Inheritance

One of the key concepts of OOP
- A hierarchical relationship among classes
- Establishes "is a" subclass relationship
  - e.g. a BetterBot "is a" Bot

Benefits:
- Reusability of code
  - Put code in one class, use it in all the subclasses
  - Revisions only needs to be done in 1 place
- Polymorphic code (works on "many forms")
  - Write general purpose code designed for a supertype that works for all subtypes

Example: Bot and BetterBot

```java
public class Bot{
    private int x;
    private int y;
    public int getX() { .. }
    public int getY() { .. }
    public void eatDot(){ .. }
    public void move(){.. }
    public void turnLeft(){.. }
}

public class BetterBot extends Bot{
    public void turnRight(){
        turnLeft();
        turnLeft();
        turnLeft();
    }
}
```

Inheritance

Establishes a superclass/subclass relationship
- "is a" relationships
  - BetterBot "is a" Bot

Benefits:
- Reusability of code
- Polymorphic code (works on "many forms")

The “extends” keyword

Inheritance is established via the "extends" keyword
```java
public class Bot{
}  
public class BetterBot extends Bot{
}
```

Now we say
- BetterBot inherits from Bot
  - However, based on the visibility modifiers, certain instance variables and methods defined in Bot may not be accessible
- A BetterBot "is a" Bot
- BetterBot is a subclass/subtype of Bot
- Bot is the superclass/supertype of BetterBot
What can you inherit?

- Visibility modifiers determine which class members are accessible and which do not.
- Members (variables and methods) declared with `public` visibility are accessible, and those with `private` visibility are not.

Problem: How to make class instance variables visible only to its subclasses?

Solution: Java provides a third visibility modifier that helps in inheritance situations: `protected`.

Constructors and Inheritance

```java
public class BetterBot extends Bot{
    public BetterBot (BotWorld world){
        super(world);
    }
}
```

Another Example with super

```java
public class Dog{
    private String name;
    private int age;
    Dog(String name, int age){
        this.name = name;
        this.age = age;
    }
}
class BetterDog extends Dog{
    public Dog(String name, int age){
        super(name,age);
    }
}
```

Inheritance Tree

- Java has `single inheritance`; each node has one parent.
- Except for `Object` which has no parent.
The Object Class

All classes inherit from the Object class

- The Object class is the root of the class hierarchy
- When we create a new class, "extends Object" is implied/implicit

```java
public class Car {
}
public class Car extends Object{
}
```

The Object class has several methods which all object inherit, most notably: `toString()` and `equals()`

- Once we inherit these, we can also override the behavior i.e. make it conform to what the object of subclass will do when this method is called.

The toString() Method

- By default, it returns a String containing an object's heap address
- By convention, it is overridden to describe the object's state
- Most common usage: debugging

```java
public class Car {
    private int miles;
    private String model;
}
public class Car(String model) {
    this.model = model;
    miles = 0;
}
public String toString() {
    return "model: " + model + ", miles: " + miles;
}
```

The toString() Method contd..

- When you try to print a reference variable's value, the `toString()` method is called
- This happens behind scenes
- E.g. Dog d = new Dog();
  - In Dr Java
    > d
    Dog@a010ba
  - Is equivalent to
    > d.toString()
  - Using System.out.println(d) is equivalent to System.out.println(d.toString())

Need for `equals()`: Comparison of Strings

- If the `==` operator is used for Strings
  - Java compares the addresses where the String objects are stored, not the letters in the String
  - For example:
    > String a = "hi";
    > String b = "hi";
    > a == b
    > false
  - Use the String class' `equals` method to compare two Strings for equality
    > a.equals(b)
    > true
    > b.equalsIgnoreCase("HI")
    > true
The equals() Method

- By default, compares heap addresses
- By convention, it is overridden to match the developer’s notion of equality

```
public class Person {
    private int social; // social security #
    private String name;
}

public int getSocial() { return social; }

public boolean equals(Person p) {
    return this.social == p.getSocial();
}
```

Type Rules

- A reference variable of type t may hold a value of its own type or any subtype (but not of a supertype).
- Given the following variable declaration:

  ```java
  Bot b;
  ```

  Which of the following assignments are valid (compile)?
  ```java
  b = new Bot();
  b = new String();
  b = new BetterBot();
  b = new Object();
  ```

  How about these?
  ```java
  BetterBot bb;
  bb = new Bot();
  bb = new String();
  bb = new BetterBot();
  bb = new Object();
  ```

The “instanceof” Operator

```
> Bot b = new Bot();
> b instanceof Bot
true
> b instanceof Object
true
> b instanceof BetterBot
false

> BetterBot bb = new BetterBot();
> bb instanceof BetterBot
true
> bb instanceof Object
true
> bb instanceof Bot
true

> Bot bbb = new BetterBot(); // a variable can store a subtype
> bbb instanceof BetterBot()
true
> bbb instanceof Bot
true
```