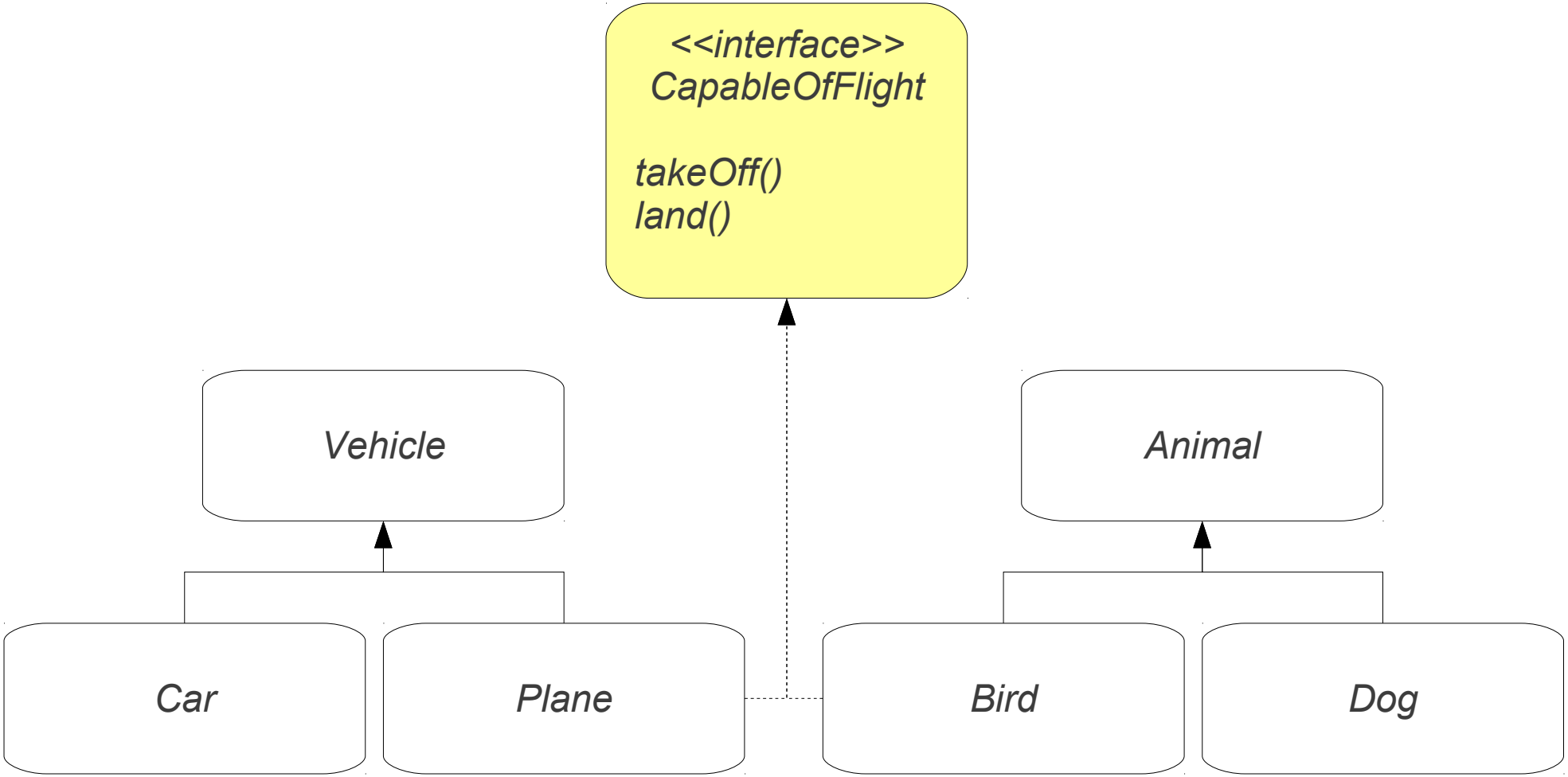
- An interface variable must refer to an object of a class that implements the interface:

```
p = new SkyPlusBox(. . .); // OK
```

Interfaces

- **Why are interfaces useful?**
- Interfaces can be implemented by any class, from any inheritance tree
- Interfaces allow us to capture similarities between classes that cannot be captured using inheritance
- For example, class *Plane* extends class *Vehicle* and class *Bird* extends class *Animal*
- The classes are radically different but share common characteristics

Interfaces



Plane and Bird objects can now be treated as things that can fly (i.e. objects on which the methods takeOff() and land() can be legally invoked)

Interfaces

- **Why are interfaces useful?**
- They can be used *polymorphically*
- Assume that you need to build a collection of objects that can fly, then invoke a shared method on each element in that collection

```
ArrayList<CapableOfFlight> a;
a = new ArrayList<CapableOfFlight>;
a.add(new Plane());
a.add(new Bird());
for (int c=0; c < a.size(); c++)
{ a.get(c).takeOff();};
```

This works because of the IS-A relationship between an interface and implementing class

