Lecture Six
Refence Types

Announcements

- You can use either *Skansholm* or *Bravaco & Simonson* on the exam
- Two lectures next week

Recap: Programming Principles

- Principle of Least Surprise
  - Reading/debugging code is *harder* than writing it.
  - If you use 100% of your smarts to write your programs, you are by definition not smart enough to debug it.
- DRY: Don’t Repeat Yourself
  - Code duplication? Use helper methods and/or constructor chaining!
- Single Responsibility Principle
  - A class/method should have exactly **one** responsibility!
  - Recall `Hangman` and `HangWord` from last week’s lecture

Recap: `this`

The keyword `this` refers to the current object.

```
public void setRadius(double r) {
   this.radius = r;
}

…

myCircle.setRadius(42.0);
```

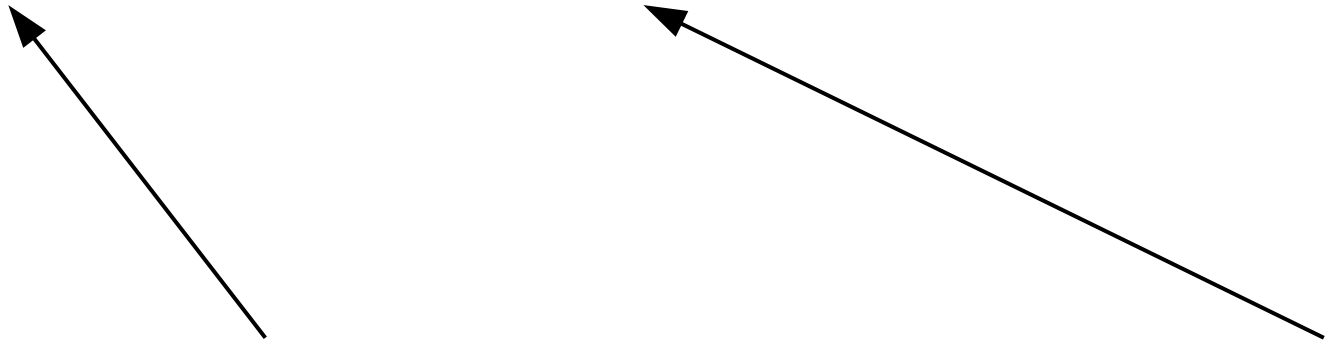
Recap: `this`

The keyword `this` refers to the current object.

```
public Circle(double r) {
   this.radius = r;
}

…

Circle myCircle = new Circle(42.0);
// myCircle.getRadius() == 42.0
```



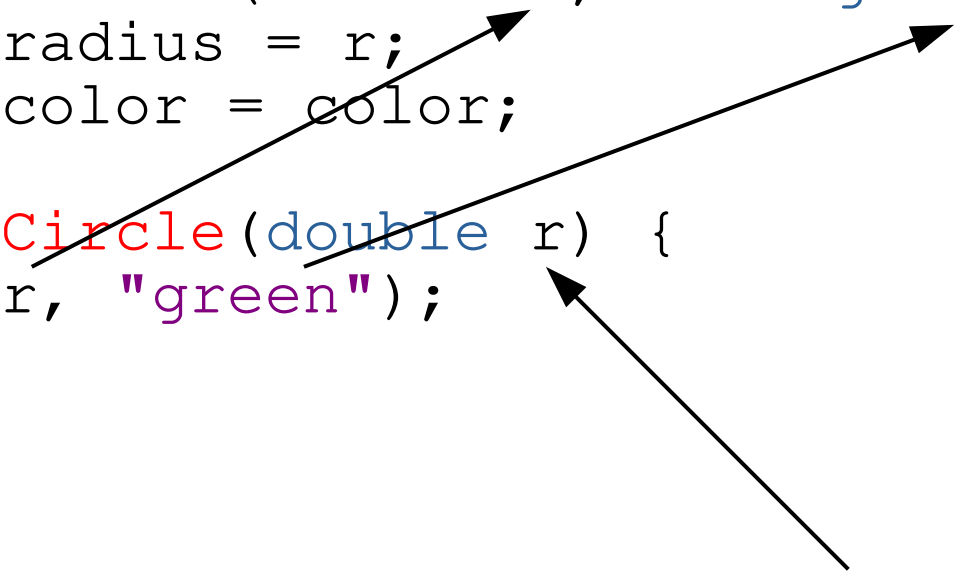
Recap: `this`

The keyword `this` can also be used to *chain* constructors

```
public Circle(double r, String color) {
   this.radius = r;
   this.color = color;
}
public Circle(double r) {
   this(r, "green");
}

…

Circle myCircle = new Circle(42.0);
// myCircle is now a green circle with radius 42
```