Interfaces

- The *instanceof* operator can be used to test if an object is of a specified type
- An object is of type X if
  - It is an instance of class X, or a subclass of X
  - It is an instance of a class that implements X
- Use the operator in the following way
  - if (*variable* instanceof *type*)

```
String s = "Hello";
if (s instanceof java.lang.String) {
System.out.println("is a String");
}
```

Interfaces

- So, assume that p is a *Plane* object and b is a *Bird* object. *Plane* is a subclass of *Vehicle* and *Bird* is a subclass of *Animal*. *Plane* and *Bird* implement *CapableOfFlight*
  - b instanceof Bird // true
  - b instanceof Animal // true
  - p instanceof Plane // true
  - p instanceof Vehicle // true
  - p instanceof Bird // false
  - b instanceof Plane // false
  - p instanceof CapableOfFlight // true
  - b instanceof CapableOfFlight // true

Interfaces

- The polymorphic qualities of interfaces come in handy when writing methods
  - This method can process any object that implements the *CapableOfFlight* interface
  - After the method is written, new implementing classes can be safely added to the system and passed to this method e.g. class *JetPack*

```
public void trackFlyingThing(CapableOfFlight f)
{
…
}
```

Interfaces

- **Considerations when modifying interfaces**
- You develop an interface called called *DoIt*:

```
public interface DoIt
{
   void doSomething(int i, double x);
   int doSomethingElse(String s);
}
```

- Some time later, you add a third method to *DoIt*, called didItWork(). Consequences?

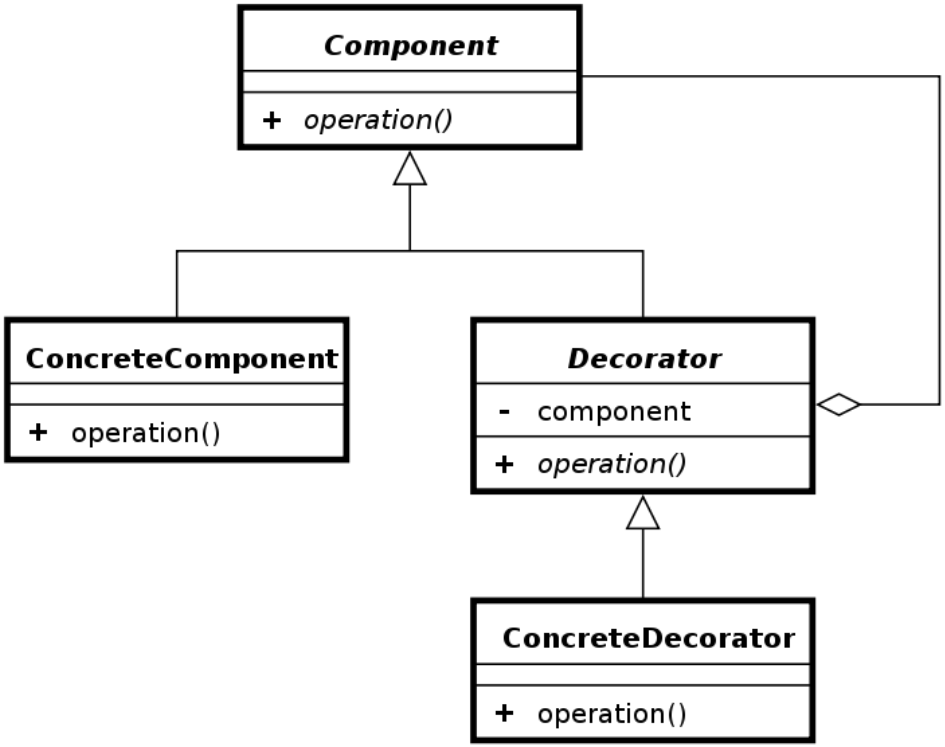
Interfaces

- All of the implementing classes will break, because they no longer implement all of the methods declared in the interface
- Solutions to this problem
  - Try to anticipate all uses for your interface and to specify it completely from the beginning (very difficult)
  - Don't change the original interface, extend it! This allows developers using the old interface to update slowly

```
public interface DoItPlus extends DoIt
{
   boolean didItWork(int i, double x, String s);
}
```

Interfaces

- **Why are we covering interfaces now?**
- Many of the design patterns we are about to study rely on the use of interfaces



The decorator pattern, one of the many design patterns that uses interfaces


The static modifier

The static modifier

- `static` is one of the 50 reserved words in Java
- In Java there are a number of access member modifiers that can be applied to classes, methods and attributes
  - e.g. public, private, protected etc.
- There are also a number of *non-access* member modifiers that can be applied to classes, methods and attributes
  - e.g. final, abstract, synchronized etc.
- The most important non-access modifier is static
- You need to understand this modifier because the first pattern we study, the singleton pattern, depends on it

The static modifier

- The static modifier can be applied to methods and variables
- It is used to create variables and methods that will exist independently of any instances created for the class
- All static members exist before you ever make a new instance of a class, and there will only be one copy of a static member regardless of the number of instances of that class
- Things you cannot mark as static
  - Constructors
  - Interfaces
  - Local variables i.e. variables without class-wide scope

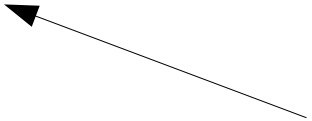
The static modifier

```
class ScopeExample
{
   int foo;

   public void doSomething()
   {
      int bar;

   }
}
```

Instance (or class-wide scope) variable.
Declared outside of method or constructor.