Recap: `toString`

- Returns a `String` representation of the object
- All classes have this method, but the default is generally unhelpful
- Always write your own when creating a new class

```
public String toString() {
  return "I'm a " + this.color +
         " circle with radius " +
         this.radius;
}
…

Circle c = new Circle(42.0, "blue");
System.out.println(c);
// prints I'm a blue circle with radius 42.0
```

New tool: `String.format`

- `String.format(format string, arg1, arg2, …)`
- Format string: plain text with *format specifiers* mixed in
- Example: `"%d is an integer and %.2f is a decimal number"`
- Format specifiers are replaced with `arg1`, `arg2`, etc.
- Specifier format: `%[width].[precision][type]`
  - Left-pad the value until it is `[width]` characters wide
  - Show decimal numbers with `[precision]` decimals (only valid for decimal numbers, not for int, string, etc.)
  - The value is a `[type]`
    - `d` = integer
    - `f` = decimal
    - `s` = string

New tool: `String.format`

- `String.format(format string, arg1, arg2, …)`
- Gives us more control than `"the value is " + x`

```
String s1 = String.format("the value is %f", 42.0);
// s1 is "the value is 42.000000"

String s2 = String.format("the value is %.2f", 42.0);
// s2 is "the value is 42.00"

String s3 = String.format("the value is %10.2f", 42.0);
// s3 is "the value is      42.00"

String s4 = String.format("%f + %d is %.1f", 42.0, 3, 42+3);
// s4 is "42.000000 + 3 is 45.0"
```

- More about formatting: https://dzone.com/articles/java-string-format-examples

`toString` with `String.format`

```
public String toString() {
   String s = "I'm a %s circle with radius %.1f";
   return String.format(s, this.color, this.radius);
}
…

Circle c = new Circle(42.0, "blue");
System.out.println(c);
// prints I'm a blue circle with radius 42.0
```

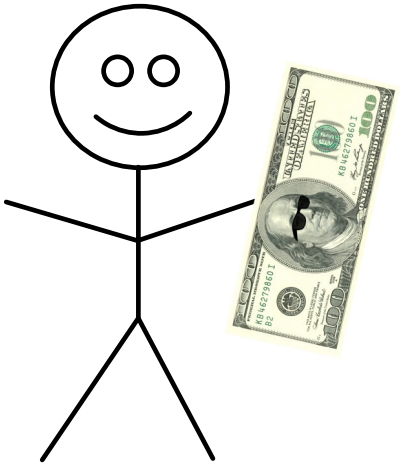
Primitive and Reference Variables

- Java's types are divided into *primitive types* and *reference types*
- Primitive types:
  - `byte`, `short`, `int`, `long`, `char`, `float`, `double`, `boolean`
- Reference types:
  - classes (including `String`), array types

Reference Variables

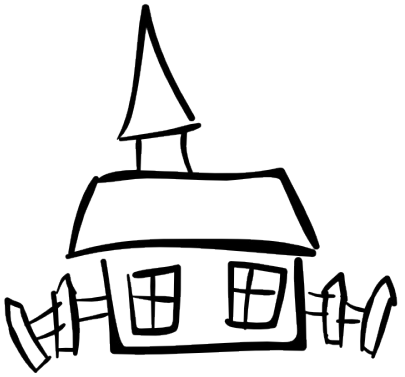
- A reference variable is a variable whose type is a reference type.
- It holds a reference to an object. (Think of this as the memory location where the object is held.)
- Two reference variables can refer to the same object!
- `x = y;` makes `x` refer to the same object as `y`. It **does not create a copy** of the object.
- `x == y` tests whether x and y **refer to the same object**. To test whether two different objects are equal, use `x.equals(y)`
- When all references to an object are destroyed, then the object is destroyed (garbage collection).
- `null` is a special reference that points to nothing.
  - Attempting to access fields or call methods on `null` always throws an exception

Primitive values are like cash



- It's *immediate*: when you have a 100 SEK bill, you *know* you have 100 SEK ready and nobody can tell you otherwise.
- It's *immutable*: your 100 SEK bill is a 100 SEK bill, regardless of the state of your bank account.
- These properties make cash and primitive types handy and safe for small transactions!

Primitive values are like cash



- It's *inefficient*: when you need a lot of money/values, you need to carry them around!
- This makes large amounts of cash or primitive values extremely inefficient for large transactions!

References are like bank accounts

- It's *indirect*: when you have 100 SEK in the bank, you must contact the bank to use it.
- It's *mutable*: just because you had 100 SEK in the bank this morning doesn't mean that you still do!
- Less safe than cash/primitive types for small transactions!

References are like bank accounts

- It's *shareable*: your whole family can share the same bank account.
- It's *efficient*: you only need to pass around the account number, regardless of how much money you want to spend.
- These properties make bank accounts and references practical for *large transactions*!

Example: A Simple Bank Account

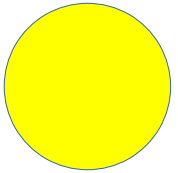
Mutability

- When we can *change* (“mutate”) a variable, it is *mutable*.

```
int price = 10;
price = price * 1.25;
```

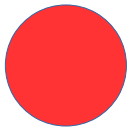
- This is often convenient, but makes programs *harder to reason about*!
- Instead, prefer to create new variables when possible.

```
int price = 10;
int priceWithTax = x * 1.25;
```



`final`

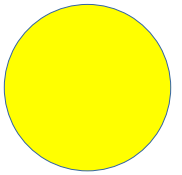
- A variable which *can't* change is *immutable*.
- We can use the `final` keyword to mark such variables.



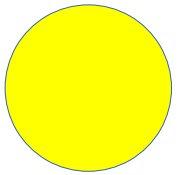
```
final int price = 10;
price = price * 1.25;
// error: cannot assign a value to final variable price
```

- This lets the compiler guarantee their immutability.
- Always mark variables intended to be immutable as `final`!

```
final int price = 10;
final int priceWithTax = price * 1.25;
```



Immutable Classes



- It is often possible to make your own classes immutable.
- Recovers the safety advantage of primitive types, while preserving the efficiency of reference types
- An immutable class:
  - Has only `final` fields
  - Has only fields of primitive types and other immutable classes
- Good candidates for immutability:
  - Pure data classes (strings, `RatNum`, vectors, database records, etc.)
- Even when the whole class can't be immutable, many fields can often be marked `final`!

`final` and constructors

- `final` fields may be assigned either where they are declared or in the object's constructor

```
public class HangWord {
    private final String word = "blah";

    public HangWord() {
    }
}
```

vs

```
public class HangWord {
    private final String word;

    public HangWord(String word) {
        this.word = word;
    }
}
```

However, in this case you should probably just make the field `static` as well, since it will have the same value in every object.

Assigning `final` fields in the constructor is generally a lot more useful.

Immutable Classes

- It's **impossible** to create an immutable array in Java!
- Java will refuse to compile the following code:

```
public class Foo {
  private final int[] values = new int[] {1, 2, 3};

  public void breakTheValues() {
    this.values = new int[] {4, 5, 6};
  }
  // Error:
  // cannot assign a value to final variable values
}
```

Immutable Classes

- It's **impossible** to create an immutable array in Java!
- However, it happily compiles the following:

```
public class Foo {
  private final int[] values = new int[] {1, 2, 3};

  public void breakTheValues() {
    this.values[0] = 42;
  }
}
```

Immutable Classes

- It's **tricky** to create immutable class in Java!
- This is also perfectly OK according to Java:

```
public class Foo {
  private final MyClass[] obj = new MyClass();

  public void breakTheObject() {
    this.obj.setSomething(42);
  }
}
```

Immutable Classes

- It's **tricky** to create immutable class in Java!
- This is also perfectly OK according to Java:

```
public class Foo {
  private final MyClass[] obj = new MyClass();

  public void breakTheObject() {
    this.obj.setSomething(42);
  }
}
```

- `final` only prevents *overwriting* variables, not *mutating their contents*!
- Always copy objects and arrays before use, if immutability is important!
  - Exception: classes you ***know for sure*** are immutable

Example: Reference Ponies and Mutability

The Method `equals()`

- Every object has a method `equals`
- This tests whether two objects have the same value.
- Java standards require that equals can take an argument of any type.
- There is a standard pattern for writing an equals method:

```
public boolean equals(Object o) {
  if (this == o) {
    return true;
  }
  if (o == null || this.getClass() != o.getClass()) {
    return false;
  }
  MyClass other = (MyClass) o;

  test whether this and other have the same value
}
```