Local variable. Declared inside a method or constructor. Scope limited to closing brace.

The static modifier

- **Why do we need static variables?**
- Imagine you want to keep a running count of the number of instances created from class X
- Where do you store that number?
- You cannot store it as a normal instance variable in class X – every instance of X have its copy, all with identical values
- We could write some data to a text file every time we created a new instance
- But the solution is much simpler – we use a static variable

The static modifier

```
class Student
{
    int count;

    public Student()
    {
        count++;
    }

    public int getCount()
    {
        return count;
    }
}
```

In some test class, we write

```
Student s = new Student();
Student t = new Student();

int x = s.getCount(); // x=1
int y = t.getCount(); // y=1

We want X and Y = 2
```

The static modifier

```
class Student
{
    static int count;

    public Student()
    {
        count++;
    }

    public int getCount()
    {
        return count;
    }
}
```

In some test class, we write

```
Student s = new Student();
Student t = new Student();

int x = Student.getCount(); // x=2
int y = Student.getCount(); // y=2
```

The static modifier

- **Why do we need static methods?**
- Imagine you have a utility class with a method that always does the same thing e.g. it returns a random number
- It would not matter which instance of the class performed the method – it would always behave exactly the same way
  - In other words, the method's behaviour has no dependency on the state (instance variable values) of an object
- In this scenario, why do you need an object when the method will never be instance specific?
- Why not just ask the class to run the method?

The static modifier

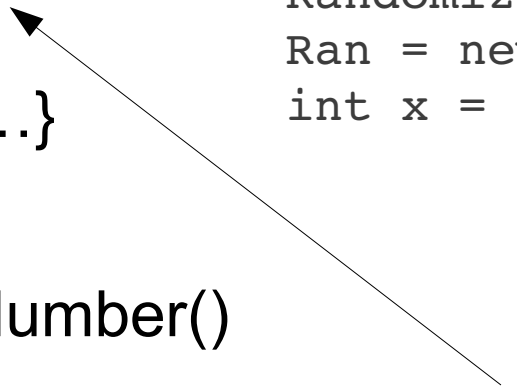
```
class Randomizer
{
    boolean b; double d; int i;

    public Randomizer() {…}


    public int getRandomNumber()
    {
        Random r = new Random();
        return r.nextInt(100);
    }
}
```

Without static methods, to create a random number we have to instance the class and then call the method.

```
Randomizer ran;
Ran = new Randomizer();
int x = ran.getRandomNumber();
```



This has an overhead. We have to create an object in memory, which includes all of the instance variables

The static modifier

```
class Randomizer
{
    boolean b; double d; int i;


    public static int getRandomNumber()
    {
        Random r = new Random();
        return r.nextInt(i);
    }
}
```

Now we can just call the static method. No instantiation = no overhead. Remember to use the class name, not an identifier

```
int x = Randomizer.getRandomNumber();
```

The static modifier

- One of the mistakes made by new Java programmers is attempting to access a (non-static) instance variable from a static method

```
class Randomizer
{
   int i = 100;
   public static int getRandomNumber()
   {
      Random r = new Random();
      return r.nextInt(i);
   }
}
```

Non-static variable i cannot be referenced from a static context

When a static method is called, there is no instantiation. No instantiation = no instance variables = compiler error

The static modifier

- This error often appears when people are working in the main method (a static method)

```
class DoStuff
{
   int x = 5;

   public static void main(String[] args)
   {
      System.out.prinln(x); doOtherStuff();
   }

   public void doOtherStuff() {....}
}
```

The static modifier

- However, a static method can access a static variable or call another static method without any problems

```
class DoStuff
{
   static int x = 5;
   public static void main(String[] args)
   {
      System.out.prinln(x); doOtherStuff();
   }
   public static void doOtherStuff() {....}
}
```

But think carefully before you do this. Try to use the class with the main method as the starting point of your application. Don't put all your code in it!

Summary

- An interface is a skeleton of a class showing the methods the class will have when someone implements it
- The fundamental reason for interfaces is to allow classes to say, in a way the compiler can check on it, “I have the behavior X”
- A static variable belongs to the class – there is only one copy of a static variable even if there are many instances of the class
- A static method can be invoked without instantiating the class
- Reading assignment
  - Singleton pattern – chapter 5 HFDP, chapter 5 DPFD
  - Observer patter - chapter 2 HFDP, chapter 4 DPFD