- See *Java Direkt med Swing* section 10.12.2

The Method `equals()`

- Every object has a method `equals`
- This tests whether two objects have the same value
- Java standards rec of any type.
- There is a standard pattern for writing an e ethod:

```
public boolean equals(Object o)
  if (this == o) {
    return true;
  }
  if (o == null || this.getClass() != o.getClass()) {
    return false;
  }
  MyClass other = (MyClass) o;

  test whether this and other have th
}
```

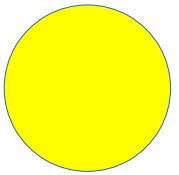
`getClass` tells you the type of an object.

Only use `getClass` if you know *exactly* why (i.e. in `equals`)!

- See *Java Direkt med Swing* section 10.12.2

Example: Pony Equality

Copying Constructor

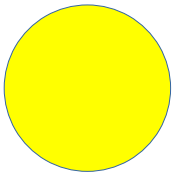


- A copying constructor creates a copy of an object.
- If your class is immutable, it should probably have one.
- Example:

```
class Circle {
  private int x;
  private int y;
  private int radius;
  …
  public Circle(Circle c) {
    this.x = c.x;
    this.y = c.y;
    this.radius = c.radius;
  }
  …
}
```

Example: Copying Ponies

Copying Constructor

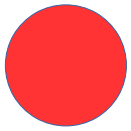


- Normally, the copy constructor needs to copy all reference variables (i.e. objects and arrays) in the object.
  - Exception: immutable objects
- This is called a *deep copy*.
- Example:

```
class Hangman {
  private int wrongGuesses;
  private HangWord word;
  …
  public Hangman(Hangman h) {
    this.wrongGuesses = h.wrongGuesses;
    this.word = new HangWord(h.word);
  }
  …
}
```

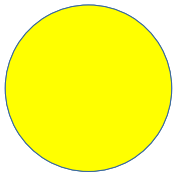
Class Variables

- Just like we can have class methods, we can have *class variables*.
- Accessible to every object of the class if private…
- ...or to the whole world, if public.
- Class variables are accessed by `ClassName.VARIABLE_NAME`.



```
class Math {
  public static double PI = 3.1415926535;
  …
}
…
double area = radius*radius*Math.PI;
String message = String.format("The area is %.2f", area);
System.out.println(message);
```

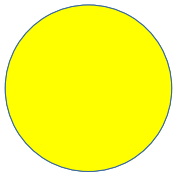
Class Variables



- Public class variables should **ALWAYS** be declared `final`!
- Private class variables should **usually** be `final`.
- `final` class variables are called **constants**, and are usually named in `ALL_CAPS_WITH_UNDERSCORES`.

```
class Math {
  public static final double PI = 3.1415926535;
  …
}
…
double area = radius*radius*Math.PI;
String message = String.format("The area is %.2f", area);
System.out.println(message);
```

Class Variables



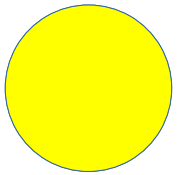
- Important part of DRY and PoLS
  - Some constants may change (you **don't** want to forget to update tax rates in half your application)
  - Most values are incomprehensible without a good name
- Use constants instead of "magic numbers"
- Good:

```
public Double getArea() {
  return Math.pow(this.radius, 2)*Math.PI;
}
```

- Bad:

```
public Double getArea() {
  return Math.pow(this.radius, 2)*3.1415926535;
}
```

Class Variables



- Make your class variables public *if and only if* the user of your class is expected to use them somehow.
  - They're useful constants in your class' problem domain (i.e. `Math.PI`)
  - They're used as input to your methods
- Otherwise make them private.

```
class Pony {
  public static final int MIN_AGE = 0;
  public static final int MAX_AGE = 30;
  …

  public Pony(int age) {
    if(age < Pony.MIN_AGE || age > Pony.MAX_AGE) {
      throw new IllegalArgumentException("bad age");
    }
    …
  }
}
```

- **Reading**

  Java Direkt med Swing sections 2.3, 2.5, 2.6, 3.3, 3.6, 3.7, 3.9, 10.12.2

- **Exercises**

  Same as last week